The Pennsylvania State University
The Graduate School

IRREGULARITY-AWARE COMPUTATION AND DATA MANAGEMENT IN MANYCORE SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Xulong Tang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2019

The dissertation of Xulong Tang was reviewed and approved* by the following:

*Signatures are on file in the Graduate School.

Abstract

During the past decade, the slow down of scaling in transistor technology has brought the chip design to the "post-Moore" era, where integrating more transistors in a single core system can no longer yield performance because of the power wall and the utilization wall. As a revolutionary success, manycore systems have rapidly penetrated into various markets such as desktop, laptop, servers, mobiles, and IoT devices. The amount of resources in those manycore systems are scaling out, forming various parallel systems such as Graphics Processing Units (GPUs), manycore CPUs and heterogeneous datacenters. These systems provide huge computing capability and become the default platforms for different communities such as scientific computing, large-scale data analytics, entertainment and deep learning, where high performance, accuracy, and quality of service are of concern. However, the delivered performance rarely keeps up with the growing amount of resources. This is because of two major challenges. First, the applications' intrinsic irregularity makes them unable to utilize the resources effectively and efficiently. Second, current systems are not able to dynamically and automatically adapt to those application characteristics. Targeting these challenges, this dissertation systematically researches the opportunities existing in the software-hardware stack (i.e., compiler, runtime system, and architecture) with the goal to effectively improve performance and energy-efficiency for applications, especially for those applications with irregular computation and data access patterns. Specifically, this dissertation consists of four parts. First, focusing on irregular applications running on Graphics Processing Units (GPUs), it proposes controlled computation spawning to dynamically improve compute resource utilization and balance computation across parallel computing engines. Second, targeting poor cache performance of irregular applications, it proposes a dynamic runtime approach to exploit data reuse and improving cache locality. Third, focusing on data access parallelism, it proposes a compiler directed approach to improve memory bank-level parallelism. Finally, in addition to memory bank-level parallelism, it proposes co-optimization strategies to maximize cache level parallelism while keeping the memory bank-level parallelism maximized.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere thanks and appreciation to many people who helped me during my Ph.D. study and research. This dissertation will not be possible without all the help I received.

Foremost, from my deep heart, I would like to sincerely thank my advisor, Prof. Mahmut Taylan Kandemir, for his vigorous, professional, patient, and constant support during the past five years of my Ph.D. study. He is a very knowledgeable and professional researcher. Whenever I got confusions in the research project, his door is always open and I can find him to discuss. He also taught me how to be a good researcher and how to conduct a project from ideas to implementations and to papers.

I also want to express my sincere gratitude to my dissertation co-advisor, Prof Chita R. Das, for his motivational inspiration and guidance. He is a very nice mentor who always encourages students to think independently and systematically.

Many thanks to Prof. John (Jack) Sampson for providing me feedbacks and his guidance during my research and job hunting. I also want to thank Prof. Dinghao Wu for serving as my dissertation committee member.

I would like to thank all the lab-mates in MDL and HPCL at Penn State for their accompany during my Ph.D. study. Thanks for all the discussions and collaborations. Special thanks to Ashutosh Pattnaik, Adwait Jog, and Onur Kayiran for their help, inspired discussion, and technical collaboration.

Finally, I would like to thank my parents and my wife for providing me unconditional support.

This dissertation contains material which is copyright by ACM and IEEE.

Chapter 1 Introduction

Since the first working computer being implemented in 1946, the evolution in transistor technology (known as Moore's law) has driven the revolution of computer chip developments for over several decades, from single cycle execution to pipelining, to instruction level parallelism (ILP). Recently, there were plenty of evidences showing that, as the transistors scaling smaller, the power wall and thermal constrains prohibits the performance scaling expected from Moore's law, bringing us to the "post-Moore's law" era [2–5].

Since 2004, the focus from both industrial and academia shifted from single-core processor optimizations to multicore/manycore systems (also known as chip-multiprocessors). Since then, the amount of resources in computing systems are scaling out, forming various parallel systems such as Graphics Processing Units (GPUs), manycore CPUs and heterogeneous datacenters. These systems with huge compute capability become the default platforms adopted by various communities such as scientific computing, large-scale data analytics, entertainment, and deep learning, where high performance, accuracy, and quality of service are of concern. The fundamental design philosophy behind the manycore system is that, instead of integrating many transistors in a single socket and increasing the core frequency, multiple simpler designed cores and other parallel resources (e.g., ASICs and 3D-stacked memories) are connected through an on-chip network to provide tremendous parallelism for different application execution scenarios. For instance, Apple A11 chip is manufactured with 6 ARM cores [6] and IP cores for video processing. Qualcomm Snapdragon [7] has 4 big cores and 4 little cores targeting both high-performance execution scenarios and low-power execution scenarios. For the server-level processors, Intel Knight's Landing Xeon Phi co-processor consists of 60 cores working at 1.5 GHz [8]. NVIDIA Volta GPUs have 84 streams multiprocessors (SMs) with each SM having 64 32-bit floating point cores, providing a total of 5120 cores on a single die [9]. The tremendous parallel resources, either homogeneous or heterogeneous equipped in modern manycore systems act as fertile soil for the programmers and application designers to improve their application performance, energy-efficiency, and quality of services through parallelization and high-throughput computing.

The Problem: However, the delivered performance of applications running on manycore systems varies. For some applications, the performance improves proportionally with the scaling of the resources, whereas for other applications, having more resources does not yield any performance benefit, and sometimes even degrades the overall performance. This is because of two major challenges. First, the application's intrinsic characteristics make them unable to utilize the resources effectively and efficiently. One of these characteristics is *irregularity*. Unlike regular applications which deal with dense matrices or dense vectors. Irregular applications deal with data structures such as graphs, linked lists, etc. Most of these data structures are implemented using pointers. As a result, the *irregularity* within applications makes the runtime behaviors (e.g., computation distribution, data locality) very difficult to predict. Second, current systems are not able to dynamically and automatically adapt to those application characteristics. This brings one question onto the table: *how to effectively and efficiently exploit the parallel resources for performance and energy-efficiency?* First, different applications have different execution scenarios and hence there does not exist one-size-fit-all solutions. For example, GPUs are known to be energy efficient and performance beneficial for streaming applications with regular control flow and regular data access pattern. However, for those irregular applications, running on GPUs can lead to severe resource under-utilization or resource contention [10, 11] which consequently degrades overall performance. Second, while computation parallelization is absolutely important to the performance of manycore systems and has received substantial attention, data access parallelism also plays an important role in shaping the overall performance. Specifically, modern manycore systems generally adopt on-chip networks, such as mesh, to connect compute cores, caches, and memory. This spatial distribution of resources introduces non-uniformity in terms of data access latency. An example is that, as the memory system is portioned, memory utilization can be very low and the performance suffered in the circumstance that all data accessed are routines to the same memory partition in a short time of period.

Targeting those challenges, this dissertation adopts a holistic approach to research on how to effectively and efficiently utilize parallel resources in manycore architectures systematically. First, using GPU as a substrate, this dissertation investigates the inefficiency of irregular application executions and researches on the question of when to launch compute blocks such that the massive underlining compute resources are exploited in a balanced fashion. Second, targeting the poor data locality while running irregular applications on GPUs, this dissertation introduces a runtime hierarchical scheduler to dynamically co-locate compute blocks with intensive data reuse such that it can take advantages of the local cache hierarchy. Third, on a mesh-connected manycore system, this dissertation researches improving memory access parallelism and memory utilization by exploring the parallelism among memory banks (sub-regions of memory). This is done by analyzing the application program at compile time and reorganize/schedule the program statements at runtime. Finally, in addition to memory-level parallelism, this dissertation explores cache-level parallelism in manycore systems, and co-optimizes both memory-level parallelism and cache-level parallelism.

Contribution 1: Managing Computation in GPGPUs. GPUs are known to provide significantly high degree of parallelism and energy efficiency for applications with regular data structures. Such structured and load-balanced mapping of the computational workload facilitates efficient harnessing of the available compute throughput and memory bandwidth in GPUs. However, for many emerging data-intensive applications that work on irregular and unstructured inputs, the performance on GPU drops significantly. To better support irregular applications, NVIDIA introduced dynamic parallelism (DP) which provides applications with the flexibility to launch kernels at the GPU side without CPU intervention. Such dynamically-generated kernels can expose additional parallelism to GPU and potentially improve resource utilization. However, there are two primary drawbacks of DP. First, launching of such child kernels is not free. Aggressively launching too many child kernels can incur significant performance penalties arising from the launch overheads. Second, as each GPU core can only run a fixed number of Cooperative Thread-Arrays (CTAs) and each GPU can only execute a maximum number of concurrent kernels due to the hardware-limitations, cores can be severely underutilized. To address the aforementioned two drawbacks, we developed a runtime framework, underpinned by the observation that a better workload distribution (partitioning) between the parent and child kernels can minimize the *exposed* launch overheads and queuing latencies, while maintaining enough parallelism to improve performance. The proposed approach mitigates the aforementioned issues by dynamically controlling the launch of child kernels depending on the state of the GPU. The framework estimates the launch overhead and queuing latency, and based on this, it makes judicious decisions regarding child kernel launches. Experimental results show that our approach significantly improves the performance of the baseline dynamic parallel execution with an average speedup of 57% across 13 dynamic parallelism benchmarks.

Contribution 2: Managing Data Reuse in GPGPUs. Dynamic kernel launching in DP introduces parent-child and sibling-sibling relationships among GPU schedulable units (i.e., kernels, thread blocks, and warps). Such relationships affect the data access patterns of DP applications. For example, a parent kernel can produce the data for its child kernel before launching it, and the child kernel can consume the data. This provides us an opportunity to explore data reuse among the schedulable units based on the new relationships in DP applications. Prior techniques on GPU cache optimization involved throttling the available parallelism, bypassing the cache for certain memory requests to reduce contention, and building efficient cache management policies tuned towards GPU applications. While these techniques improve cache performance for certain applications, they cannot be ported to improve cache performance for DP applications, as they are agnostic to the on-demand kernel launch behavior. Therefore, there are four questions emerged related to data reuse and cache locality of DP applications: 1) how prevalent are intra-kernel and inter-kernel data reuses in DP applications?; 2) what is the effect of launch overhead on data reuse patterns?; 3) do neighboring CTAs have more temporal reuse or spatial reuse?; 4) is it necessary to always prioritize the child CTAs? To answer these questions, we quantitatively characterized the data reuse of dynamic applications in three different granularities of schedulable units: kernel, CTA, and warp. We observe that, for DP applications, data reuse is highly irregular and is heavily dependent on the application and its input. Thus, existing techniques cannot exploit data reuse effectively for DP applications. To this end, we first conduct a limit study on the performance improvements that can be achieved by hardware schedulers that are provided with accurate data reuse information. The limit study shows that, on average, performance improves by 19.4% over the baseline scheduler. We next propose LASER, a Locality-Aware SchedulER, where hardware schedulers employ data reuse monitors to help make scheduling decisions to improve data locality at runtime. Our experimental results on 16 benchmarks show that LASER, on average, can improve performance by 11.3%.

Contribution 3: Managing Memory Access Parallelism. To maximize the performance of multithreaded applications mapped to manycore CPUs, one needs to consider end-to-end data access performance, not just the cache performance. In fact, trying to maximize last-level cache (LLC) hit rates (which is the main goal of many compiler schemes) does not guarantee good, let alone being optimal, end-to-end data access performance. This is because off-chip accesses can consume a lot of cycles, and more importantly, latencies they experience are not uniform, being dependent on several factors such as bank-level parallelism (BLP), row-buffer locality, memory scheduling policy, etc. Targeting data access parallelism, our first strategy, built upon the inspector/executor paradigm, reorganizes LLC misses at runtime to maximize memory bank level parallelism. We evaluate the proposed approach in both simulator and real multicore hardware, and results indicate that the proposed strategy reduces execution time by 18.3% on average. Our second strategy tries to maximize both cache-level parallelism (CLP) of LLC hits and memory-level parallelism (MLP) for LLC misses. Results indicate that (i) optimizing MLP first and CLP later can bring, on average, 11.31% performance improvement over an approach that already minimizes the number of LLC misses, and (ii) optimizing CLP first and MLP later can bring 9.43% performance improvement. In comparison, balancing MLP and CLP can bring 17.32% performance improvement on average.

Contribution 4: Managing Data Parallelism. While many commercial compilers already employ a large suite of optimizations that target cache miss minimizations (e.g., loop permutation, iteration space tiling, loop fusion), the impact of these techniques is becoming increasingly limited as (i) emerging applications are processing enormous amounts of data, (ii) the increases in cache capacities are lagging far behind the increases in application data volume [12, 13], and (iii) as a result, caches are becoming unable to maintain application working sets even after aggressive cache miss minimization. Targeting emerging NoC-based manycore systems/accelerators and multithreaded workloads, in this dissertation, we propose a novel compiler framework oriented towards reducing the latencies of *both LLC hits and LLC misses*, by increasing their *parallelism.* At a high level, this is achieved by maximizing the number of accesses to distinct cache banks and the number of accesses to distinct memory banks in a given period of time.

The rest of the dissertation is organized as follows: Chapter 2 discussed the deficiency while running irregular application on GPGPUs and proposed light-weigh runtime support to dynamically control the workload distribution across GPU threads. Chapter 3 investigated the data reuse opportunities for irregular applications and proposed a hierarchical scheduling mechanism to improve the cache performance of GPGPUs. Focusing on the memory system in manycore systems, Chapter 4 introduces a software approach to improve performance by exploiting memory bank access parallelism. Chapter 5 introduces an approach to co-optimize both cache access parallelism and memory access parallelism. In Chapter 6, we summarized the related prior works and Chapter 7 draw the conclusion and future research directions.

Chapter 2 | Controlled Kernel Launch for Dynamic Parallelism in GPUs

Dynamic parallelism (DP) is a promising feature for GPUs, which allows on-demand spawning of kernels on the GPU without any CPU intervention. However, this feature has two major drawbacks. First, the launching of GPU kernels can incur significant performance penalties. Second, dynamically-generated kernels are not always able to efficiently utilize the GPU cores due to hardware-limits. To address these two concerns cohesively, this chapter proposes SPAWN, a runtime framework that controls the dynamically-generated kernels, thereby directly reducing the associated launch overheads and queuing latency. Moreover, it allows a better mix of dynamically-generated and original (parent) kernels for the scheduler to effectively hide the remaining overheads and improve the utilization of the GPU resources. Our results show that, across 13 benchmarks, SPAWN achieves 69% and 57% speedup over the flat (non-DP) implementation and baseline DP, respectively.

2.1 Introduction

Graphics Processing Units (GPUs) are known to provide significantly high performance and energy efficiency for a variety of applications from different domains, such as medical science [14, 15], finance [16, 17], social media, graphics [18], and computer vision [19]. The CUDA and OpenCL programming models allow most of these applications to naturally map thread computations to regular data structures.

Such structured and load-balanced mapping of the computational workload facilitates efficient harnessing of the available compute throughput and memory bandwidth in GPUs. However, such balanced mapping is not always possible, especially for many emerging data-intensive applications that work on irregular and unstructured inputs (e.g., graphs [20–22] and adaptive meshes [23]). Consequently, with continuously growing dataset sizes, it is becoming increasingly harder to effectively map such applications to GPUs and achieve high throughput with the desired energy efficiency [10, 24, 25].

Dynamic Parallelism (DP), supported by both CUDA [26] and OpenCL [27], is a promising feature that enables superior portability of irregular applications on GPUs. It provides applications with the flexibility to launch kernels at the device (GPU) side. In other words, if some threads are assigned higher computational workload than other threads, these threads (*parent threads*) can offload their workload by launching additional kernels (*child kernels*).Such dynamically-generated kernels can expose additional parallelism to GPU and potentially improve resource utilization [26]. However, there are two primary drawbacks of DP. First, launching of such child kernels is not free. Aggressively launching too many child kernels can incur significant performance penalties arising from the *launch overheads* [28]. Second, as each GPU core can only run a fixed number of Cooperative Thread-Arrays (CTAs[1]) [29] and each GPU can execute a maximum number of concurrent kernels due to the hardware-limits [30], cores can be severely underutilized in phases where only child kernels[2] are executing. This leads to an increase in *queuing latency* for the CTAs and kernels that cannot be scheduled due to the hardware-limits.

To address the above two drawbacks, we develop a new runtime framework, called *SPAWN*, underpinned by our observation that a better workload distribution (partitioning) between the parent and child kernels can minimize the *exposed* launch overheads and queuing latencies, while maintaining enough parallelism to improve performance. SPAWN mitigates the aforementioned issues by dynamically controlling the launch of child kernels depending on the state of the GPU. The framework

[1]A CTA is called as a "Workgroup" in OpenCL, and a "Thread-Block" in CUDA.

[2]Most of the child kernels launched are lightweight, and the CTAs associated with each child kernel can have very few warps.

estimates the amount of launch overhead and queuing latency based on the current GPU workload, and based on this, it makes judicious decisions regarding child kernel launches. If the framework decides not to launch child kernels for specific parent threads, the overhead of launching child kernels is significantly reduced. Also, as more computations are performed in the parent threads, the number of pending child kernels and CTAs reduces. Therefore, the queuing latency that is exposed substantially reduces as well. We make the following **contributions** in this chapter:

• We conduct an in-depth characterization of DP applications and quantitatively study three parameters (factors) that affect the performance of dynamic parallelism. We demonstrate that the workload distribution (partitioning between parent and child kernels) is the most significant factor that affects the performance of a dynamic parallel application. We observe that by tuning the workload distribution statically, one can achieve performance improvements ranging from 4% to as much as 8.6×.

• We propose a novel *runtime framework*, called SPAWN, which dynamically tunes the workload distribution between the parent and the child kernels. SPAWN improves the applications' resource utilization and minimizes the launch overhead and queuing latency, and therefore, improves performance.

• Experimental evaluations show that SPAWN significantly improves the performance of the baseline dynamic parallel execution with an average speedup of 57% across 13 benchmarks. It is also able to perform within 6% of the performance achieved by the best offline workload distribution. SPAWN outperforms the flat (non-DP) implementations by 69% on average, making dynamic parallelism a viable option in GPUs.

2.2 Background

In this section, we provide a brief background on dynamic parallelism (DP) and critical factors that affect its behavior.

2.2.1 Irregular Applications and DP

To help understand the inefficiencies of irregular applications running on a GPU, let us consider Breadth-First-Search (BFS) as an example. Assuming that each thread represents a vertex, threads that traverse more edges (the vertices that have high number of neighboring vertices) require more computation. Figure 2.1 shows a snippet of BFS threads. Threads T1, T5 and T7 have few edges to traverse, while the threads T3 and T6 traverse more edges. In such a scenario, when threads T1, T5 and T7 finish, a lot of compute resources are left underutilized. Clearly, the overall performance is determined by threads T3 and T6. Many other irregular applications also suffer from this workload imbalance, causing performance loss when running on GPUs [10, 31–33].

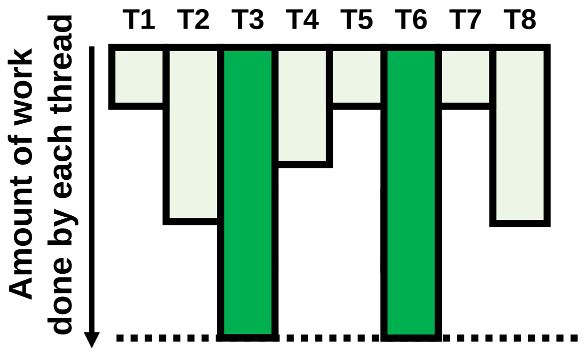
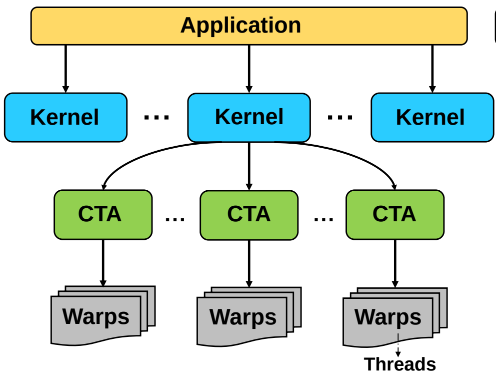
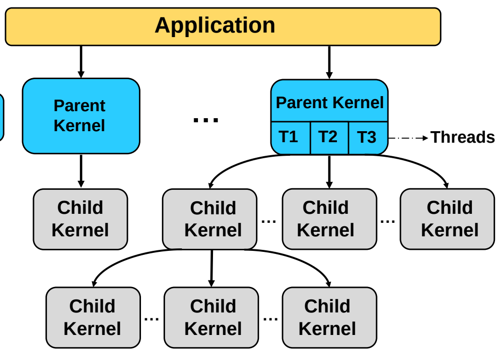


Figure 2.1: Illustrating workload imbalance in BFS.

Dynamic Parallelism (DP) is a mechanism supported by both CUDA [26] and OpenCL [27] that enables device-side kernel launches. Figure 2.2a shows the high-level structure of a conventional (non-DP) GPGPU application consisting of threads, CTAs, and kernels. A kernel contains multiple CTAs which can execute independently of each other. A CTA is a batch of threads which can communicate and synchronize with one another. The GPU hardware schedules threads into the pipeline in groups called "warps". As opposed to a conventional GPU application, a DP application can launch nested kernels from the device, as illustrated in Figure 2.2b. Each parent kernel can launch one or more child kernels. A child kernel itself can launch further child kernels and exhibit a nested launching pattern. Synchronizations are provided on device to guarantee the execution correctness. Through child kernel launches, a DP application can exploit more parallelism than its flat (non-DP) counterpart. This feature is particularly useful for irregular applications, where there can be large imbalances across the workloads assigned to different threads.



(a) Conventional application.



(b) DP application.

Figure 2.2: High-level structures of conventional GPU applications and DP applications.

2.2.2 Properties of DP Applications

To trigger device kernels, a DP application is structured differently from a conventional GPU application. Figure 2.3 is an example code fragment extracted from BFS[3]. In this figure, (a) shows the code segment executing on the CPU (host), which is agnostic of any specific DP implementation. (b) shows the implementation of a parent kernel with the ability to launch device side kernels (child kernels), and (c) shows the application code for child kernels. For each child kernel, there are three unique parameters: *THRESHOLD*, (*c_grid*, *c_cta*), and *c_stream*, shown in red in Figure 2.3b.

***THRESHOLD*:** As explained previously, if a thread has a lot of edges to traverse in BFS, spawning a new kernel from that thread can increase parallelism. To achieve this, a *THRESHOLD* is set for a parent thread to decide whether to launch a child kernel or to traverse all the edges serially. For example, if the *THRESHOLD* is set to 128, threads with more than 128 edges to traverse will launch a child kernel to perform the work. Other threads with less than 128 neighboring vertices will perform the traversal in loops (that is they will not create child kernels; instead, they will do the work by themselves in an iterative fashion). CUDA programming model allows applications to set any value as a *THRESHOLD*: a large value will result in a few heavyweight child kernels, whereas a small value will lead to a large

[3]Although the same approach is applicable to both OpenCL and CUDA, we show an implementation of BFS written in CUDA.

(a)
```
int main( int argc, char** argv){
    …
    dim3  p_grid; dim3 p_cta; /**parent kernel dimension*/
    parent<<< p_grid, p_cta>>>(type *workload); /**parent kernel launch*/
    …}
```

(b)
```
__global__ void parent (type *workload){
    int pid = blockIdx.x*blockDim.x + threadIdx.x;
    type *local_workload = workload[pid]; /**each parent threads pick up its workload*/
    if (local_workload > THRESHOLD){
        dim3  c_grid; dim3 c_cta;
        cudaStream_t c_stream;
        cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
        child<<< c_grid, c_cta, shmem, c_stream>>>(type *local_workload);}
   else
         while(local_workload){…}
      …
      cudaDeviceSynchronize(); /**waiting all children finishing*/
 }
```

(c)
```
__global__ void child(*c_workload){
    int cid = blockIdx.x*blockDim.x + threadIdx.x;
    …
}
```

Figure 2.3: Structure of BFS using DP. (a) Host code segment. (b) Parent kernel code segment. (c) Child kernel code segment.

number of lightweight child kernels. Clearly, setting a proper *THRESHOLD* value is a non-trivial task, as the value selected needs to reduce workload imbalance while avoiding significant overheads (Section 2.2.3). Most DP applications [28, 34–36] make use of a small *THRESHOLD* value.

(*c_ grid*, *c_ cta*): Another important responsibility of the parent thread is to specify the dimensions of its child kernel. *c_ grid* specifies the grid dimension in terms of the number of CTAs, and *c_ cta* specifies the number of threads per CTA. *c_ grid* and *c_ cta* capture how the workload is parallelized in a child kernel.

***c_ stream*:** The last important responsibility of the parent thread is to assign Software-managed Work Queue (SWQ) IDs to child kernels. These SWQs are called *c_ stream* in CUDA programming. Child kernels with the same SWQ ID execute sequentially. In other words, all child kernels with the same SWQ ID execute sequentially but those with different SWQ IDs can potentially execute in parallel.

An application creates a SWQ ID for each child kernel by initializing *c_stream* before launching a child kernel (lines 6 and 7 in Figure 2.3b). If the application does not specify *c_stream*, each parent CTA assigns the same SWQ ID to all its child kernels [37]. As a result, all the child kernels launched from the same parent CTA execute sequentially.

2.2.3 Hardware Architecture

The necessary architectural support for DP is shown in Figure 2.4. Similar to the traditional GPU applications (i.e., those without DP), a DP application starts running on the host (❶), and the parallel portion of the code is offloaded to the GPU through a runtime API (❷). A GPU kernel is tagged with a SWQ ID (❸), and pushed into Pending Kernel Pool located in Grid Management Unit (GMU) (❹). Kernels with the same SWQ ID are mapped into a single hardware work queue (HWQ). CTAs from a chosen HWQ's head-of-the-line kernel are dispatched to the GPU multiprocessor units (❺). The number of HWQs is 32 according to publicly-available documents from NVIDIA [30]. Therefore, the maximum number of kernels that can concurrently execute on the GPU is 32. Note that a CTA needs to wait in GMU if its required resources are not available or the hardware-limits are reached. The amount of time spent in GMU is called *queuing latency*.

Child kernels are launched through by invoking the related Runtime API function calls (❻). These API functions prepare the child kernel parameters and push the kernel into Pending Kernel Pool in GMU. Note that these API calls are asynchronous [37], and allow the parent thread to continue its execution without waiting for the child kernel to be launched. The parent thread stops and waits for its child kernels to finish only when it finishes its execution or reaches an explicit synchronization point. If an entire parent CTA is waiting for synchronization, it relinquishes the occupied GPU resources so that other CTAs can be scheduled. It is important to emphasize that full memory consistency is only guaranteed at launching point and synchronization point; DP provides weak memory consistency between the launching point and synchronization point [37].

Launching a child kernel is not free, and entails performance overheads. The time

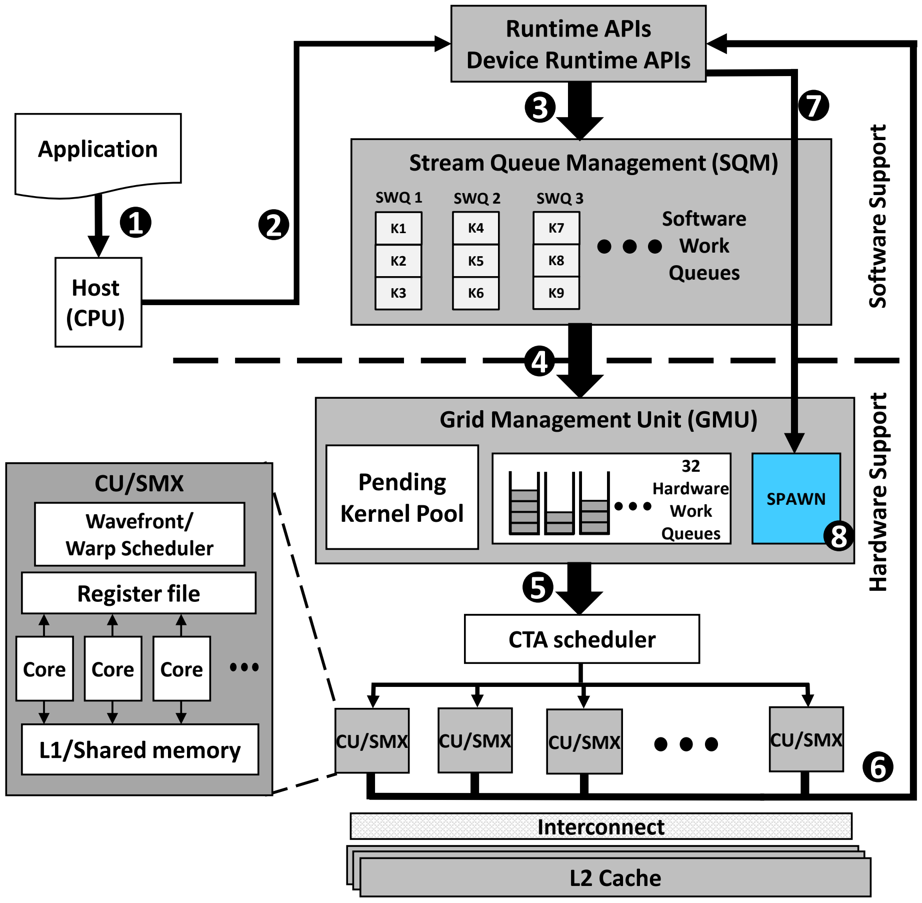


Figure 2.4: Hardware architecture realizing DP.

spent on invoking the API (❻) and pushing the child kernel into Pending Kernel Pool (❸ + ❹) is called *launch overhead*. This launch overhead can potentially be hidden by overlapping the execution of other available warps on SMXs. However, in cases where a majority of running parent threads launch child kernels within a short period of time, such high number of API calls cannot be serviced simultaneously. As a result, the resulting launch overheads can degrade performance.

2.3 Application Characterization and Motivation

In this section, we first characterize all three parameters mentioned above using our benchmarks. We observe that *THRESHOLD* is the most significant contributor towards performance since it directly controls the workload distribution between the parent and child kernels. We next show how this workload distribution can affect: 1) the launch overheads and queuing latency, and 2) the GPU utilization.

2.3.1 Benchmarks, Metrics, and Observations

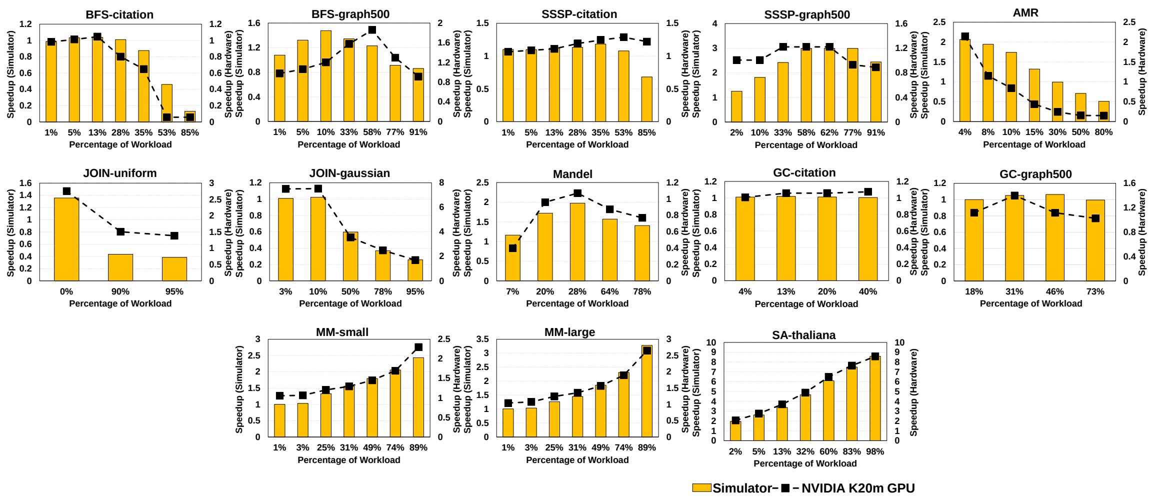


Figure 2.5: Effect of parent-child workload distribution on overall performance. We calculate the speedup in simulator (bars) and hardware (dashed curve) separately, by normalizing performance to the performance of running application's flat (non-DP) implementation on simulator and hardware, respectively. The x-axis shows the percentage of workload offloaded by launching child kernels.

We use 8 applications and generate 13 benchmarks (each benchmark is an <application, input> pair) by varying input sets of a few applications. The applications along with the benchmarks are listed in Table 2.1. `MM` and `SA` are two applications written by our group. In `MM`, each parent thread multiplies one row (or couples of rows) of the multiplicand matrix with an entire multiplier matrix. In the DP version, a parent thread launches a child kernel and each thread of that child kernel picks up one column from the multiplier matrix to perform multiplication. In `SA`, all the reads [4] are divided into sections. Each parent thread handles one section of reads. For each read, there are several candidate locations in the reference index

[4]A read is a substring of genome.

to match. The number of candidate locations varies among reads. In the DP version of this application, a thread launches a child kernel for a read if it has too many candidate locations. All the applications have a *flat* variant that does not use dynamic parallelism.

Table 2.1: List of benchmarks.

Applications	**Input Sets**	**Benchmarks**
Adaptive Mesh Refinement [28]	Combustion Simulation [38]	AMR
Breadth-First Search [21,28]	Citation Network [39]	BFS-citation
	Graph 500 [39]	BFS-graph500
Single Source Shortest Path [10,28]	Citation Network [39]	SSSP-citation
	Graph 500 [39]	SSSP-graph500
Relational Join [28,40]	Uniform Data	JOIN-uniform
	Gaussian Data	JOIN-gaussian
Graph Coloring [41]	Citation Network [39]	GC-citation
	Graph 500 [39]	GC-graph500
Mandelbrot Set	N/A	Mandel
Matrix Multiplication	Small sparse matrix	MM-small
	Large sparse matrix	MM-large
Sequence Alignment [42]	Arabidopsis Thaliana [43]	SA-thaliana

We measure performance using **speedup**, which is the ratio of the execution time of the flat (non-DP) implementation to the execution time of the DP implementation. We use *geometric mean* to represent the average speedup across all benchmarks. We also define **resource utilization** as the *maximum* of the register file utilization, shared memory utilization, and GPU compute unit (SMXs) utilization.

For our 13 benchmarks (Table 2.1), we study the performance impact of varying the workload distribution ratio between the parent and child kernels. Each plot in Figure 2.5 represents one benchmark and the percentage numbers on x-axis represent the amount of workload offloaded to child kernels. Note that this analysis is *static (off-line)*, performed by changing *THRESHOLD* in the application code. It is important to emphasize that offloading 100 percent of a workload to child kernels is also possible. However, this would lead to intra-warp inefficiency because a very small workload might not use all the threads in a warp.

We show the results obtained from both the simulator and a real hardware in Figure 2.5. The yellow bars represent the performance results obtained using a modified version of GPGPU-Sim [1,44], and the dashed lines represent the performance results obtained using NVIDIA Tesla K20m GPU. We use NVIDIA CUDA profiler [45] to profile the performance on hardware. The performance trends observed when using the simulator and the real hardware are similar. All the other observations and results provided in the rest of this chapter are based on simulation results. From this analysis, one can make four major observations:

Observation 1: The preferred workload distribution ratio for each benchmark is different. Further, a given application (e.g., `BFS`) can have different preferred workload distribution ratios for *different inputs*.

Observation 2: Two of the benchmarks (`Join-uniform` and `AMR`) prefer processing the majority of work within the parent threads instead of launching child kernels. `Join-uniform`'s input is regular, and the workload is balanced across all parent threads, leading to its preference of performing the workload within parent threads without launching child kernels. On the other hand, `AMR` launches nested child kernels and it is bottlenecked with the concurrent CTA limitation, and thus it also prefers to perform computations within the parent threads.

Observation 3: Three of the benchmarks `MM-small`, `MM-large`, and `SA-thaliana` prefer offloading a significant amount of workload to child kernels. In `MM`, both inputs are sparse matrices, resulting in severe workload imbalance among threads. Similarly, the number of candidate positions in `SA` varies among different reads, leading to workload imbalance among threads. Additionally, both `MM` and `SA` launch a small number of heavyweight child kernels, which means that the launch overheads have already been effectively hidden by the interleaved execution.

Observation 4: All the other benchmarks gain significant (8.6$\times$ in `SA-thaliana`) to modest (4% in `Join-Gaussian`) performance improvements by offloading parts of their computational workloads to child kernels, except `GC-citation`. In `GC-citation`, the number of child kernels is few (<2300 child kernels), and the amount of work in a parent is still significant to hide the launch overheads, leading to little variance between processing in the parent kernel and offloading to the child kernels.

To understand how a workload distribution impacts the GPU core utilization, consider Figure 2.6 which shows an execution snippet of `BFS-graph500`. The figure plots the number of concurrently-executing CTAs along with the resource utilization

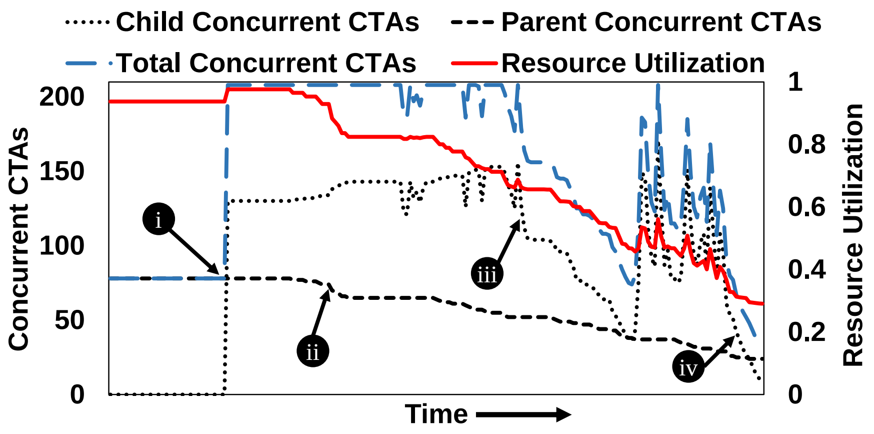


Figure 2.6: CTA concurrency and resource utilization over the course of execution of `BFS-graph500` Baseline-DP. The maximum number of concurrently-running CTAs across all SMXs is 208. The total number of concurrently-running CTAs is the sum of the number of concurrent-executing child and parent CTAs.

(as defined in Section 2.3.1). Initially, until cycle ❶, only the parent CTAs are executing. The child CTAs start their executions beyond that point, increasing resource utilization until the maximum concurrent CTAs is reached (between ❶ and ❸). Due to this hardware-imposed limit, even with enough available hardware resources, the GPU cannot run more CTAs. Starting from time ❷, the parent CTAs start to finish and relinquish resources, allowing more child CTAs to be scheduled. The resource utilization keeps decreasing because the child CTAs usually tend to be lightweight, not requiring as much hardware resources as the parent CTAs [28]. The number of concurrent child CTAs fluctuates between ❸ and ❹ because of two reasons. First, apart from the concurrent CTA limitation, there is a concurrent kernel limitation due to the limited number of HWQs. As a result, a large number of child kernels with a few CTAs per kernel will hit the concurrent kernel limit instead of the concurrent CTA limit, leading to a few concurrent child CTAs. Second, the trailing child kernels have long latencies before they can start executing, resulting in system idleness due to launch overheads. We show in Section 2.4 how an intelligent workload balance can allow a better GPU core occupancy, thereby improving the overall GPU utilization.

(*c_ grid*, *c_ cta*): Figure 2.7 shows the performance variation with varying child CTA dimensions. The speedup is *normalized* to the CTA dimension with 32 threads. We observe from this plot that only certain applications such as `AMR` and `SSSP-graph500` are sensitive to the CTA dimensions. `AMR` is bottlenecked by the hardware CTA concurrency limit under small CTA dimensions. Larger CTA dimensions prevent `AMR` from reaching this CTA concurrency limit. `SSSP-graph500` prefers smaller child CTA dimensions, because the resource requirement for each of the child CTAs is high due to the unavailability of hardware threads. As a result, in `SSSP-graph500`, having smaller CTAs helps the CTA scheduler allocate more CTAs on SMXs, as the resource requirement is low compared to a larger-sized CTA.

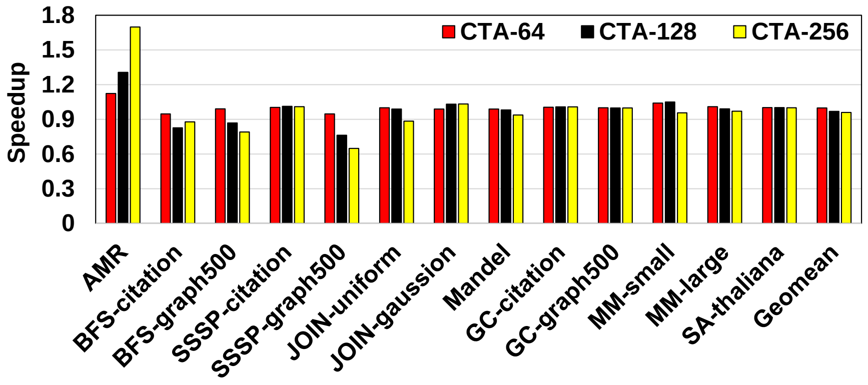


Figure 2.7: Performance sensitivity to different CTA sizes (64, 128, and 256 threads/CTA).

***c_ stream*:** We also studied the impact of the number of SWQs on performance. As discussed in Section 2.2.2, child kernels can be assigned with 1) a unique SWQ id for each child kernel, or 2) the same SWQ id for all child kernels being generated by a given parent CTA. The former enables more kernels to run concurrently, whereas the latter has fewer SWQs to manage. We compare these two mechanisms in Figure 2.8, and observe that assigning each child kernel a unique SWQ id always performs better. This is mainly because, in the second mechanism, a sequential execution of kernels limits concurrency. Therefore, we choose to assign each child kernel a unique SWQ id in all of the experiments presented in the rest of this chapter.

In conclusion, our characterization shows that varying the workload distribution ratio (*THRESHOLD*) results in significant performance impact for our applications, while the other parameters do not affect most of the applications.

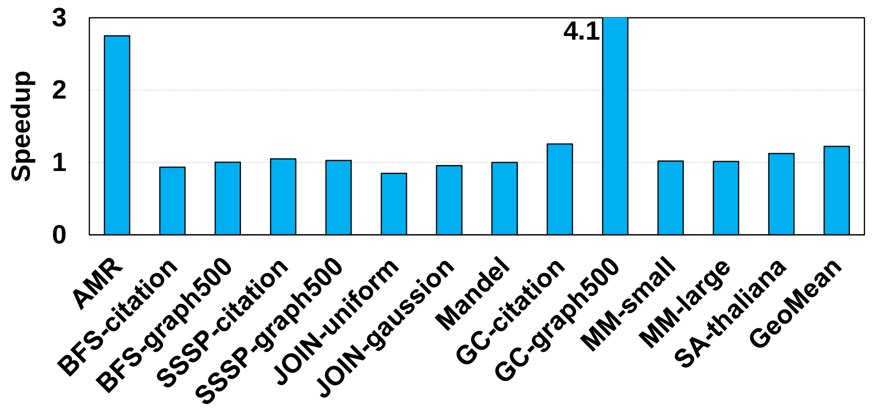


Figure 2.8: Performance of one SWQ per child kernel, normalized to performance of one SWQ per parent CTA.

2.3.2 Potential Benefits of Parent-Child Workload Distribution

We now show the potential benefits of different workload distributions (partitioning) between the parent and child kernels with the help of an example. For the convenience of explanation, we assume there are 3 HWQs. In Figure 2.9, Ⓘ shows the execution time-line of the baseline DP scenario. At the very beginning, the parent kernel starts its execution, and there are multiple parent CTAs that are being executed concurrently. At some point during the execution, the local workload of some threads is found to be greater than *THRESHOLD*. These threads launch child kernels while the other threads proceed normally. As discussed in Section 2.2.3, these child kernels need to wait for a period of time before they can start executing due to the *launch overhead* (🅐). We further assume that each child kernel is associated with one unique SWQ id. However, since the number of HWQs is 3, there can be only 2 child kernels running concurrently along with the parent kernel. The remaining kernels have to wait and this results in increased queuing latencies. In Ⓘ, most of the parent threads launch child kernels, and consequently, the amount of computation performed by the parent kernel is less. As a result, most parent threads finish their executions faster and the GPU is under-utilized as child kernels are not able to start executing right away. There are two major shortcomings in this baseline DP execution. First, it *cannot* hide all the launch overheads. Second, due to the large number of child kernels in the queue and limited concurrency (number of HWQs) of the GPU hardware, the queuing latency of the child kernels can be quite high, leading to performance degradation.

Figure 2.9 II shows a possible solution to mitigate these performance penalties. By limiting the workload offloaded to child kernels, first, the overall number of child kernels is reduced. This results in few and sparse child launching API calls and consequently reduces the launch overhead. In addition, more computation is performed within the parent threads. As a result, the parent thread execution is extended and can hide the launch overhead and queuing latency more effectively. A better workload balance, although not optimal, is achieved in II. It saves us B execution cycles.

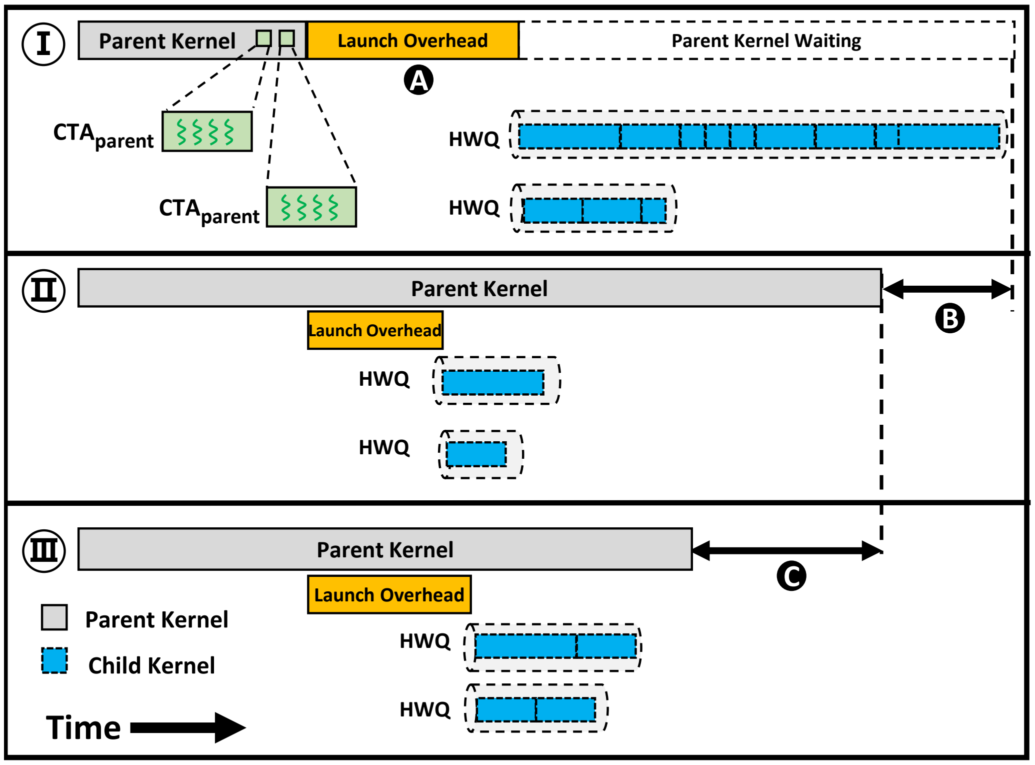


Figure 2.9: Time-line graph showing the benefits of balanced workload distribution between the parent and child kernels.

Obviously, in the best case scenario, the launch overhead is completely hidden while all necessary child kernels are launched to improve parallelism. Queuing latency also reduces since there are fewer pending kernels. III depicts such a case. Further execution time savings can be achieved by balancing the workload between the parent and child kernels if more concurrency is available. Such an approach takes full advantage of the available parallelism in a workload-balanced fashion, resulting in additional savings of Ⓒ cycles.

In summary, the workload distribution (partitioning) between the parent and child kernels is the most important parameter, and has a significant performance impact on DP applications. Since the preferred ratio varies among different applications (even with different inputs for the same application), setting a proper ratio is non-trivial and requires the knowledge of GPU runtime state. This, in turn, motivates the need for a *dynamic mechanism* that can control the workload distribution ratio between the parent and child kernels *on the fly* . To this end, we propose our runtime framework SPAWN.

2.4 SPAWN: Dynamic Launch Control of Child Kernels

In this section, we describe our proposed approach to determine a balanced workload distribution between the parent and child kernels.

Overview: To achieve a balanced workload distribution between the parent and child kernels, we propose a runtime framework called SPAWN, oriented towards improving the GPU performance. The goal of SPAWN is to 1) improve GPU occupancy, 2) prevent the application from reaching the hardware-limits, and 3) dynamically control the performance trade-offs between increasing parallelism (launching child kernels) and incurring overheads.

Challenges: In order to effectively achieve a balanced workload distribution between the parent and child kernels, we should be able to estimate how beneficial it will be to launch a new child kernel, as opposed to performing the specified computation within the parent thread. To better explain this, let us consider the example depicted in Figure 2.10, which shows the child kernel launches from three different parent threads (PT_i).

At time t_1, two parent threads PT_1 and PT_2 launch their respective child kernels (C_1 and C_2), and these child kernels start their executions at time t_3. PT_3 makes

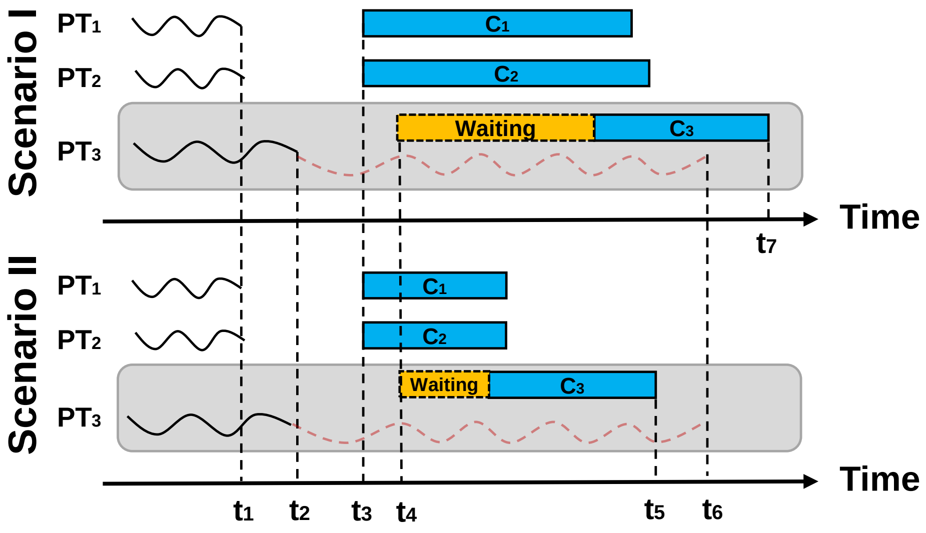


Figure 2.10: Illustrating the advantages and importance of knowing the runtime status while a parent thread is launching a child kernel.

a decision whether to launch C_3 or not at t_2. if C_3 is launched, it cannot start its execution immediately due to the launch overhead. Let us assume that C_3 is launched and can start its execution at time t_4. Based on the hardware requirements of C_1 and C_2 at time t_4, one can have two different scenarios. In **Scenario I**, C_1 and C_2 occupy most of the GPU resources for a long duration. In such a case, child kernel C_3 needs to wait for a long time for GPU resources to be freed up so that it can start its execution. Finally, C_3 finishes its execution at t_7. However, if PT_3 performs the computations itself without launching C_3 at time t_2, it finishes its execution at t_6, resulting in shorter execution time than the case where PT_3 launches C_3. On the other hand, as illustrated in **Scenario II**, C_1 and C_2 could be short running kernels and occupy resources for a short period of time. This would cause C_3 to start its execution earlier and thus, finish faster at t_5, where $t_5 < t_6$. Therefore, in this second scenario, launching a child kernel for PT_3 would be beneficial for improving performance.

2.4.1 The SPAWN Model

There are two major components of our SPAWN framework: Child CTA Queuing System (CCQS) and SPAWN Controller. As shown in Figure 2.11, CCQS monitors the launched child kernels and provides feedback information to the SPAWN controller enabling the latter to make a decision about child kernel launchings.

Child CTA Queuing System (CCQS): CCQS models the Grid Management Unit (GMU) as a "queue" and the SMXs as a server. The launch of child kernels generates CTAs, which act as "jobs" for CCQS[5]. As shown in Figure 2.11, the arrival rate of the jobs is denoted by λ. It conveys the spawning rate of CTAs from the new child kernels into the system. The throughput of CCQS is denoted by T. It conveys the rate of processing the child kernel CTAs on the GPU. Let n be the number of total jobs in CCQS, including both the running and pending child CTAs. Since CCQS works in a FCFS fashion, newly-launched child CTAs need to wait for the previous CTAs to be drained from CCQS and relinquish the occupied resources. Note that, if the child CTA arrival rate (λ) is greater than the throughput (T), CCQS accumulates more child CTAs, leading to long queuing latencies for newly-launched kernels.

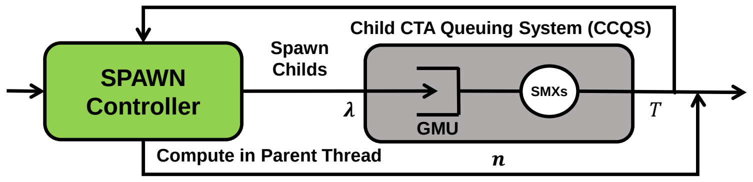


Figure 2.11: High-level view of SPAWN.

The SPAWN Controller: At each kernel launch call, SPAWN controller is invoked, and it is responsible for estimating the benefit of launching that child kernel, and making a decision on launching or not. For each child kernel, there are three time components involved: 1) launch overhead, 2) queuing latency, and 3) execution time. In our SPAWN framework, the launch overhead is modeled as the time to push child CTAs from SPAWN controller to CCQS. Note that we separate the launch overhead from CCQS, as CCQS tracks the child kernel CTAs only *after* they are

[5]We use CTA granularity for our model because of two reasons: 1) each CTA execution is independent, and 2) CTAs cannot be preempted, or migrated to another core [37].

pushed into GMU. Queuing latency is modeled in CCQS as queuing time, and is calculated by examining throughput (T) and the number of jobs (n) residing in CCQS. Execution time on cores is modeled as the service time in CCQS, and is calculated using throughput (T) and the number of CTAs (x) that the new kernel has. Therefore, we can approximate the time it takes for a new child kernel to finish its assigned workload using Equation 2.1,

$$t_{child} \approx Launch\ overhead\ + \frac{n}{T}\ + \frac{x}{T} \tag{2.1}$$

where:

$T\ = \frac{\text{Average Number of Concurrent CTAs}}{\text{Average Child CTA Execution Time}}$ and

x is the number of CTAs in the new kernel.

Similarly, Equation 2.2 estimates the time needed by the parent thread to perform the computations within itself rather than performing them in a child kernel. Generally, the parent thread will perform the computation in an iterative fashion. Each iteration time is approximately similar to the counterpart's child warp execution time.

$$t_{parent} \approx Workload \times t_{warp} \tag{2.2}$$

where:

t_{warp} is Average Child Warp Execution Time

By comparing the results of these two equations, our SPAWN controller chooses the option with the lower estimated execution time. Algorithm 1 gives the working of SPAWN in detail. Initially, it decides to launch child kernels because there is no CTAs in CCQS (line 2 to 3). Line 5 and line 6 represent Equations 2.1 and 2.2, respectively. Note that, there is a maximum queue size in CCQS, which we set to 65,536 in our implementation, based on the Kepler architecture [30].
Accuracy: SPAWN controller uses the historical average child CTA execution time to estimate the execution time of newly-launched child CTAs. In other words, SPAWN might make wrong decisions and lose opportunities if the execution time

Algorithm 1 SPAWN Controller

```
INPUT:
    $n$ :             Total child CTAs in CCQS.
    $x$ :             number of CTAs in new child kernel.
    $workload$ :      Workload hold by parent thread.
    $t_{overhead}$ :  Child launch overhead.
    $t_{cta}$ :       Average child CTA execution time.
    $t_{warp}$ :      Average child warp execution time.
    $n_{con}$ :       Average number of concurrent CTAs.
    $t_{child}$ :     Estimated child kernel execution time.
    $t_{parent}$ :    Estimated parent thread execution time.
 1: Initialization
 2: if $t_{cta} = 0$ then
 3:     Spawn child kernel
 4: end if
 5: $t_{child} \leftarrow t_{overhead} + (x + n) \times t_{cta}/n_{con}$
 6: $t_{parent} \leftarrow workload \times t_{warp}$
 7: if $t_{child} \leqslant t_{parent}$ and $n + x \leqslant max\_queue\_size$ then
 8:     $n \leftarrow n + x$
 9:     Spawning Child Kernel
10: else
11:     Process computation in parent thread
12: end if
```

has a big variance among most child CTAs. However, this does not happen in most DP applications because: 1) all child CTAs share the same instructions and thus require similar hardware resources, and 2) child kernels are essentially lightweight and contain lightweight CTAs. Therefore, it is unlikely that the child CTA execution times significantly vary. In Figure 2.12, we show the PDF of child CTA execution time from four of our benchmarks. As one can see, 95% of the child CTAs (80% in `SSSP-graph500`) have their execution time within 10% of the average child CTA execution time. Because of this characteristic, even though SPAWN needs time to get t_{cta} converge to the average at the beginning of execution (within 5% of total execution), it can accurately estimate most child kernel execution times and make proper decisions for the remaining execution of the program.

2.4.2 Implementation Details

Figure 2.13 shows the high-level implementation of our SPAWN runtime framework. This implementation has two parts: 1) a source-to-source translator, and 2) an extension to the CUDA runtime that acts as a wrapper for the SPAWN controller

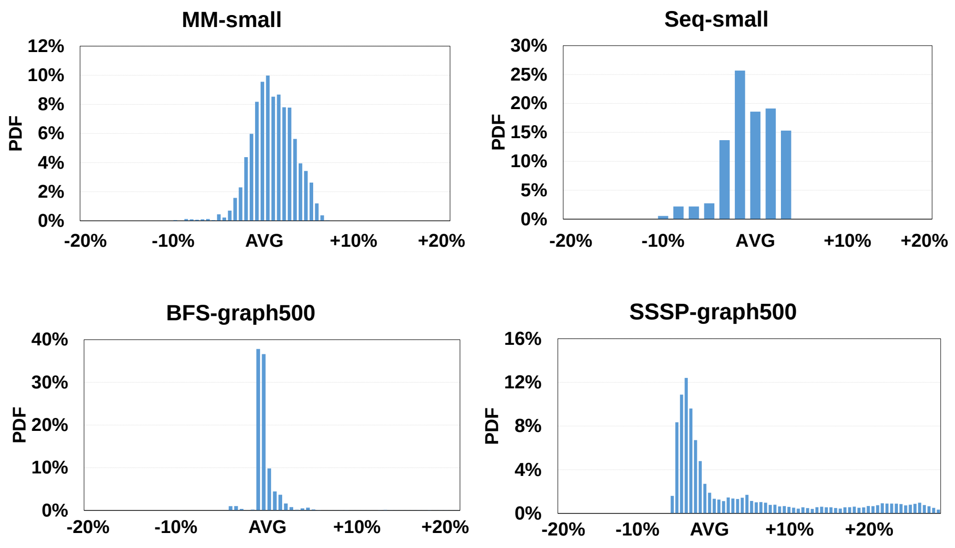


Figure 2.12: PDF of Child kernel CTA execution time.

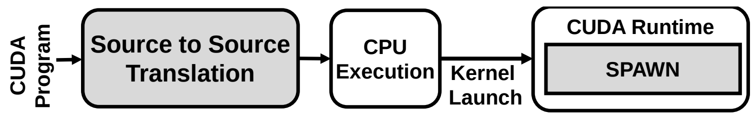


Figure 2.13: High-level view of SPAWN implementation.

function.

Source-to-Source Translator: Figure 2.14 shows the translated source code. First, the declaration of the kernel environment variables are moved outside the condition block, and the CUDA device launch function is used as the condition clause. The API function call returns with a flag of "success" when the child kernel is launched; otherwise, it returns with "fail" and the workload will be computed by the parent thread. Second, the child kernel launch needs to integrate the *local_workload* parameter into the CUDA runtime call to facilitate the estimation of the execution times in the SPAWN controller. This relieves the programmer from specifying any value of *THRESHOLD*.

CUDA Runtime Extension: We extend the CUDA Runtime, specifically the device kernel launch API call to integrate the SPAWN controller. At runtime, when the child kernel launch API is executed, SPAWN makes the decision regarding the launch of a child kernel by examining CCQS.

```
__global__ void parent (type *workload){
    type *local_workload = workload[pid]; //each parent threads pick up its workload
    dim3  c_grid; dim3 c_cta;
    cudaStream_t c_stream;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    if (child<<< c_grid, c_cta, shmem, c_stream, local_workload>>>(type
*local_workload)){ … }
    else
        while(local_workload){…}
     …
      cudaDeviceSynchronize(); //waiting all children finishing.
}
```

Figure 2.14: Translated version of the source given in Figure 2.3b.

Monitored Metrics: We monitor the following metrics: 1) n, 2) t_{cta}, 3) n_{con} and 4) t_{warp}. As mentioned in Section 2.4.1, we need n and T to calculate the child kernel execution time. In order to compute T, we use two proxy metrics: i) t_{cta}, average child CTA execution time and ii) n_{con}, average number of concurrent child CTAs. Similarly, we monitor t_{warp}, average child warp execution time, to estimate the parent thread execution time. At the start of an application execution, all the metrics are initialized to 0. n is incremented/decremented in the SPAWN controller whenever a child CTA either enters or leaves CCQS. t_{cta} is updated only when a CTA finishes its execution and leaves CCQS. We compute n_{con} over a window of 1024 cycles. At every cycle, we add the number of concurrently executing child CTAs to n_{con} and, at the end of the window, we bit-shift n_{con} by 10 bits to the right to obtain the average number of the concurrently-running child CTAs in the window. This average number is then used over the next window until a new value of n_{con} is calculated. Similarly, t_{warp} is also calculated in a windowed fashion.

Hardware Overheads: The main hardware overheads involve storing and updating the monitored metrics and computing the execution time. As shown in Figure 2.4, GMU is extended with the SPAWN logic (❽). It requires a 416 bytes table to keep track of each running child CTA's execution time[6]. It also requires one 16-bit register to hold n, two 16-bit adders and one shift register to calculate the estimated execution time. When a child CTA finishes its execution, it updates the related metrics located in GMU. Since the cores and GMU already communicate every cycle, this does not cause any extra communication overhead. The child kernel launch API communicates

[6]The table includes 208 entries, and each entry is a 16-bit cycle counter.

with SPAWN (❼) and returns the decision immediately, as the kernel launch API call is asynchronous.

2.5 Experimental Evaluation

2.5.1 Simulated System

We use a modified version of the cycle-accurate GPGPU-Sim v3.2.2 [44] that is able to simulate concurrent kernel execution and support dynamic parallelism. Table 2.2 provides the configuration details of the simulated system. The simulated system is modeled with 32 Hardware Work Queues (HWQs), therefore, limiting the maximum number of concurrently executing kernels to 32. In our simulation framework, we modify the GPU runtime to support SPAWN as described in Section 2.4.2.

Table 2.2: GPU configuration parameters.

SMX	13 SMXs, 1400MHz, 5-Stage Pipeline
Resources per SMX	48KB Shared Memory, 64KB Register File, Max.2048 threads (64 warps, 32 threads/warp)
cache per SMX	16KB 4-way L1 D-cache, 12KB 24-way Texture cache, 8KB 2-way Constant cache, 2KB 4-way L1 I-cache, 128B cacheline .
L2 Unified cache	128KB/Memory Partition, 1536KB Total Size, 128B cacheline, 8-way associativity
Scheduler	Greedy-Then-Oldest (GTO) [46] dual warp scheduler, Round-Robin (RR) CTA scheduler
Concurrency	16 CTAs/SMX, 32 HWQs across all SMXs
Interconnect	1 crossbar/direction (13 SMXs, 6 MCs) 1.4GHz, islip VC & Switch Allocators
DRAM Model	2 Memory Partition/MC, 6 MCs, FR-FCFS (128 Request Queue Size/MC)
Child Kernel Launch Overhead	$Latency = Ax + b$ where A is 1721 cycles, b is 20210 cycles, x is number of child kernels launched per warp [1]

2.5.2 Experimental Results

We study the effects of utilizing our SPAWN mechanism across 13 benchmarks (Table 2.1). All the speedup results have been *normalized* to the execution of a flat (non-DP) variant of each benchmark. For each benchmark, we analyze the results for three different schemes: 1) the baseline dynamic parallelism execution (Baseline-DP), 2) the best workload distribution ratio[7] (Offline-Search), and 3) SPAWN. Figure 2.15 shows the speedups obtained when using three different schemes. Across the 13 benchmarks evaluated, we observe an average speedup of 69% and 57% compared to the flat variant and Baseline-DP execution, respectively. That is, although Baseline-DP performs better than flat version, our SPAWN significantly outperforms both flat and Baseline-DP. For the Offline-Search execution with the best workload distribution ratio, we obtain performance improvement of 61% over Baseline-DP execution.

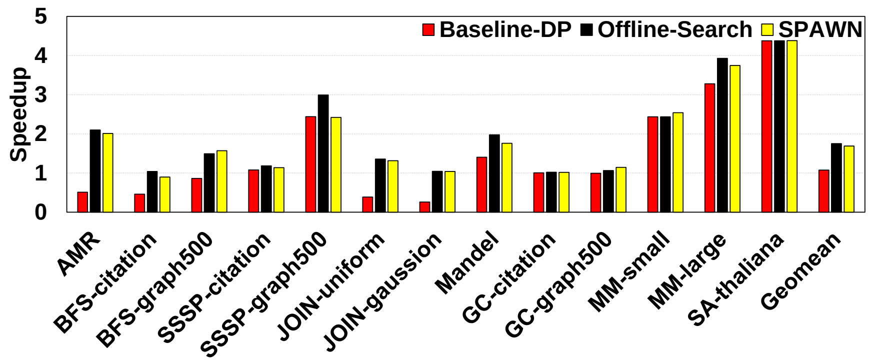


Figure 2.15: Speedup over the flat (non-DP) implementation.

We make three important observations based on these results. First, SPAWN is able to match the speedup obtained by Offline-Search, irrespective of whether the benchmark prefers launching child kernels or performing the computations within the parent thread. For example, `SA-large` has high performance when offloading a significant amount of work to the child kernels, whereas `AMR` prefers processing within the parent threads. SPAWN successfully captures the characteristics of these two dissimilar benchmarks. Note that, SPAWN is able to achieve within 4% of the Offline-Search's performance. Second, for three benchmarks, `BFS-graph500`, `GC-graph500`, `MM-small`, SPAWN performs better than Offline-Search. The slight performance improvement in SPAWN is due to Offline-Search being agnostic to the GPU hardware state. SPAWN is able to dynamically tune the workload distribution over the course

[7]We pick the best workload distribution ratio by performing an exhaustive sweep of the THRESHOLD metric, as mentioned in Section 2.3.

of execution, taking into consideration the current state of the GPU, and also, it is able to control workload distribution decisions on a per kernel basis rather than using a statically fixed THRESHOLD value. Third, SPAWN under-performs for `SSSP-graph500` compared to Offline-Search and is similar to performance of Baseline-DP. This is because, for the monitored metrics to be useful to SPAWN, some child CTAs need to finish for the metrics to be updated to an accurate value. However, in `SSSP-graph500`, by the time the first child kernel finishes and updates the metrics, SPAWN had already made incorrect decisions and launched all the child kernels at this phase of execution.

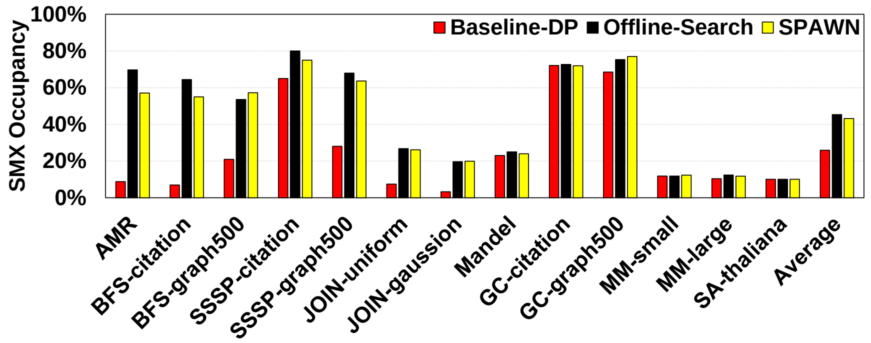


Figure 2.16: SMX occupancy.

Figure 2.16 shows the achieved occupancy across all SMXs. SMX occupancy is defined as the ratio of the average active warps per active cycle to the maximum number of warps supported on all SMXs. A higher SMX occupancy could potentially improve the GPU performance and provide more latency tolerance towards child kernel launch overheads and queuing latency as seen by correlating Figure 2.15 and Figure 2.16. SPAWN achieves, on average, 1.96× higher SMX occupancy over Baseline-DP, and is within 4% of the SMX occupancy achieved by Offline-Search.

We next evaluate the impact of SPAWN on cache performance. Figure 2.17 shows the L2 cache hit rate for the three evaluated schemes. Although SPAWN does not take data reuse and data access pattern into account, the L2 hit rate increases by around 10% compared to the Baseline-DP execution. This is mainly due to two reasons: 1) L2 cache contention is significant in Baseline-DP due to the high number of concurrently-executing child kernels, and 2) child kernels cannot execute *immediately* because of the launch overheads and queuing latency. This delay in execution of child

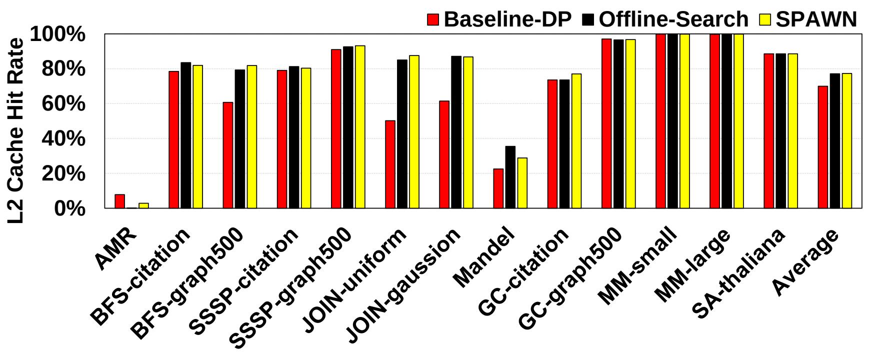


Figure 2.17: L2 cache hit rate.

kernels causes the loss in both temporal and spatial locality between the parent and child kernels [47]. SPAWN is able to increase locality by providing more computations to parent (improving spatial locality) and allowing the parent execution to last longer and overlap with the launched child kernels (improving temporal locality).

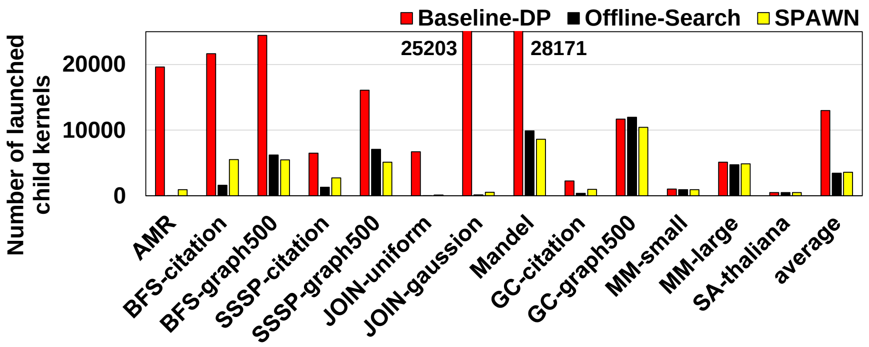
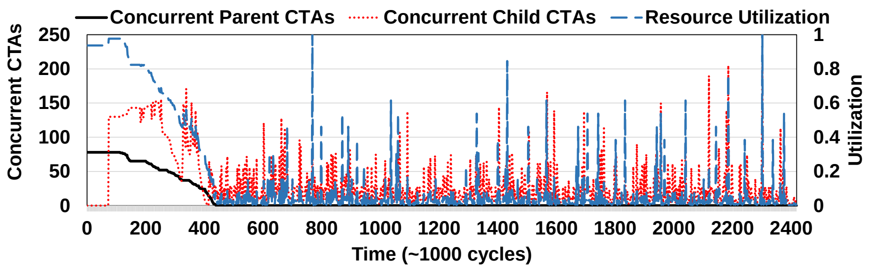
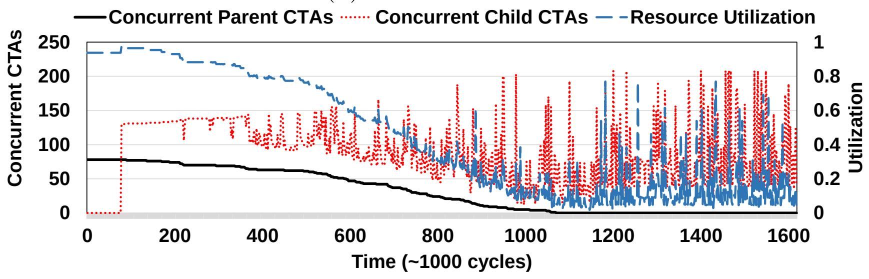


Figure 2.18: Number of child kernels launched.

Figure 2.18 shows the number child kernels that are launched during the benchmark's execution for the three different schemes. Note that the trend in the number of child kernels launched in Offline-Search execution and SPAWN are similar to each other. With SPAWN, the number of child kernels launched significantly reduces (by 73% on average). This reduction in the child kernel count helps in reducing the launch overhead and queuing latency. In the following subsection, we discuss the working of our SPAWN mechanism in detail.



(a) Baseline-DP.



(b) SPAWN.

Figure 2.19: Concurrent CTAs of BFS-graph500 over time.

2.5.3 Dynamic Workload Distribution

To better understand the working of our SPAWN mechanism, we describe the child kernel launch patterns for the Baseline-DP execution and our SPAWN mechanism. Figure 2.19a and Figure 2.19b show the number of concurrent CTAs in `BFS-graph500` scheduled on the SMXs at any given time during the course of execution for the Baseline-DP and SPAWN, respectively. Initially, only parent CTAs execute, following which the child CTAs start execution at 75k cycles. Since Baseline-DP of `BFS-graph500` gives significant work to child kernels, parent threads do not have much edges to traverse. As a result, they finish execution at 436k cycles, after which child kernels start dominating the SMXs' resources. However, there are two issues in the Baseline-DP. First, the child kernels cannot start execution immediately due to the launch overhead and queuing latency. Consequently, the concurrency and resource utilization dramatically drop. Second, many child kernels are launched in Baseline-DP, and they cannot execute concurrently because of the limited number of HWQs. Since each child kernel in `BFS` is lightweight (traversing only the neighboring nodes), the resource utilization is low during the phase when only child kernels execute (from cycle 436k to cycle 2,400k).

In SPAWN (Figure 2.19b), since more parent threads traverse the edges in a loop, the parent CTAs execute for longer duration and fewer child kernels are launched. As a result, the parent CTA execution is now able to hide the child kernels' launch overhead efficiently. In addition, as fewer child kernels are launched, it results in lower launch overhead and reduced queuing latency. Therefore, it leads to higher resource utilization, and allows the application execution finish at 1600k cycles, unlike the Baseline-DP execution which takes 2400k cycles.

Figure 2.20 depicts the cumulative child kernel launch decisions that are taken over the entire execution for `BFS-graph500`. We see that SPAWN is dynamically able to make kernel launch decisions which are similar to the decisions taken by Offline-Search, and achieve similar workload distributions. From the figure, we see that Baseline-DP has considerable high child kernel launch rate compared to SPAWN. Since launch overhead and queuing latency dramatically increase when large number of child kernels are intensively launched[8], a reduction in child kernel launch rate effectively reduces the overheads and improves performance.

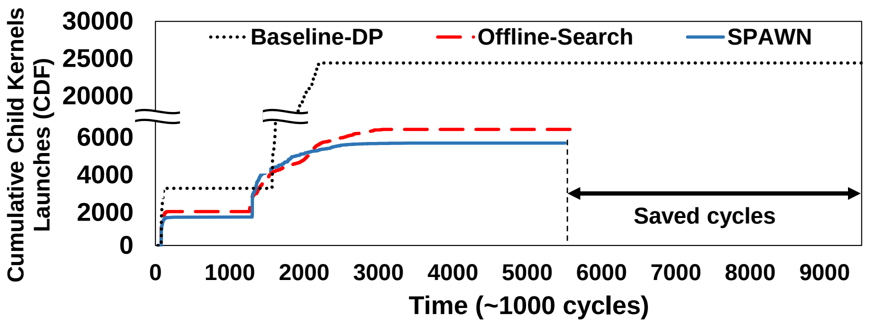


Figure 2.20: CDF of child kernels launched over time

2.5.4 Comparison with an Alternate Strategy

We are not aware of any runtime scheme that tunes the workload distribution (partitioning) between the parent and child kernels in DP applications. Wang *et al.* [1] proposed a mechanism called Dynamic Thread Block Launch (DTBL). Instead

[8]With an intensive child kernel launch rate, the launch overhead and queuing latency gets exposed when there is lack of work in the GPU to hide this increased latency.

of launching child kernels, they propose to launch child CTAs and coalesce them with an existing kernel, thereby removing the launch overheads associated with launching kernels. However, this coalescing of CTA to an existing kernel can happen only when the CTA is equal in dimensions to the CTAs in the existing kernel and have the same instruction sequence for execution. This reduces the applicability of the scheme to a limited set of programs. Also, the number of CTAs launched remains the same in DTBL, which still incur significant queuing latency if the concurrent CTA limitation is reached. We show results from three representative applications in Figure 2.21. `SA` is bottlenecked due to concurrent CTA limitation and SPAWN outperforms DTBL by 1.8× and 1.4× in thaliana and elegans [43], respectively. `MM` launches a lot of large child kernels and suffers from both launch overhead and queuing latency. SPAWN and DTBL perform similarly in this scenario. SPAWN is able to reduce both the launch overheads and queuing latency while DTBL largely eliminates only the launch overhead. DTBL performs better than SPAWN in `SSSP` because `SSSP` launches small child kernels and the execution is bottlenecked by the launch overhead, which DTBL is designed to eliminate.

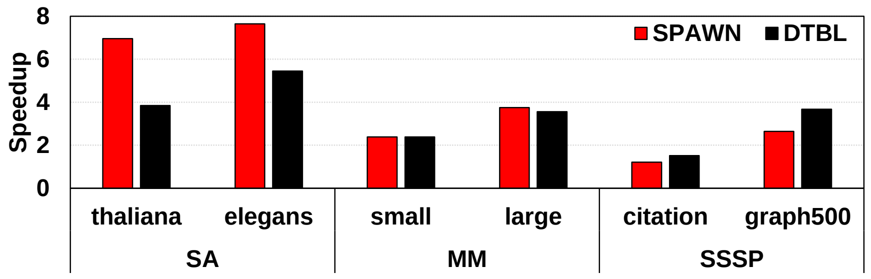


Figure 2.21: Comparison with DTBL [1]. Normalized performance to flat (non-DP) implementation.

2.6 Conclusion Remarks

Although GPUs can be very effective in executing parallel programs, many irregular applications (e.g. graph algorithms with irregular data inputs) that have been ported to GPUs execute inefficiently due to the workload imbalances across its threads. Dynamic parallelism supported by OpenCL and CUDA help in reducing this imbalance by allowing GPU kernels to launch additional kernels on-demand without involving the CPU. However, this approach entails extra performance overheads for launching child kernels; and a straightforward way of launching kernels can lead to both resource underutilization and uneven work across concurrently-executing kernels. Our proposed hardware-based solution, SPAWN, improves the GPU performance by hiding and reducing the performance overheads of child kernel launches, and improving the load balance across different kernels. Using our approach, programmers can port existing irregular applications to GPUs without having to go through extensive architecture-specific software optimizations that balance the work across different kernels. To the best of our knowledge, this is the first work that dynamically tunes the workload distribution (partitioning) ratio among parent and child kernels, to find the sweet spot to minimize launch overhead and queuing latency while maximizing parallelism.

Chapter 3 | Quantifying Data Locality in Dynamic Parallelism in GPU

GPUs are becoming prevalent in various domains of computing and are widely used for streaming (regular) applications. However, they are highly inefficient when executing irregular applications with unstructured inputs due to load imbalance. Dynamic parallelism (DP) is a new feature of emerging GPUs that allows new kernels to be generated and scheduled from the device-side (GPU) without the host-side (CPU) intervention to increase parallelism. To efficiently support DP, one of the major challenges is to saturate the GPU processing elements and provide them with the required data in a timely fashion. There have been considerable efforts focusing on exploiting data locality in GPUs. However, there is a lack of quantitative analysis of how irregular applications using dynamic parallelism behave in terms of data reuse.

In this chapter, we quantitatively analyze the data reuse of dynamic applications in three different granularities of schedulable units: kernel, work-group, and wavefront. We observe that, for DP applications, data reuse is highly irregular and is heavily dependent on the application and its input. Thus, existing techniques cannot exploit data reuse effectively for DP applications. To this end, we first conduct a limit study on the performance improvements that can be achieved by hardware schedulers that are provided with accurate data reuse information. This limit study shows that, on an average, the performance improves by 19.4% over the baseline scheduler. Based on the key observations from the quantitative analysis of our DP applications, we next propose LASER, a Locality-Aware SchedulER, where the hardware schedulers employ data reuse monitors to help make scheduling decisions to improve data locality at runtime. Our experimental results on 16 benchmarks show that LASER, on an average, can improve performance by 11.3%.

3.1 Introduction

Graphics Processing Units (GPUs) provide massive computational throughput for a wide spectrum of applications from various domains such as computer vision [19], finance [16, 17], machine learning [48, 49], and bioinformatics [50]. As progressively more applications get ported to GPUs for parallelization, the shortcomings of the traditional GPU execution model become evident. Particularly, execution of irregular applications with unstructured inputs on GPUs leads to severe bottlenecks such as imbalanced computational load across the GPU Compute Units (CUs). This inefficiency is widely observed in adaptive meshes and graph applications which are becoming important classes of applications due to their increasing popularity. Therefore, it is becoming more and more difficult to effectively utilize GPUs for such applications [28].

Dynamic Parallelism (DP) is a feature supported by CUDA [37] and OpenCL™ [27]. It allows the generation and scheduling (launching) of kernels dynamically on GPUs without the involvement of a host (CPU). This model of computation is quite useful for irregular applications with unstructured and irregular inputs, as it essentially increases parallelism *on-the-fly*. Specifically, if there are threads which are more compute-intensive than other threads, then these threads (*parent threads*) can parallelize their work by launching more threads (*child kernels*). This would allow for better (and on-demand) load balancing as there are higher number of threads to distribute across the GPU cores. However, by increasing the parallelism and redistributing the threads, the data reuse and access pattern can change dramatically. For example, original intra-thread temporal data reuse (i.e., the data blocks that are reused within a thread) can translate to inter-thread temporal reuse, due to the fact that parent thread offloads computations to its child threads. Moreover, this intra-thread temporal data reuse can even translate to inter-thread spatial locality, as the child kernel has multiple threads, and each child thread can work on a small portion of data.

Prior techniques on GPU cache optimization involved throttling the available parallelism [51, 52], bypassing the cache for certain memory requests to reduce contention [53, 54], and building efficient cache management policies tuned towards GPU applications [55, 56]. These techniques cannot be ported to improve cache performance for DP applications, as they are agnostic to the on-demand kernel launch behavior in DP applications. Furthermore, it is extremely difficult to control the temporal data reuse in applications via cache optimizations as they lack mechanisms to control the scheduling of instructions for execution. To be able to efficiently control the temporal data reuse of applications, we need intelligent scheduling strategies. Prior efforts efficiently scheduled applications based on their reuse behavior using runtime statistics, or via compiler based approaches, *but* they do not consider DP applications [52, 57, 58]. Wang *et al.* [47] proposed a work-group (WG) scheduling mechanism that maps child WGs together with its parent WG to the same compute unit (CU) during execution. This strategy does not differentiate between the children of a parent or take into account the data reuse behavior of the children among themselves. No prior work quantitatively studied the data reuse nor explored the potential performance benefits by *fully* exploiting the data reuse opportunities in DP applications.

Our goal in this chapter is three-fold: (1) to quantitatively analyze the intrinsic data reuse opportunities in DP applications; (2) to reveal the best achievable performance improvements through a limit study; and (3) to propose a practical scheduling mechanism that improves data locality. To this end, we first conduct an in-depth data reuse analysis. We define a new "metric" called *reuse ratio* which captures the "intensity" of data reuse. The reuse ratio is computed for each pair of "schedulable units" where a schedulable unit is either a *kernel*, a *work-group*, or a *wavefront*. We then perform a limit study that exploits the reuse ratios for all possible permutations of the schedulable units at each level of the scheduling hierarchy and optimizes the placement (temporally and spatially) of the schedulable units to maximize data locality. Finally, we modify the hardware schedulers and propose a practical scheduling mechanism that schedules the schedulable units in a locality-aware fashion. this chapter makes the following major contributions:

- It provides an in-depth *data reuse* quantification and analysis of GPU dynamic parallelism applications. The analysis is at multiple granularities (kernel, work-group, and wavefront) with respect to suitability for improving cache locality via scheduling techniques.

- It defines a new metric called *reuse ratio*, which captures the intensity of data reuse among schedulable units. We discuss the merits of this metric and demonstrate how to use it to guide scheduling strategies for DP applications.
- It performs a limit study by proposing an *optimal* scheduling mechanism that optimizes compute placement for cache locality by exploiting reuse ratios at each level of scheduling, viz. kernel, work-group and wavefront. This is achieved by providing accurate "reuse ratio" information to the hardware schedulers. The *optimal* scheduler provides, on an average, 19.4% performance improvement over the baseline scheduler.
- Based on the key observations from the data reuse analysis, it proposes LASER, a Locality-Aware SchedulER, that monitors the reuse ratio metric and makes scheduling decisions based on it. Our experimental results show that, on an average, LASER provides 11.3% performance improvement over the baseline scheduler.

To our knowledge, this is the first work that systematically investigates the data reuse and access patterns of DP applications at various granularities of schedulable units. Our work is most closely related to the work of Wang *et al.* [47]. However, their approach does not differentiate between the children of a given parent kernel, nor does it consider the intra-kernel data reuse. Therefore, we quantify the memory behavior of DP applications in detail and analyze their data reuse patterns to answer the following key questions. **Question 1:** *How prevalent are intra-kernel and inter-kernel data reuses in DP applications?* **Question 2:** *What is the effect of launch overhead on data reuse patterns?* **Question 3:** *Do neighboring work-groups have more of temporal or spatial reuse?* and **Question 4:** *Is it necessary to always prioritize the child work-groups?*

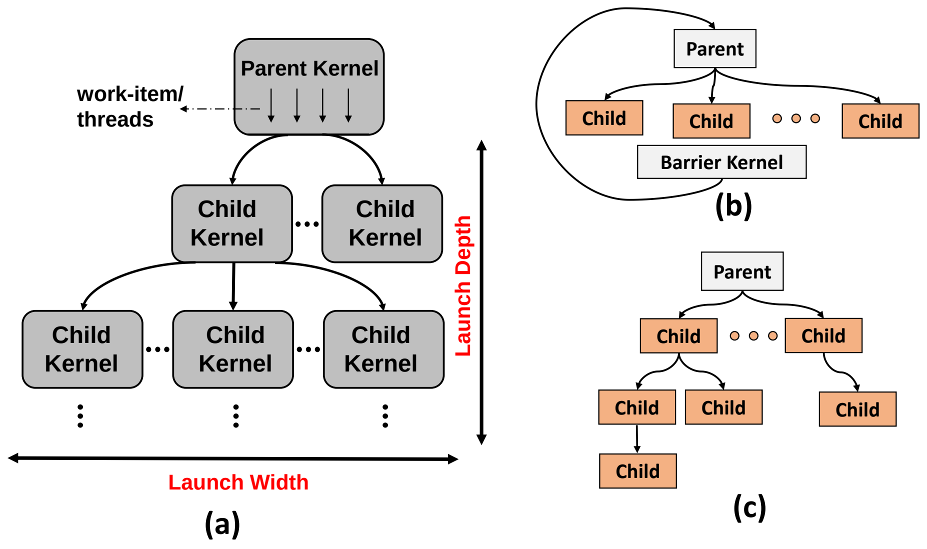


Figure 3.1: (a) DP programming model. (b) Type I DP applications. (c) Type II DP applications.

3.2 Background

3.2.1 Dynamic Parallelism

In conventional GPU programming model, computations in an application are mapped to work-items (threads in NVIDIA terminology), work-groups (thread-blocks in NVIDIA terminology), and kernels. A kernel consists of multiple work-groups (WGs) and a WG consists of multiple work-items. At runtime, work-items are scheduled in groups, called "wavefronts". All threads in a wavefront execute the same instruction in a lock-step fashion. Unlike conventional applications, a DP application can launch nested device kernels (child kernels), as shown in Figure 3.1(a). Each work-item in a kernel has the capability to launch kernels. This feature is particularly useful for irregular applications, as threads with heavy computations can launch a child kernel and *offload* some of its computations to that child kernel. Therefore, one can potentially achieve both parallelism and better resource utilization. Note that, it is the programmer's responsibility to decide whether to launch a child kernel or not from within a parent kernel. Specifically, a *threshold* is set by the programmer and used in DP applications to make kernel launch decisions [26, 59]. In Figure 3.1(a), we call the depth of the nested launched kernels as *Launch Depth*. It represents the number of nested levels (depth) at which child kernels are launched. Note that the GPU hardware needs to reserve memory space for the child kernels [26, 28]. As a result, the maximum launch depth is limited to 24 by the hardware [37]. At each launch depth, there can be multiple kernels being launched. We refer to the number of kernels launched at a given depth as *Launch Width*.

3.2.2 Baseline Architecture

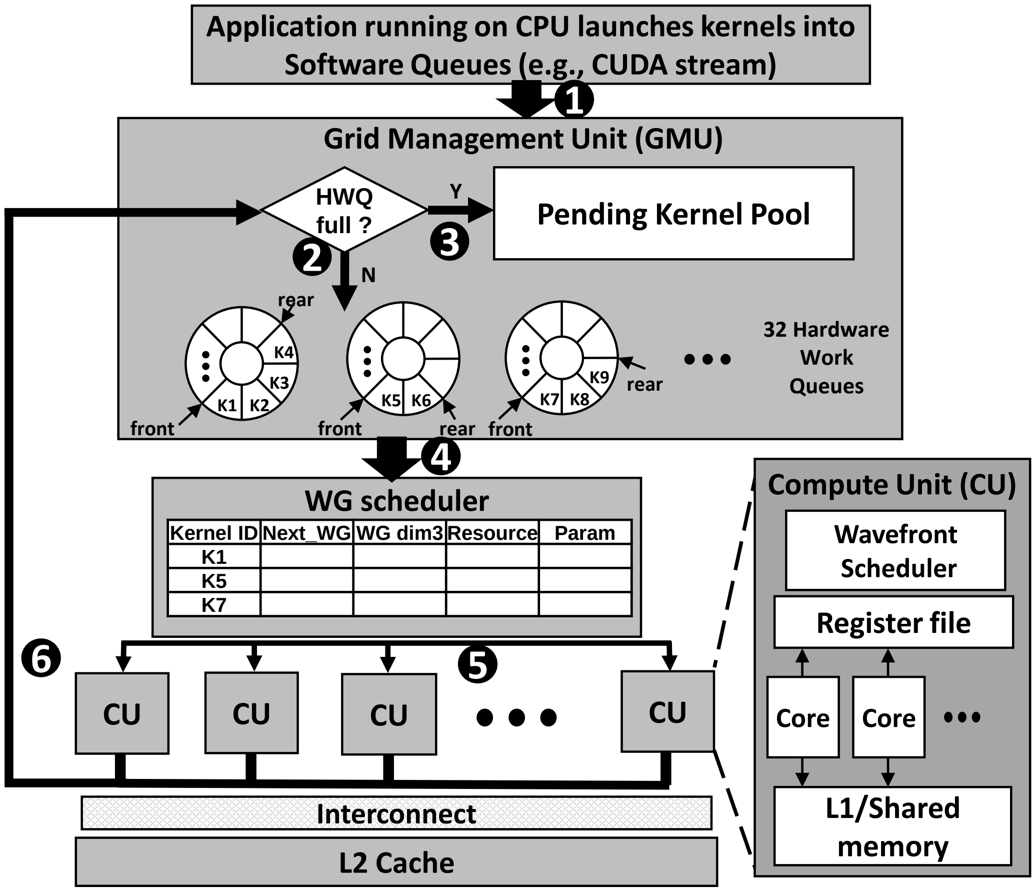


Figure 3.2: Baseline GPU architecture.

Figure 3.2 shows our *baseline* GPU architecture with support for DP. A DP application (similar to a regular GPU application) starts running on a host CPU and the very first kernel (parent) is launched to the GPU from the host (❶). This kernel is also labeled with a software queue ID (e.g., CUDA stream ID in CUDA terminology), which is used to provide execution ordering among kernels[1] There are 32 hardware work queues (HWQs) located in the Grid Management Unit (GMU). Kernels with the same software queue ID are mapped to the same HWQ, and all kernels in the same HWQ are executed sequentially. If there is an available slot in a HWQ, the launched kernel is pushed into that HWQ (❷). Otherwise, the kernel is suspended in the pending kernel pool (❸), waiting for a free slot in HWQ. Kernels are scheduled for execution in a first-come-first-serve (FCFS) order (❹), with respect to kernel dependencies. The kernels at the head of HWQs can potentially execute concurrently. That is, the WG scheduler can schedule any work-groups (WGs) from the "head-of-queue" kernels, and map them onto CUs, if there are enough available resources (❺). In the baseline execution, the WG scheduler picks up WGs from the "head-of-queue" kernel and distributes them across all CUs in a round-robin fashion. On a CU, wavefronts from WGs are executed on cores. At any time during wavefront execution, multiple wavefronts may be standing by and waiting for execution in order to hide long latency operations (e.g., memory accesses, expensive math operations, etc.). Once a wavefront encounters a long latency operation, that wavefront is switched out, and another "ready" wavefront is scheduled to overlap with the long latency operation. The ready wavefront is chosen by the *wavefront scheduler* which uses a greedy-then-oldest policy (GTO) [46] to select candidate wavefronts. Specifically, in GTO, the same wavefront is always issued to execute if it does not encounter any long latency operations. Whenever a long latency operation is observed, the wavefront is replaced with the "oldest" pending wavefront.

With DP, any thread running on a CU can launch device kernels by calling the driver API. The newly-launched kernels are pushed into GMU by device driver (❻). Note that launching child kernels incurs extra latencies, (called *Launch Overhead*) [28]. Due to this overhead, a child kernel cannot start its execution immediately after launching. This overhead is accurately modeled in our simulation framework, and multiple approaches have been proposed in the literature to mitigate it [1, 59–61].

[1]In DP, the parent thread can choose two ways to assign software queues (SQs) to child kernels. First, it can create a new SQ before launching a child kernel. Second, each parent WG has a default SQ. Parent threads in the same WG launch child kernels to the default SQ, if no new SQs are explicitly created for the child kernels.

3.2.3 Evaluation Methodology

Table 3.1: GPU configuration parameters.

CU	13 CUs, 1400MHz, 5-Stage Pipeline
Resources per CU	32KB Shared Memory, 64KB Register File, Max.2048 threads (64 wavefronts, 32 threads/wavefront)
cache per CU	32KB 8-way L1 D-cache, 12KB 24-way Texture Cache, 8KB 2-way Constant cache, 2KB 4-way L1 I-cache, 128B cacheline
L2 Unified cache	128KB/Memory Partition, 1536KB Total Size, 128B cacheline, 8-way associativity
Scheduler	Greedy-Then-Oldest (GTO) [46] dual wavefront scheduler, Round-Robin (RR) WG scheduler
Concurrency	16 WGs/CU, 32 HWQs across all CUs
Interconnect	1 crossbar/direction (13 CUs, 6 MCs), 1.4GHz, islip VC and switch allocators
DRAM Model	2 Memory Partitions/MC, 6 MCs, FR-FCFS (128 Request Queue Size/MC)
Child Kernel Launch Overhead	$Latency = Ax + b$ where A is 1721 cycles, b is 20210 cycles, x is number of child kernels launched per wavefront [1]

Infrastructure: We use a cycle-level simulator, GPGPU-Sim [44], that enables DP [1] as our evaluation framework. Table 3.1 provides the detailed configuration of the baseline GPU architecture. The maximum number of HWQs is 32, and the maximum number of concurrent WGs per CU is 16.

Table 3.2: Benchmark characteristics: Number of kernels, type, depth (refers to the number of stages (parent-child-barrier) for Type-I applications and Launch Depth for Type-II applications), and high-reuse chain length (kernels/WGs).

Application	**Input Sets**	**Benchmark**	**Total # of Kernels**	**Type**	**Depth**	**Chain of Kernels (Max,Avg)**	**Chain of WGs (Max,Avg)**
Adaptive Mesh Refinement [28]	Combustion Simulation [38]	AMR	1261	II	24	(6, 1.5)	(6, 1.5)
BFS [28]	Small Graph	BFS-small	1030	I	1	(1, 1)	(240, 11)
	Citation Network [39]	BFS-citation	126	I	23	(1,1)	(240, 37)
	Graph 500 [39]	BFS-graph500	6899	I	6	(27, 3)	(240, 17)
Graph Coloring [41]	Citation Network [39]	GC-citation	221	I	87	(1, 1)	(443, 2)
	Graph 500 [39]	GC-graph500	924	I	139	(10, 6)	(18, 4)
Relation Join [28]	Gaussian Distribution	JOIN-Gaussian	159	I	1	(21, 4)	(227, 4)
	Uniform Distribution	JOIN-uniform	6725	I	1	(721, 361)	(7, 2)
Mandelbrot Set [59]	N/A	Mandel	1025	II	6	(1, 1)	(1, 1)
Sparse Matrix Multiplication [59]	Small Matrices	SPMM-small	1025	I	1	(1024, 28)	(256, 23)
	Large Matrices	SPMM-large	5121	I	1	(824, 39)	(256, 72)
Quick Sort [34]	N/A	Quicksort	56	II	24	(48, 28)	(48, 28)
Radix Sort	N/A	Radixsort	4765	II	24	(176, 21)	(176, 21)
Sequence [42]	Arabidopsis Thaliana [43]	SA	24	I	1	(22, 19)	(289, 16)
Single Source Shortest Path [28]	Citation Network [39]	SSSP-citation	6524	I	23	(2612, 121)	(165, 7)
	Graph 500 [39]	SSSP-graph500	16087	I	6	(5830, 219)	(189, 28)

Table 3.3: Input characteristics.

Name	Size
Combustion Simulation [38]	Cell count: 150,000,000
Small Graph	Vertices: 1024 Edges: 1,047,552
Citation Network [39]	Vertices: 227,320 Edges: 814,134
Graph 500 [39]	Vertices: 65,536 Edges: 2,456,071
Gaussian Distribution	Array 1: 300,000 Array 2: 300,000
Uniform Distribution	Array 1: 204,800 Array 2: 204,800
Small Matrices	Matrix 1: 512*512 Matrix 2: 512*512
Large Matrices	Matrix 1: 5,120*512 Matrix 2: 512*5,120

Benchmarks: We use ten GPU DP applications from various benchmark suites. For applications whose data access patterns are highly influenced by input data, we also provide various inputs. As shown in Table 3.2, we call an <application, input> pair as one "benchmark", and there are a total of 16 benchmarks. We also provide the total number of launched kernels in Table 3.2. As one can see, these DP applications launch many more kernels compared to conventional (static) GPU applications [28], which motivates us to explore the kernel-level data reuse.

For a DP application, there can be two types of kernel launch patterns: Type I in Figure 3.1(b), and Type II in Figure 3.1(c). In type I, the launch depth for each parent kernel is 1. After the parent kernel and all its child kernels finish their executions, the host launches another barrier kernel to check some application related criteria in order to decide whether to launch another parent kernel or not. Let us consider `BFS`, where each thread in a parent kernel is responsible to traverse the edges of a node from the frontier node set that contains the nodes visited in last level of search. Based on the number of edges connected to a node, the parent thread may launch a child kernel to help traverse those edges. Before the next level of search starts, the visited edges and the frontier node set should be updated. Therefore, the host launches a barrier kernel to perform the update and also to check whether the search is over. If there are nodes not visited yet, another parent kernel will be launched from the host to perform the next level of search. Applications that have Type I feature include `BFS`, `Graph Coloring`, `Relation Join`, `Sparse Matrix Multiplication`, `Sequence Alignment`, and `Single Source Shortest Path`. In type II, the launch depth can be any number up to the hardware limit (24 levels). For example, in `Quick Sort`, the left-pivot array elements are sorted by a child kernel, whereas the right-pivot array elements are sorted by another child kernel. As a result, the kernel launch pattern is similar to a binary tree. If the maximum depth is reached or there are very few elements for processing (i.e., below the *threshold*), bubble sort is used to avoid any potential overheads involved in launching additional child kernels. Applications that have type II feature include `AMR`, `Mandelbrot Set`, `Quick Sort`, and `Radix Sort`. We list the types of each benchmark in Table 3.2. In the table, the number next to the type indicates either the number of barrier kernels in Type I, or the launch depth in Type II. We also provide the inputs used to execute our applications in Table 3.3.

3.2.4 Data Locality in DP

Prior works have shown that irregular applications exhibit data locality [62–64]. However, this data locality is hard to capture due to the dynamic, unpredictable, and divergent memory access patterns that generate it. As a result, they are not exploited well in current GPU architectures. By using DP, some of the data reuse is exposed between the boundary of kernels, WGs, and wavefronts due to child kernel launches. Specifically, a parent thread generally prepares or pre-processes the data before launching its child kernel. The child kernel operates on this data and returns the result back to the parent kernel. This "producer-consumer" relationship introduces *temporal* data locality between the parent-child kernels, WGs, and wavefronts. Meanwhile, child kernels (WGs or wavefronts) operate on neighboring data elements in the data layout. Since GPUs generally have large cachelines, *spatial* data locality among sibling-sibling kernels (WGs or wavefronts) is also exposed.

3.3 Reuse Characterization

In this section, we conduct an in-depth characterization of data reuses at different schedulable units for our benchmarks listed in Table 3.2. We profile each benchmark using GPGPU-Sim (discussed in Section 3.2.3) to get the memory footprint traces and analyze data reuse by parsing the memory traces. The memory footprints are extracted at a data block (128 Byte cache line) granularity after memory coalescing.

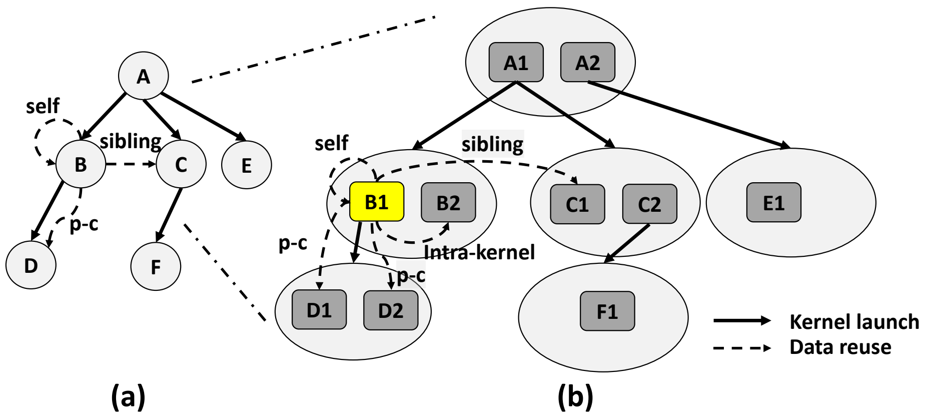


Figure 3.3: Launch sequence and data reuse. (a) kernel-level. (b) work-group-level.

3.3.1 Kernel-Level Reuse

We explore three types of kernel-level data reuse based on the kernel relationships: *self-kernel*, *parent-child*, and *sibling-sibling*. To help explain these three different types of data reuses, let us consider a representative kernel launch sequence, shown in Figure 3.3(a). The parent kernel A launches child kernels B, C and E. Child kernels B and C further launch grandchild kernels D and F, respectively. A solid line in the figure denotes the kernel launch sequence, while a broken line denotes a potential data reuse. From a particular kernel's perspective (kernel B for example), a *self-kernel* data reuse happens when a data block is referenced multiple times by itself during execution. In comparison, a *parent-child* data reuse happens when a data block is accessed at least once by both the parent kernel and child kernel (e.g., parent B and child D). Similarly, a *sibling-sibling* data reuse is said to occur when a data block is accessed at least once by both the sibling kernels (e.g., B and C). Note that sibling kernels are the child kernels launched from the *same* parent WG. For example, child kernels B and C are considered as sibling kernels; however, child kernels B and E (likewise, D and F) are not. We use this definition due to the fact that the child kernels launched by different parent WGs rarely share any data blocks and are unlikely to work on the same portion of the input data [28].

We count the number of memory accesses to the same data block made by self-kernel, parent-child kernels, and sibling-sibling kernels. Figure 3.4 plots the percentage of the shared memory footprints over the total memory footprints. Note that, a particular memory access can be counted multiple times toward different types of

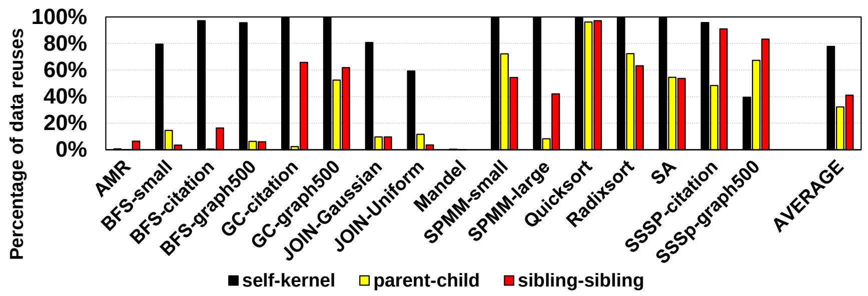


Figure 3.4: Quantifying kernel-level data reuse.

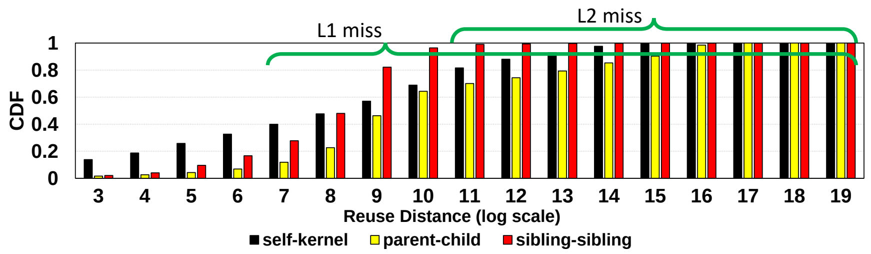


Figure 3.5: Reuse distances for kernel-level data reuse.

reuses. For example, a data block accessed by a kernel can also be accessed by its parent kernel and its siblings. As a result, the access to that data block is counted as both parent-child data reuse and sibling-sibling data reuse. This is the reason why the total sum of the three types of reuses can exceed 100% in Figure 3.4. From this figure, we make the following *critical observations*:

Observation 1: There exists significant data reuse in DP applications at a kernel granularity. Data blocks are heavily reused across all three types of kernel relationships. Specifically, on an average, across all 16 benchmarks, self-kernel, parent-child kernel, and sibling-sibling kernel account for 77.8%, 32.2%, and 41.1% of the total data reuse with respect to the total memory footprint, respectively. Recall the **Question 1** from Section 3.1, our characterization results show that data blocks are frequently reused among different kernels in these DP applications.

Observation 2: The amount of data reuse is different across different applications. For instance, applications such as `AMR` and `Mandel` do not have much data reuse for any of the three types of kernel relationships. This is because these two applications are compute-intensive with few data reuse. All the other applications have significant data reuses in different types of kernel relationships. This is due to the significant data reuse exposed by the algorithms implemented in the applications.

Observation 3: For a given application, different inputs can lead to different data reuse patterns. This can be observed in GC, SPMM, and SSSP. For example, SSSP-citation shows significant self-kernel and sibling-sibling reuses, whereas SSSP-graph500 exhibits more data reuse in parent-child and sibling-sibling kernels. This is because the irregularity of input (e.g., a graph) in such applications can cause different number of child kernels to be launched with different kernel dimensions, eventually leading to different data access and reuse patterns. For example, the average number of neighboring nodes in citation graph is relatively small when compared to the graph500 graph. As the threshold which determines child kernel launch is fixed to a smaller number in SSSP by the programmer, SSSP-citation launches fewer as well as smaller child kernels compared to SSSP-graph500. On the other hand, the child kernels in SSSP-graph500 process more neighboring nodes than the child kernels in SSSP-citation. Consequently, SSSP-graph500 has more parent-child data reuse, but less self-kernel data reuse than SSSP-citation.

Next, we perform a study on characterizing the reuse distances of the applications. We define "reuse distance" as the number of *unique* data blocks between two references to the same data block. Generally, references to the same data block with short reuse distances are expected to hit in the caches. Figure 3.5 gives the CDF of the average reuse distances (in log_2 scale) between the references to the same data block for three types of data reuses, across all the benchmarks. From the figure, we make the following observations. First, in DP applications, the reuse distances of all three reuse types are generally larger than the GPU caches can take advantage of. For example, in self-kernel reuse, more than 60% and 25% of the data blocks referenced by the kernels are evicted from L1 cache and L2 cache, respectively. Second, the distances exhibited by the parent-child data reuses are longer when compared to the distances exhibited by the self-kernel and sibling-sibling data reuses. This is because the child launch overhead (discussed in Section 3.2.2) delays the child kernel execution, leading to long distances in parent-child data reuses. This provides an answer to **Question 2** from Section 3.1. As these child kernels are generally launched in bursts, they execute concurrently, leading to shorter reuse distance for the sibling-sibling relationship.

Takeaway: Unlike traditional GPU applications (i.e., those with static/compile-time parallelism), DP applications launch massive numbers of kernels (children). In addition to the data blocks being reused within a kernel, these light-weight child kernels exhibit high data reuses with their parent kernels as well as among themselves. However, long reuse distances prevent the underlying GPU caches from taking advantage of most reused data blocks.

This motivates us to explore a locality-aware kernel scheduling strategy which can schedule kernels with high degrees of data reuse among themselves close to each other during execution. However, we first need to quantify the "degree of data reuse" between the two kernels. In other words, we need to determine the "strength" of the kernel-to-kernel relation (in terms of the "intensity" of data reuse). Kernels having intensive data reuse among themselves should have a higher priority to be scheduled together when compared to kernels rarely having any data reuse. To this end, we define *Reuse Ratio* as a measure of the degree of data reuse among kernels.

Self-Kernel Reuse Ratio (R_k): Given a kernel k, the memory footprint of kernel k is denoted as M_k. Each entry in M_k is a memory access to a data block. We define the self-kernel reuse ratio, R_k, as:

$$R_k = 1 - \frac{uniq(M_k)}{size(M_k)}, \tag{3.1}$$

where $uniq(M_k)$ is the number of *unique* data blocks referenced by kernel k, and $size(M_k)$ is the total number of data block accesses. Therefore, $1 - uniq(M_k)/size(M_k)$ captures the fraction of "repeated accesses" to the same data block.

Parent-Child Reuse Ratio (R_{p-c}): Given a parent kernel p and its child kernel c, the traces of memory footprints from parent kernel p and child kernel c are denoted respectively as M_p and M_c. The parent-child data reuse ratio, R_{p-c}, is defined as:

$$R_{p-c} = \frac{size(x, x \in M_c \mid x \in uniq(M_p) \cap uniq(M_c))}{size(M_c)}. \tag{3.2}$$

The numerator is the total number of data blocks referenced by child kernel c, where each data block is also referenced by parent kernel p at least once. The denominator is the total number of data blocks accessed by the child kernel. Note that a data block can be referenced multiple times by either the parent kernel or the child kernel. Our definition of parent-child reuse ratio captures the "intensity" of data reuses in a child kernel with respect to its parent. Note that, we do not count child-parent kernel reuse ratio as child kernels need to be launched by their parent kernels. For this to happen, the parent kernels already need to be scheduled. Therefore, child-parent kernel reuse ratio does not help in dynamic kernel scheduling. Since a DP application can launch multiple levels of child kernels, a child in level l is considered as a parent in level $l+1$.

Sibling-Sibling Reuse Ratio (R_{cx-cy}): Given two child kernels c_1 and c_2 launched by the *same* parent WG p_{wg}, the memory footprints of these two child kernels are denoted as M_{c1} and M_{c2}, respectively. We define the sibling-sibling data reuse ratio, R_{cx-cy}, as:

$$R_{cx-cy} = \frac{size(x, x \in M_{cx} \mid x \in uniq(M_{cx}) \cap uniq(M_{cy}))}{size(M_{cx})}, \tag{3.3}$$

where (cx, cy) can be either $(c1, c2)$ or $(c2, c1)$ which represent the reuse ratio in terms of child kernel $c1$ or $c2$, respectively. It is important to note that, we separate the reuse ratios for the two child kernels in the child kernel pair because doing so allows us to capture the scenario where two child kernels have different memory footprint intensity. For example, let us consider the scenario in Figure 3.7. Suppose child $c1$ has 100 accesses, child $c2$ has 50 accesses, and child $c3$ has 200 accesses. Let us assume that the data blocks referenced by 30 accesses from $c2$ are also referenced by $c1$. In this case, the reuse ratio R_{c2-c1} is $30/50 = 0.6$. However, there can be 40 accesses from $c1$ that reference the same set of data blocks. This is due to the fact that a data block can be referenced multiple times by a kernel. As a result, the reuse ratio R_{c1-c2} is calculated as $40/100 = 0.4$.

It is important to emphasize that the reuse ratio does *not* capture the absolute number of data blocks being shared between the involved schedulable units. For example, both R_{c2-c1} and R_{c3-c1} are 0.6. However, there are 120 data blocks accessed by $c3$ that are also accessed by $c1$, whereas only 30 data blocks accessed by $c2$ that are also accessed by $c1$. Although the absolute value would be more accurate to represent the quantity of data reuses, it is less effective in managing caches. For instance, a kernel pair having one million total accesses with a 10% reuse ratio will have many more data blocks being reused when compared to a kernel pair having one thousand total accesses with 90% reuse ratio. However, the first kernel pair is not friendly to caches: there are 90% of data blocks that are not being reused, and therefore, it may lead to severe cache contention and poor cache performance.

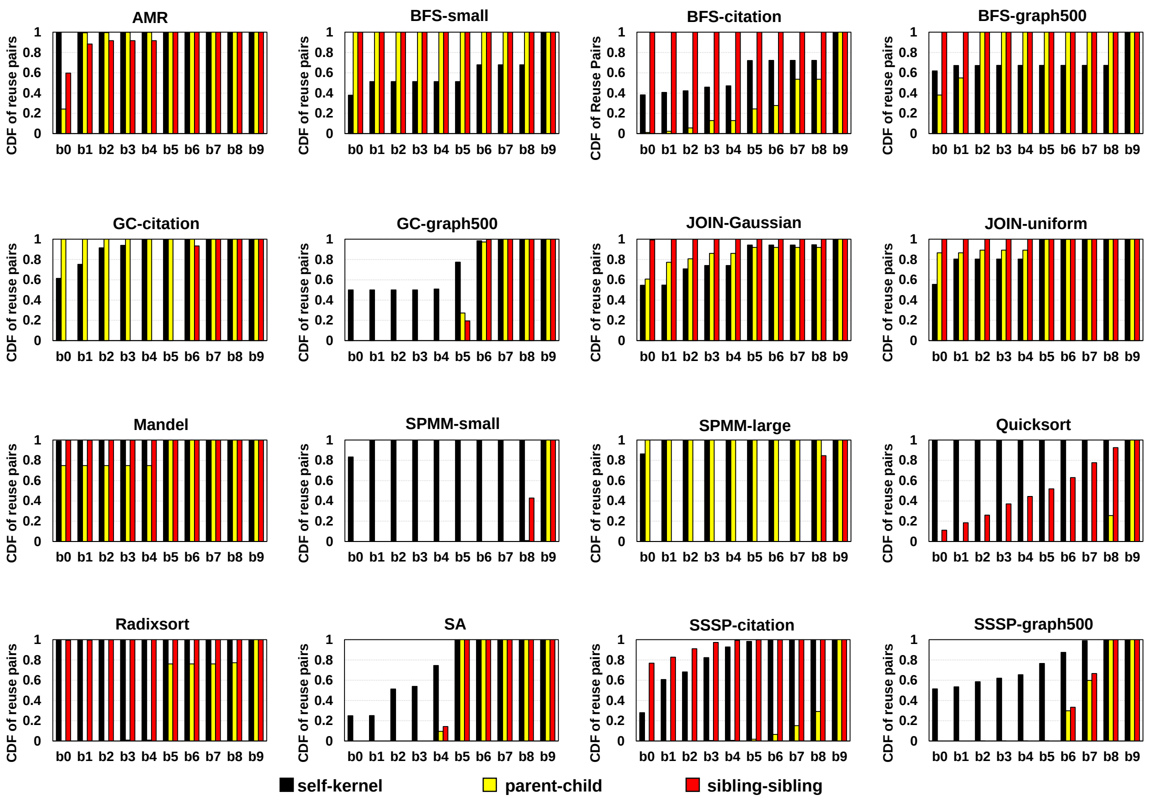


Figure 3.6: Kernel-level data reuse ratios for all benchmarks. Each plot includes the results of one benchmark. The X-axis represents the data reuse ratio. The reuse ratio is divided into 10 bins (b0, b1, ..., b9), using 0.1 as stride size. If the reuse ratio of a kernel pair is between $(x,x+0.1)$, that kernel pair is counted in bin bx. The Y-axis represents the CDF of reuse pairs. The black bar in each plot represents the CDF of self-kernel reuse, whereas the yellow and red bars represent the CDFs of parent-child and sibling-sibling reuses, respectively.

Results: Figure 3.6 shows the data reuse ratio at the kernel-level for all 16 benchmarks. We divide the data reuse ratio into 10 ratio bins ($b0--b9$), with the stride size of 0.1. All 10 ratio bins are labeled on the X-axis. The Y-axis plots the CDF of the kernel pairs, which captures the number of kernel pairs that fall into the different ratio bins. Specifically, for two kernels in a kernel pair, we first calculate the three types of data reuse ratios using Equations (1)–(3) given above. Then, for each ratio bin, we count the number of kernel pairs whose reuse ratio falls into that bin. For example, if two kernels in a kernel pair have, say, 0.46 parent-child reuse ratio, we count this kernel pair in the ratio bin $b4$. From the cumulative results shown in Figure 3.6, we make the following important observations:

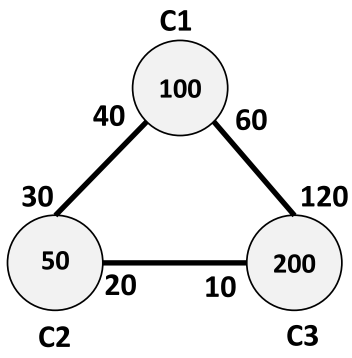


Figure 3.7: Sibling-sibling data reuse.

Observation 1: Different benchmarks show different kernel pair distributions in all three types of kernel reuse ratios. For benchmarks `AMR` and `Mandel`, most of the kernel pairs have low self-kernel reuse ratio (less than 0.1) as these two benchmarks have few data blocks being reused (Figure 3.4). For benchmarks such as `GC-citation`, `GC-graph500`, `SPMM-small`, `SPMM-large`, `Radixsort`, `SA`, `SSSP-citation` and `SSSP-graph500`, most of the parent-child kernel pairs and/or sibling-sibling kernel pairs have similar reuse ratios. For example, in `SA`, 85% of parent-child kernel pairs fall in to $b5$. Benchmarks such as `Quicksort`, `BFS-small`, `BFS-citation`, `BFS-graph500`, `JOIN-Gaussian` and `JOIN-Uniform`, have a uniform distribution in one or more of their kernel relationships. For example, in `Quicksort`, the sibling-sibling kernel pairs are uniformly distributed across the 10 reuse bins.

Observation 2: For applications such as `SPMM`, `BFS`, `GC`, and `SSSP`, different inputs can lead to different distributions of kernel pairs in terms of their reuse ratios. For example, when `SPMM` is used with small sparse matrices (`SPMM-small`) as inputs, we see that there is a significant fraction of parent-child and sibling-sibling kernel pairs having high reuse ratios, whereas, with large sparse matrices (`SPMM-large`) as input, the number of parent-child kernel pairs with high reuse ratio reduces significantly (i.e., most of the parent-child kernel pairs fall into bin $b0$). The main reason is that the *threshold* (discussed in Section 3.2) set by the programmer significantly affects the number of child kernels and their properties. Therefore, for large matrices, parent thread always opts to launch child kernels to perform the multiplications in child kernels (due to more elements per row and column), whereas for small matrices, the multiplications are usually performed by the parent thread itself in an iterative fashion as there are fewer elements in the rows and columns of the input matrices.

Observation 3: By comparing Figure 3.4 and Figure 3.6, we observe that a benchmark with a high "data reuse" does *not* necessarily have high "reuse ratio". This is because, data reuse is measured as an *aggregated* metric, whereas the reuse ratio is measured as a *pair-wise* metric. For example, `SPMM-small` has high self-kernel data reuse than parent-child and sibling-sibling kernel data reuses (Figure 3.4). However, all of the parent-child kernel pairs fall into $b9$ in Figure 3.6 and all sibling-sibling kernel pairs fall into $b8$ and $b9$, but most of the self kernel pairs fall into $b0$ and $b1$. This discrepancy is because the reuse ratio is normalized to the memory footprints of two kernels whereas data reuse is computed using the memory footprint of the entire application. As we discussed earlier, reuse ratio is a more effective metric in managing caches.

Takeaway: Our results show that DP applications introduce parent-child and sibling-sibling kernel relationships in addition to the self-kernel relationship. The reuse distance analysis indicates that GPU cache system is unable to take advantage of inherent data reuses in DP applications. We define and analyze reuse ratio to indicate the "intensity" of the data reuses among kernels, which is used later to design a locality-aware kernel scheduling mechanism.

3.3.2 Work-group-Level Reuse

Work-groups are the smallest granularity of scheduling at the CU level. Note that, while it may seem possible to extend the kernel level data reuse information (discussed above) to the WG level, it is not the case. This is due to the fact that the kernels from a given application might have different number of WGs, and some of these WGs contribute to significant data reuse whereas others may not. Conceptually, we need to understand the data reuses among WGs in order to map WGs to CUs in a "locality-aware" fashion, and schedule them to execute close to each other in time to take full advantage of the per CU L1 cache. Similar to kernel level data reuse, we explore data reuse along four types of WG relationships: *self-WG*, *intra-kernel-WG*, *parent-child-WG*, and *sibling-sibling-WG* (shown in Figure 3.3(b)). Self-WG reuse and intra-kernel-WG reuse are defined within a kernel boundary (i.e., intra-kernel), whereas parent-child-WG reuse and sibling-sibling-WG reuse are defined across different kernels (i.e., inter-kernel). Specifically, for a given WG ($B1$ from kernel B highlighted in Figure 3.3(b)), self-WG reuse captures the fraction of data blocks that are accessed multiple times by that same WG. Intra-kernel-WG reuse captures the data blocks reused between the WGs within the same kernel ($B1$ and $B2$). Parent-child-WG reuse is measured between the parent WG ($B1$ in the example) and all WGs from its launched child kernels ($D1$ and $D2$). Finally, sibling-sibling-WG reuse captures the data blocks reused among the WGs that belong to the sibling kernels (e.g., $B1$ and $C1$).

Self-WG Reuse Ratio: Similar to self-kernel reuse ratio calculation, self-WG reuse ratio can be calculated using Equation (1) by replacing kernel with WG. Specifically, for a particular WG wg, $R_{wg} = 1 - uniq(M_{wg})/size(M_{wg})$.

Intra-kernel-WG Reuse Ratio: Given two WGs wg_i and wg_j from a kernel k, we define intra-kernel-WG reuse ratio using Equation (3) by replacing (cx, cy) with (wg_i, wg_j). We want to emphasize that this equation calculates the reuse ratio in terms of WG wg_i. That is, the ratio $R_{wg_i - wg_j}$ may *not* be the same as $R_{wg_j - wg_i}$.

Parent-Child-WG Reuse Ratio: Given a parent WG wg_p and a WG wg_c from a child kernel launched by wg_p, we define the parent-child-WG reuse ratio using Equation (2) by replacing p and c with wg_p and wg_c, respectively. One may notice that this definition only captures the reuse with respect to child WG wg_c. Ideally, we should also calculate the reuse ratio with respect to that parent WG. However, this is not necessary in practice. This is because, in order to schedule a child WG with its parent WG, the child kernel should be first launched and be ready to execute. That is, for a child WG to be visible to the scheduler (i.e., for a child kernel to be launched by the parent WG), its parent WG must be already scheduled and running on the CU. Thus, we just need to focus on where to schedule the child WG based on the child WG reuse ratio, not the parent WG, as it would be already running.

Sibling-Sibling-WG Reuse Ratio: Sibling-sibling-WG reuse ratio is measured among the WGs that belong to sibling kernels. The formal definition is similar to that of the sibling-sibling kernel reuse ratio and can be obtained by replacing the kernel information with WG information. It can be treated as a finer granularity

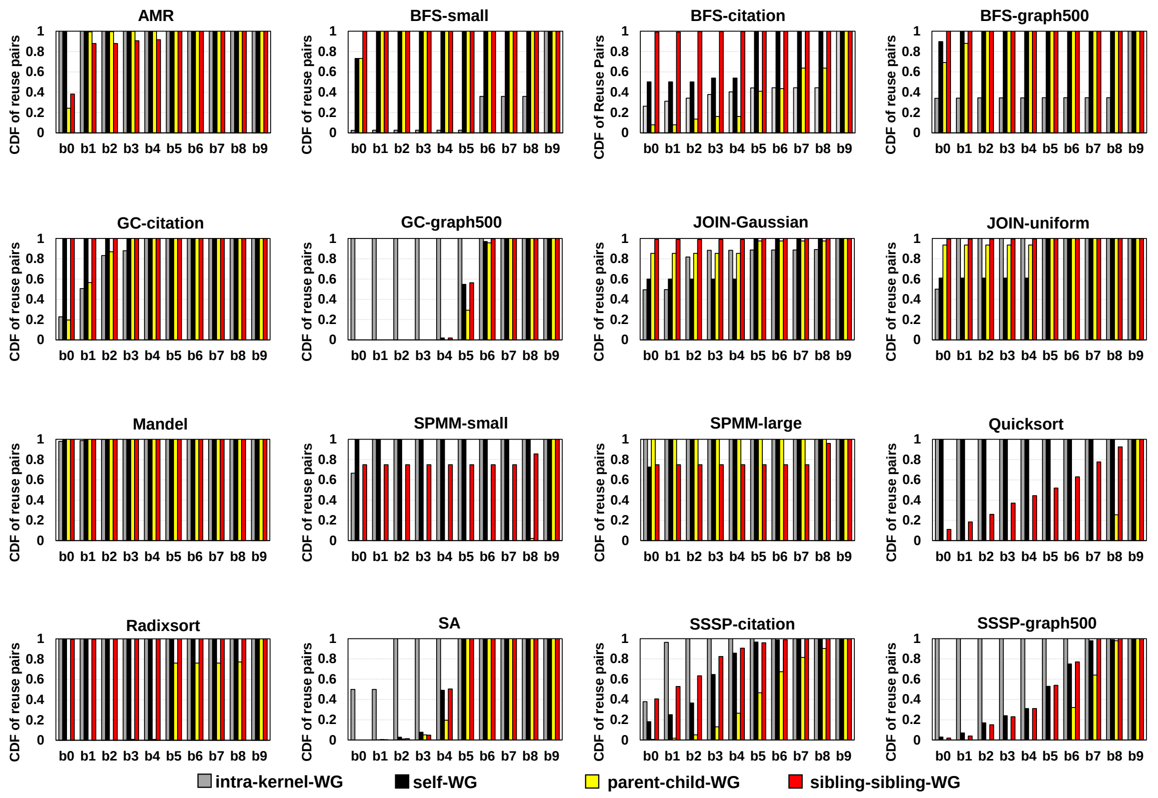


Figure 3.8: WG-level data reuse ratio. The X-axis represents 10 bins (b0, b1, ..., b9) of data reuse ratio with 0.1 as stride size. The Y-axis represents the CDF of WG pairs. The gray bar captures intra-kernel-WG. The black bar represents the self-WG, and the yellow and red bars represent parent-child-WG and sibling-sibling-WG, respectively.

of data reuse compared to the sibling-sibling kernels and can be calculated using Equation (3).

Results: Figure 3.8 shows the cumulative distribution of WG pairs for each benchmark. The four bars in each plot of this figure represent the four types of WG relationships.

We make the following observations from these results:

Observation 1: Similar to kernel level, the WG pair distributions of intra-kernel-WG, self-WG, parent-child-WG and sibling-sibling-WG are also application and input dependent.

Observation 2: By comparing Figure 3.6 and Figure 3.8, we observe that, in benchmarks `AMR`, `BFS-citation`, `BFS-graph500`, `GC-graph500`, `JOIN-Gaussian`,

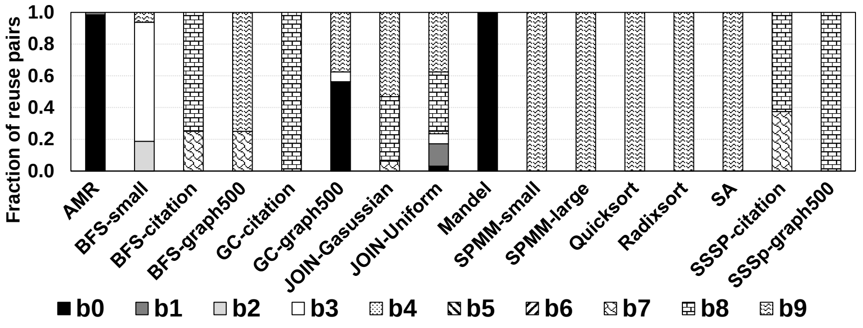


Figure 3.9: Intra-wavefront data reuse.

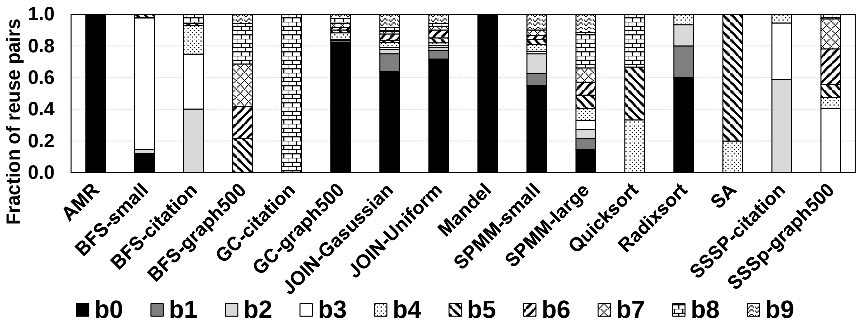


Figure 3.10: Inter-wavefront data reuse.

`JOIN-uniform`, `Quicksort`, `Radixsort`, `SA` and `SSSP-graph500`, the distribution of the WG pairs on a particular type of reuse ratio follows a similar trend to the distribution of kernel pairs on the same type of reuse ratio. In other words, the reuse ratio can translate from kernel level to WG level, due to the fact that a kernel consists of multiple WGs. It is also interesting to observe that self-kernel reuse can translate to either intra-kernel-WG reuse, or self-WG reuse, or both. For example, in `BFS-graph500` (see Figure 3.6 and Figure 3.8), the self-kernel reuse translates to intra-kernel-WG reuse. However, in `BFS-citation`, self kernel reuse translates to both intra-kernel-WG and self-WG reuse. This is because the child kernels contain more WGs in `BFS-graph500` compared to the child kernels in `BFS-citation`. For benchmarks `BFS-small`, `GC-citation`, `Mandel`, `SPMM-small`, `SPMM-large` and `SSSP-citation`, the WG pair distribution is different from the kernel pair distribution. The is because that most of these applications contain branches in their kernel codes. Based on the inputs, the runtime branch conditions are different across WGs in a kernel. As a result, WGs may execute different paths, leading to different data reuse patterns at the WG level.

Observation 3: If two kernels do not reuse data blocks, all the WGs from these two kernels do not reuse data blocks either. For example, `Radixsort` and `SPMM-small` have very few kernels with high self-kernel reuse ratio. Consequently, these two benchmarks also have very few WGs with high self-WG and intra-kernel-WG reuse ratios[2]. However, this is not true in the inverse case. If two kernels have high data reuse between them, this does *not* guarantee that all the WGs from the two kernels will have high data reuse. For instance, more than 50% of the sibling-sibling kernels in `SPMM-small` have reuse ratios larger than 90%. However, at the WG granularity, 75% of the sibling-sibling WGs have reuse ratio less than 10% and the remaining 25% have reuse ratio more than 80%. This disparity arises due to the branch instructions in the kernel code, which lead to divergent paths across WGs, as discussed in Observation 2.
Takeaway: Our results indicate that not only neighboring WGs significantly share data blocks (**Question 3**), but also the parent-child WGs and sibling-sibling WGs (**Question 4**). This information is helpful in assisting WG-to-CU mapping. Specifically, WGs with high reuse ratios should be scheduled on the same CU and executed in close proximity in time, in order to take advantage of the per-CU L1 cache. On the other hand, WG with high self-WG reuse ratios should be scheduled to low-load CUs in order to reduce the L1 cache contention.

3.3.3 Wavefront-Level Reuse

Wavefront is the smallest schedulable unit and it is the granularity at which the GPU executes instructions. Therefore, wavefront scheduler can impact data locality significantly. To quantify the data reuse in wavefronts, we characterize the *intra-wavefront* and the *inter-wavefront* data reuses for all our benchmarks. For a wavefront w, the intra-wavefront reuse is quantified using Equation (1) by replacing kernel k with wavefront w. Similarly, given two wavefronts w_x and w_y, the inter-wavefront reuse is quantified using Equation (3) by replacing (cx, cy) with (w_x, w_y). Intuitively, one can still define self-wavefront, parent-child-wavefront, and sibling-sibling-wavefront relationships and analyze data reuses along those relationships. However, we choose to classify data reuses into inter-wavefront and intra-wavefront reuses, due to following

[2]Both self-WG and intra-kernel-WG are within a kernel boundary, and a low self-kernel reuse leads to a low self-WG and intra-kernel-WG reuses.

two reasons. First, wavefronts are mapped to CUs at a WG granularity. In other words, all wavefronts from the same WG are mapped to a particular CU at the time that WG gets scheduled. Second, once mapped to a CU, wavefronts cannot migrate or be reassigned to other CUs, as WG migration is not supported. These two reasons limit the capability of scheduling “any” set of wavefronts together on the same CU. For example, suppose that two wavefronts from two sibling WGs have high inter-wavefront reuse between them. In order to convert the data reuses between these two wavefronts into data locality (cache hits), the two sibling WGs have to be scheduled first. As there are limited hardware resources, it is impossible to schedule all the WGs together to have all the wavefronts ready for the wavefront scheduler. Once both the WGs are scheduled, it is possible for the wavefront scheduler to exploit the reuse between these two wavefronts. This is captured as inter-wavefront locality. We characterize the intra-wavefront and inter-wavefront reuse ratios in Figure 3.9 and Figure 3.10, respectively. We divide reuse ratio into 10 bins ($b0 - b9$), and for each benchmark, the Y-axis reports the percentage of wavefront pairs that fall into different bins.

Takeaway: Overall, DP applications have significant number of wavefronts exhibiting intensive intra-wavefront reuse. Moreover, there is also a sizable fraction of wavefronts showing intensive inter-wavefront reuse. Since a wavefront can offload its computations to other wavefronts by launching child kernels, some of the intra-wavefront data reuses are transferred to inter-wavefront data reuse between parent wavefront and child wavefronts [28].

3.4 Optimal Locality-aware Scheduler: A Limit Study

In this section, we propose an “optimal” scheduling mechanism to realize the maximum potential performance gains that can be achieved by leveraging data reuses in DP applications. Note however that, this optimal scheduler only has *a priori* knowledge about the data reuse patterns in the DP benchmarks and cannot change any application (data dependency, correctness, etc.) or hardware (occupancy limits, cache replacement policies, etc.) constraints to improve data locality. For this limit study,

Algorithm 2 Locality-aware kernel scheduling

```
INPUT:
    List_k : kernels launched in pending kernel pool
1: while HWQ $q_i$ is empty do
2:     for kernel $k_x$ from the head of HWQ $q_j$, where $i \neq j$ do
3:         Search for high reuse kernels with $k_x$
4:         $high\_reuse(k_x) \leftarrow (k_{y1}, k_{y2}, ..., k_{yn})$
5:         if $k_{y1}$ is also in List_k then
6:             schedule $k_{y1}$ to $q_i$ and remove $k_{y1}$ from List_k
7:         end if
8:     end for
9: end while
```

we profile all the benchmarks and analyze the reuse ratio among the schedulable units at kernel, WG and wavefront granularities. Then, we discuss scheduling policies that choose the appropriate schedulable units based on the reuse ratios. Note that, the scheduling policies in this limit study are *not* implementable in practice. Instead, they reveal the "optimal" benefits one can get from realizing data reuse in DP applications.
Challenges: There are several challenges involved in building an optimal scheduling mechanism. First, hardware constraints limit the effectiveness of the schedulers. For example, high reuse kernels should be launched to different HWQs for concurrent execution, but the number of HWQs limits the number of kernels that can execute concurrently. Second, in all the three granularities of reuses studied, high reuse schedulable units can form "reuse chains". For example, if kernel $k1$ has a high reuse ratio with kernel $k2$, and $k2$ also has high reuse with kernel $k3$, all three kernels collectively form a *high reuse chain*, $k1$-$k2$-$k3$. We quantify maximum as well as average high reuse chain length at kernel and WG granularities (shown in Table 3.2). For the limit study, we define "high reuse" as a reuse ratio of greater than 0.4. Since there are limited hardware resources (e.g., register file) and limited concurrency, it is impossible to always schedule the entire chain to hardware for concurrent execution. For example, in `Radixsort`, the maximum WG chain is 176, and as a result, it is not possible to find a CU and schedule all the 176 WGs in that CU. Third, scheduling an entire reuse chain of WGs into a CU can also lead to workload imbalance, causing some of the CUs to have more computations, while other CUs are idle. The WG scheduler should be able to dynamically *balance* the workload among CUs without compromising much on data locality.

Algorithm 2 provides the high-level pseudo-code for optimal kernel scheduler. Whenever there is an empty slot in any HWQ, the kernel scheduler tries to find a candidate kernel which has the highest reuse ratio with the already-running kernels, and schedules it into the head of that empty HWQ. In other words, it tries to *maximize* the potential chances of running high reuse kernels concurrently. A significant benefit of our scheduling strategy is that it facilitates the subsequent locality-aware WG scheduling. More specifically, the WG scheduler is free and safe to choose any WGs from these high reuse kernels and schedule the selected WGs into CUs as long as there are enough hardware resources. It should also be noted that, dependencies among parent kernels and child kernels are rare in dynamic applications [26]. However, if there is a dependency, the dependent kernels are labeled with same software queue ID (e.g., CUDA stream), which in turn is mapped to the same HWQ for sequential execution. Our scheduler tries to co-locate parent and child kernels during scheduling such that the data reuse between dependent kernels are captured.

Algorithm 3 shows the WG scheduling policy used in our limit study. We define reuse ratio to be "high" if it is greater than 0.4. We form the high reuse chains of WGs based on the characterization results and import the high reuse chains to Algorithm 3 for scheduling. To take high reuse WG chains into consideration, we associate each CU with a CU queue. At runtime, we schedule high reuse WG chains into these CU queues. Note that the WGs in CU queues do not occupy CU resources (e.g., hardware threads, register file). Once a CU has sufficient available resources, it selects a candidate WG from its CU queue. Note that, scheduling an entire WG chain onto a single CU can lead to load imbalance (in terms of the number of WGs assigned) across different CUs. This is due to the different lengths of the WG reuse chains and can lead to significant GPU under-utilization and performance degradation. To avoid load imbalance, every time a WG chain is to be scheduled, all the CU queues lengths are checked. The WG chain is assigned to the CU queue with the least number of WGs in its queue. To tackle the issue of load imbalance, we enable *WG stealing* across the CU queues. Specifically, if a CU has available resources and its associated CU queue is empty, it steals a WG from another CU queue. Note that WGs that have been scheduled and are executing on a CU cannot migrate.

Once a WG is scheduled on a CU, the wavefronts in the WGs are mapped to the hardware wavefronts. There are a total of 64 hardware wavefronts in our baseline GPU, and the default wavefront scheduler is GTO [46]. GTO scheduling performs well

Algorithm 3 Locality-aware WG scheduling

```
INPUT:
    $Q_{CU_i}$: The CU queue of $CU_i$.
1: for each $Q_{CU_i}$ do        /**schedule reuse chains to CU queues*/
2:     if size of $Q_{CU_i}$ is minimum then
3:         schedule reuse chain $(wg_1, wg_2, ..., wg_n)$ to $Q_{CU_i}$
4:     end if
5: end for
6: for each $CU_i$ do     /**schedule WGs on CU*/
7:     if $CU_i$ can issue a WG then
8:         if $Q_{CU_i}$ is empty then        Steal an WG from other CU queue.
9:         else   issue WG from $Q_{CU_i}$
10:        end if
11:    end if
12: end for
```

in achieving intra-wavefront data locality. However, it is not as effective in exploiting inter-wavefront data locality. To address this, we enhance the two-level wavefront scheduler [65]. More specifically, if two WGs have high reuse ratio, the wavefronts from these two WGs are grouped together and executed in a round-robin fashion. To select between the groups, GTO is used. For example, let us assume four WGs: $wg1 = (w11, w12)$, $wg2 = (w21, w22)$, $wg3 = (w3)$, and $wg4 = (w41, w42)$. Suppose that work-groups $wg1$, $wg2$ and $wg3$ have high data reuse ratio among themselves, and $wg4$ has high self-WG reuse. Let us further assume all four WGs are running on the same CU. In this case, we make two groups of wavefronts based on the WG reuse ratios: $group1 = (w11, w12, w21, w22, w3)$ and $group2 = (w41, w42)$. First, GTO is used to select between the two groups, and once the group is selected, the wavefronts within the group are executed in a round-robin fashion.

Note that, although we use the profiled data reuse information to guide our scheduling, we still cannot achieve the optimal data locality along with the optimal performance. The reason for this is two-fold. First, there is a tradeoff between parallelism (workload balance) and data locality. For instance, to guarantee CU occupancy, we have to schedule a WG whenever a CU has available resources. Consequently, if a high reuse WG from a child kernel is not available to the scheduler (e.g., due to launch overhead), we do not want to reserve the CU resources while waiting for a high reuse WG to be scheduled. This is because leaving a CU under-utilized reduces the effectiveness on tolerating long latency operations and leads to performance degradation. Second, GPUs usually have smaller caches compared to CPUs. Even with our locality-aware scheduling strategies, it is not guaranteed that all of the reused data blocks can remain in the cache when they are reused. To further improve data locality in GPUs, our scheduling strategies can co-exist with other locality-aware cache optimizations [55, 56, 66].

Implementation Issues: It is not feasible for the optimal scheduler to be implemented in practice. This is due to two main reasons. First, the data reuse information that is needed at runtime is not known *a priori*. Unlike regular applications, where profiling some *training* applications/inputs to build a prediction model can help predict the reuse information for new applications/inputs [67, 68], it is not possible to do the same for DP applications. This is due to the irregular and unstructured behavior of the applications and their inputs, as discussed in Section 3.3. Second, even if the data reuse information was known *a priori*, the bookkeeping and hardware overheads of implementing an optimal scheduler in practice would be too high due to the increase on area and power costs.

3.5 LASER – Locality-Aware SchedulER: A Practical Approach

In this section, we distill the observations from our data reuse characterization and limit study presented above, and propose LASER, a Locality-Aware SchedulER, that makes scheduling decisions based on the reuse ratios. The reuse ratios are computed dynamically at runtime with minimal hardware overheads and no profiling requirements. Figure 3.11 depicts the necessary architectural support required to implement LASER. We modify the baseline GPU architecture by extending/adding components in the GMU, WG scheduler, and CUs.

GMU: In the baseline GPU, the newly launched kernels (device kernels) are either directly launched into the HWQs in the GMU or temporarily "stored" in the pending kernel pool (a queue based structure) if there are no empty slots in HWQ. We partition the pending kernel pool to have two priority queues: High-Priority Queue (HPQ) and Low-Priority Queue (LPQ) Ⓑ. A kernel is queued to either HPQ or LPQ based on the priority flag associated in the kernel instance Ⓐ. The priority flags of child kernels are set at the time the parent thread launches the child kernels. The priority values are determined based on the outputs of reuse monitors (discussed later in this

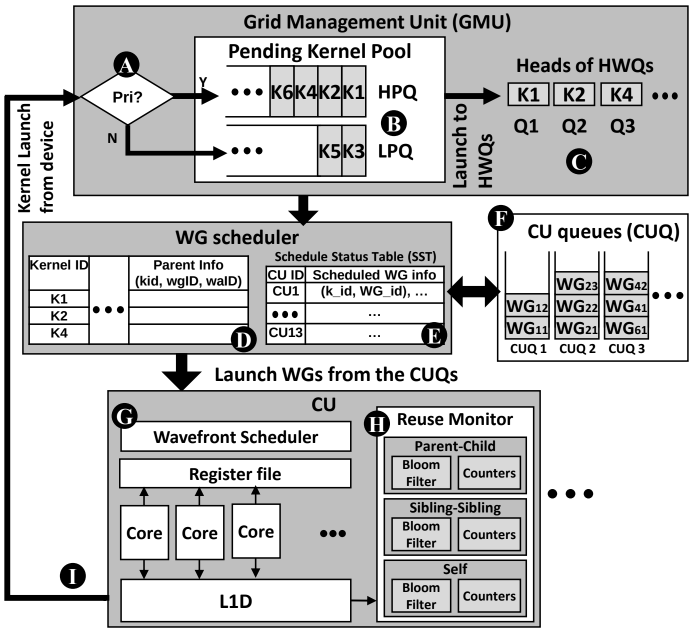


Figure 3.11: Architectural support for LASER.

section) located in each CU. Once there is an empty slot in HWQ, the GMU selects the kernel at the head of HPQ and only selects a kernel from LPQ when HPQ is empty C. Since we partition the pending kernel pool into HPQ and LPQ without increasing the pool capacity, the only incurred hardware overhead is adding one more read port and one more write port to the pending kernel pool.

WG scheduler: Recall that, in the baseline GPU scheduler, WGs from "head-of-queue" kernels in HWQs are scheduled to CUs in a round-robin fashion. The WG scheduler keeps tracks of the necessary WG scheduling information such as next WG to be scheduled and WG dimension for every kernel. In LASER, we extend the information table to include the parent information (i.e., parent kernel ID and WG ID D). We also add a new table called Schedule Status Table (SST) in the WG scheduler to track the running WGs on each CU E. Each entry in SST contains the information of the running WGs in the form of (k_id, WG_id) pair. Since each CU can have a maximum of 16 WGs resident [37], each table entry for a CU contains information from a maximum of 16 WGs. Therefore, we need 1664 bytes for the hardware SST [3]. A (k_id, WG_id) pair is inserted into the SST once the WG is

[3]1664 bytes is calculated by 13 CUs with each CU has maximum 16 WGs. For each WG, we track k_id and WG_id with each 4 bytes.

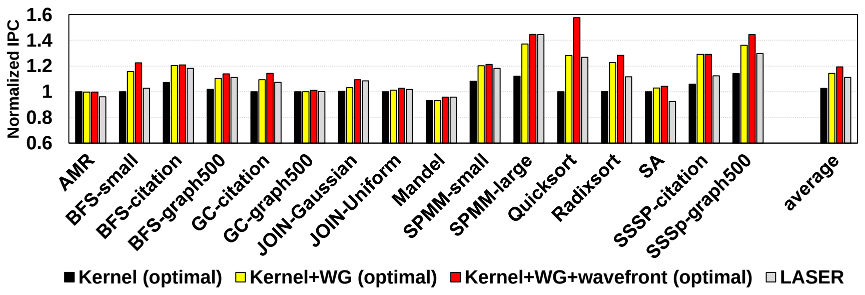


Figure 3.12: IPC normalized to baseline scheduling.

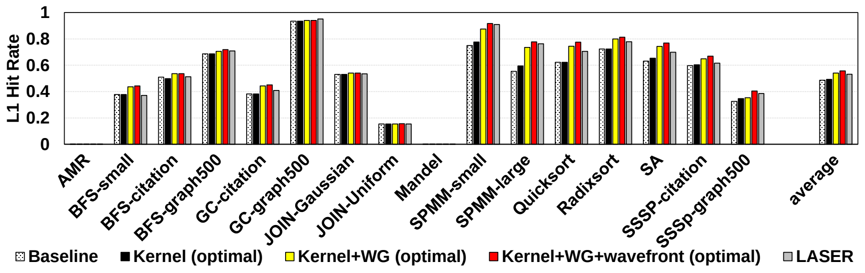


Figure 3.13: L1 hit rates.

scheduled to a CU and removed from the SST when the WG finishes execution and relinquishes its occupied resources. For scheduling a child WG, the WG scheduler relies on its parent's information and the information from SST, and schedules it on a CU where the parent WG is already running. As a result, parent-child WG reuse is captured. Recall that multiple WGs can have high data reuse among them and form reuse chains. To preserve the data reuse in reuse chains, we add a CU queue (CUQ) F for each CU. CUQs serve two purposes: (1) to ensure that the high reuse WGs are executed by the same CU as much as possible, and (2) to enable WG stealing across different CUs in order to avoid workload imbalance. The CUQs are mapped onto CUs in a one-to-one fashion. The WG scheduler always tries to schedule WGs to a CU from its CUQ (e.g., CUQ 1 to CU 1), and a CU only steals a WG from another CUQ if its CUQ is empty, and it has enough available resources for a new WG to be scheduled. We implement CUQs in the GPU's global memory such that the hardware overheads are minimized [47].

Compute Unit (CU): We modify/add two components in each CU: (1) the wavefront scheduler G, and (2) the reuse monitor H. As discussed in Section Section 3.4, we use a two-level wavefront scheduler to leverage wavefront-level data reuse. Specifically, if a child kernel is predicted to have high parent-child data reuse, all the child wavefronts are grouped with its parent wavefront (if it is not finished yet) in the same group and round-robin wavefront scheduler is applied within the group. Note that, as the majority of child kernels are light-weight [28] (i.e., they contain few WGs and each WG has few wavefronts), the total number of wavefronts in a child kernel is small, making the group size relatively small.

Reuse Monitor: Whenever there is a child kernel launch from a CU, we estimate the reuse type (i.e., parent-child, sibling-sibling, and self) of that particular child kernel. This is done through the information obtained from the reuse monitor 🅗 in each CU. The reuse monitor consists of three Bloom filters, one for each type of data reuse: parent-child kernel, sibling-sibling kernel, and self kernel reuse. Each Bloom filter is associated with two counters: number of hits and number of misses. For each L1 cache read request, the accessed cacheline address is checked in the Bloom filter and the corresponding counters are updated. The address is then added to the Bloom filter based on the kernel type (e.g., only parent kernel adds to the parent-child Bloom filter and only child kernel adds to the sibling-sibling Bloom filter). The reuse monitor predicts the child kernel to be of parent-child reuse type if the "hit rate" of the parent-child Bloom filter is greater than a predefined threshold, and is larger than the hit rates of sibling-sibling and self-kernel. We empirically set the threshold to be 0.5. Similarly, for sibling-sibling and self-kernel, the reuse monitor predicts the reuse type based on the corresponding hit rates of Bloom filters. If a child kernel is predicted to be parent-child type or sibling-sibling type, the kernel is labeled a high-priority. Otherwise, if it is predicted self-reuse type or no reuse, it is labeled a low-priority. We use MurmurHash2 [69] as the hash function in the Bloom filter.

At the beginning of application execution, the very first parent kernel is launched by the host CPU and pushed into HPQ in the GMU. Due to the lack of reuse information at the initial stage of execution, the WGs in that parent kernel are scheduled to CU queues in a round-robin fashion for load balance. In order for the reuse monitor to capture parent-child WG reuses, the first couple of child kernels are launched to the HPQ and child WGs are scheduled to the same CUs where the parent WGs execute. After this stage, further kernel launches are attached with estimated priorities based on information from reuse monitor 🅘. The kernel's priority is checked at 🅐 and is either pushed into the HPQ or the LPQ 🅑. When a parent kernel finishes its execution, we reset the Bloom filters but leave the counter values unchanged. Later, when the next parent kernel starts launching child kernels, it uses the counter information to perform the priority prediction as well as update the counters and Bloom filters. The major hardware overheads come from the structure of Bloom filters which are space-efficient data structures. A Bloom filter does not need to account for a lot of entries, especially in applications that exhibit frequent data reuse, since only the misses are added to Bloom filter and hits just update the counters. In LASER, each Bloom filter needs 605 bytes (with 1000 entries and 0.1 false positive probability). As a result, the total hardware overhead is 2 KB for the 3 Bloom filters per CU.

3.6 Experimental Evaluation

In this section, we evaluate the effectiveness of our limit study and our proposed scheduling mechanism, LASER. We also compare against two prior efforts targeting data locality on GPUs.

Results of the limit study: The first three bars in Figure 3.12 show the normalized IPC across all 16 benchmarks from our limit study. The results are *normalized* to the baseline scheduler which uses FCFS kernel scheduling, round-robin WG scheduling, and GTO wavefront scheduling. The first three bars in Our limit study is a three-part study which involves an optimal locality-aware kernel scheduling policy(**kernel**), an optimal locality-aware kernel+WG scheduling policy (**kernel+WG**) and finally, an optimal locality-aware kernel+WG+wavefront scheduling (**kernel+WG+wavefront**). On an average, the three policies achieve performance improvements of 2.6%, 14.3%, and 19.4%, respectively with respect to the baseline scheduling policy. From the results, we make the following observations. First, for most benchmarks having high data reuses in Figure 3.4, such as three inputs of `BFS` and both inputs of `SSSP`, the performance improvements are also high compared to other benchmarks. This is because we schedule the units (i.e., kernels, WGs, and wavefronts) with high reuse ratios close to each other during execution. Second, for applications `AMR` and `Mandel`, the results are similar to baseline since these two applications do not have intrinsic data reuse properties. Third, for benchmarks such as `JOIN-Uniform` and `SA`, even though we use accurate reuse information to guide scheduling, some of the data reuse opportunities may not be achievable, especially data reuse along the relationship of parent-child. This is primarily due to the launch overhead. Specifically, child kernels are not available in pending kernel pool immediately after launch. Since our approach can only choose the kernels from the pending kernel pool which contains only ready-to-execute kernels, we fail to exploit some of the parent-child data reuses.

Note that, we do not enable throttling in any of our experimental evaluation. Throttling has been proved to be very beneficial in reducing the cache contention [57, 70]. With throttling enabled, we expect both, the limit study and LASER to provide better performance improvements as it reduces the cache contention.

Evaluation of LASER: The last bar in Figure 3.12 shows the overall performance of LASER, *normalized* with respect to the baseline scheduler. LASER, on an average, achieves 11.3% performance improvement across the 16 GPU benchmarks tested. We make two observations from the results. First, in benchmarks `BFS-citation`, `SPMM-small`, and `SPMM-large`, LASER performs very well (close to the "optimal"). This is because most of the parent-child kernels pairs and/or sibling-sibling kernel pairs have similar high reuse ratios (i.e., fall into the same bin as shown in Figure 3.6). As a result, the reuse type prediction in LASER is very accurate. Second, there is still a sizable gap between LASER and the optimal scheduler for few applications. This is due to two reasons. First, for benchmarks such as `Quicksort`, `SSSP-citation` and `SSSP-graph500`, LASER fails to accurately predict the reuse type as the kernel pairs have diverse reuse ratios in these benchmarks. For example, the sibling-sibling pairs in `Quicksort` are uniformly distributed among reuse ratio bins (see Figure 3.6). Second, for benchmarks such as `BFS-small`, `Radixsort` and `SA`, the reuse information collection overheads (i.e., the parallelism is compromised at the initial stages of execution where the child kernels are bound to the same CU to collect reuse information) outweigh the performance improvements that we get with the improved data locality.

Figure 3.13 plots the L1 hit rates with our limit study and LASER. As can be seen, the L1 hit rate increases by enabling hardware schedulers to be locality-aware. The L1 hit rate significantly improves for **kernel+WG** scheduling. This is because that WGs with high reuses are now mapped to the same CU to take advantage of the L1

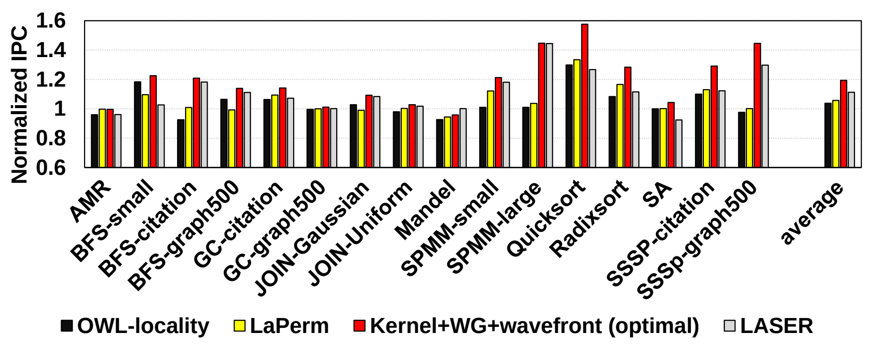


Figure 3.14: Performance comparison against prior schemes.

cache unlike the scenario in **kernel**, where the WGs are scheduled in round-robin. By using LASER, the L1 hit rate is within 3% of the optimal **kernel+WG+wavefront** scheduler.

We next compare LASER with two prior efforts that target data locality in GPUs: *OWL-locality* [71] and *LaPerm* [47]. OWL-locality implements a locality-aware wavefront scheduler to reduce cache contention and improve the latency hiding capability. It enhances the two-level wavefront scheduler with the knowledge of the data layout to WG mapping. LaPerm, on the other hand, binds child WGs with their direct parent WG, and schedules them on the same CU in a load balanced fashion. Figure 3.14 shows the comparison of OWL-locality, LaPerm, and LASER along with the optimal scheduling (**kernel+WG+wavefront**) policy. The results are normalized to baseline scheduling. On average, OWL-locality and LaPerm improve performance by 3.8% and 5.7%, respectively, over the baseline, whereas LASER provides 11.3% performance improvement. In summary, DP applications are generally complicated in data access pattern and have data reuses along different types of kernel/WG relationships. As a result, simply co-locating neighboring WGs (as in OWL-locality) or binding parent-child WGs (as in LaPerm) does not fully exploit the data reuse.

3.7 Conclusion Remarks

Dynamic parallelism is an effective approach for improving GPU performance and resource utilization when executing irregular applications. While there have been prior efforts focusing on resource management and overhead tolerance for dynamic parallelism, the data access patterns and data reuse remain unclear. In this chapter, we systematically characterize the data reuse and data locality opportunities that exist in dynamic parallel GPU applications. Based on our observations, we conduct a limit study to show the performance benefits of an “optimal” scheduler that realizes as much data reuse as possible. Furthermore, we propose a practical locality-aware scheduler, called LASER, which makes the GPU hardware schedulers locality-aware, and thus improves data reuse. Our experimental evaluations show that, on an average, 19.4% and 11.3% performance improvements can be achieved with an optimal scheduler and LASER, respectively.

Chapter 4 Improving Bank-Level Parallelism for Irregular Applications

Observing that large multithreaded applications with irregular data access patterns exhibit very low memory bank-level parallelism (BLP) during their execution, we propose a novel loop iteration scheduling strategy built upon the inspector-executor paradigm. A unique characteristic of this strategy is that it considers both bank-level parallelism (from an inter-core perspective) and bank reuse (from an intra-core perspective) in a unified framework. Its primary goal is to improve bank-level parallelism, and bank reuse is taken into account only if doing so does not hurt bank-level parallelism. Our experiments with this strategy using eight application programs on both a simulator and a real multicore system show an average BLP improvement of 46.8% and an average execution time reduction of 18.3%.

4.1 Introduction

To maximize the performance of multithreaded applications mapped to multicores/-manycores, one needs to consider *end-to-end data access performance*, not just the cache performance. In fact, trying to maximize LLC (Last-Level Cache) hit rates (which is the main goal of many compiler schemes) does not guarantee good, let alone being optimal, end-to-end data access performance [72–74]. This is because off-chip accesses can consume a lot of cycles, but more importantly, latencies they experience are not uniform, being dependent on several factors such as bank-level parallelism, row-buffer locality, memory scheduling policy, etc. Therefore, an end-to-end data access optimization strategy should consider cache performance as well as performance of the LLC misses. Unfortunately, while there are some recent hardware-based works targeting off-chip accesses [75–77], software works targeting off-chip accesses are still in their infancy.

One of the important factors that influence the performance of LLC misses is "bank-level parallelism" (BLP), which refers to the number of concurrently-served memory accesses by different memory banks in the system. Note that BLP is a measure of memory-level parallelism since in the ideal case one would want all the banks to be busy in serving memory requests (LLC misses). Note also that, in order to have high BLP, one needs (1) a large number of concurrent LLC misses and (2) a balanced distribution of these misses over the available memory banks. To achieve (1), LLC misses need to be clustered and, to achieve (2), misses should be reorganized either through code transformations or data layout transformations. While these tasks are not trivial and have not received much attention so far from the compiler and runtime system communities, they are even harder in the context of *irregular applications,* i.e., applications whose data access patterns cannot be completely analyzed at compile-time (e.g., index array-based calculations in scientific codes).

This chapter presents a novel strategy to optimize BLP of index array-based irregular programs. Our strategy, built upon the inspector/executor paradigm [78], reorganizes LLC misses at runtime to maximize BLP. To our knowledge, this is the first compiler work that targets improving BLP in irregular applications. The contributions of this work can be summarized as follows:

- It presents experimental evidence, using eight multithreaded irregular applications, showing that (1) the BLP of the original versions of these irregular applications are very poor in general, (2) simply maximizing memory-level parallelism (by clustering misses) does not bring significant improvements, and (3) maximizing bank-level parallelism on the other hand can bring significant performance benefits.

- Drawing insights from this motivational data, it next proposes a compiler/runtime based *loop iteration scheduling strategy* to maximize BLP. A unique characteristic of this strategy is that it considers both bank-level parallelism (from an inter-core perspective) and bank reuse (from an intra-core perspective) in a unified framework. Its primary goal is to improve bank-level parallelism, and bank reuse is taken into account only if doing so does not hurt bank-level parallelism.
- It gives experimental evidence showing the effectiveness of the proposed strategy. We evaluate the proposed approach in both simulator (to collect detailed off-chip statistics and compare it against hardware-based memory schedulers) and real multicore hardware. Our results indicate that the proposed strategy reduces execution time by 18.3% on average.

4.2 Background on DRAM and BLP

DRAM in modern systems is composed of various components like ranks, banks, and sub-arrays. Cores in a multicore access the data from off-chip DRAM through a component called Memory Controller (MC). Upon a last-level cache (LLC) miss, read/write requests are mapped to a specific MC based on the address mapping which we describe below. Requests to a DRAM are queued in a buffer at the MC, and are issued to the DRAM by MC. Figure 4.1 shows the basic organization of a DRAM and how it is connected to a multicore. Each MC manages a DRAM module also referred to as DIMM by issuing commands over address/data buses, also referred to as channel. Internally, each module is organized hierarchically as ranks, banks, and sub-arrays. We do not consider sub-arrays in our hierarchy as they are less common. Each DIMM is made up of multiple ranks. Each rank consists of multiple banks and all the banks in a rank share the timing circuitry. Each bank consists of a set of sense-amplifiers, referred to as *row-buffer,* where the memory row is loaded to before the data corresponding to the request is sent back over the channel. In an *open-row policy,* the row previously accessed is left open in the row-buffer. Consequently, if there is a request to the same row in the row-buffer, it need not be activated again, and hence incurs low latency resulting in a row-buffer hit. If there is a request to a different row, the current row in the row-buffer needs to be precharged and the new row has to be activated before the data is accessed and such a scenario is widely referred to as *row-buffer conflict*. Many works in the past proposed hardware-based schedulers that take advantage of the open-row policy. One such scheduler, FR-FCFS [79, 80], prioritizes accesses that target the current row in the row-buffer over other accesses.

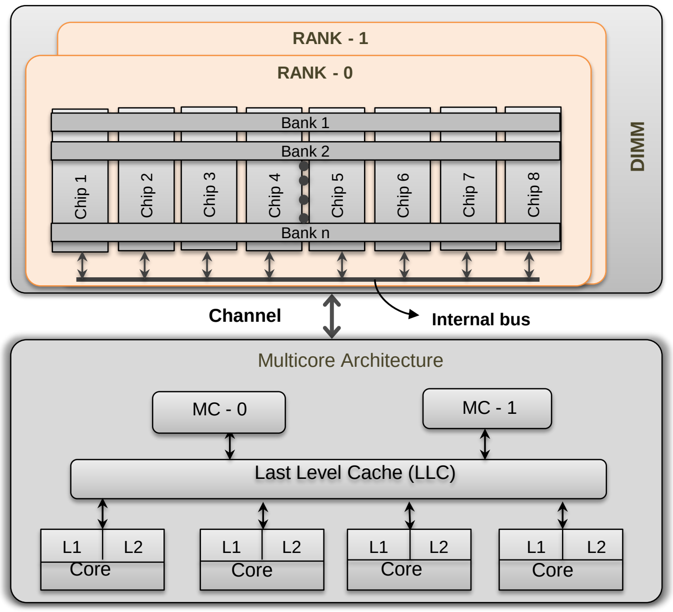


Figure 4.1: DRAM organization and DRAM-multicore interfacing.

Address mapping, also referred to as *interleaving*, governs how data are distributed across various components in the DRAM. This mapping (from physical addresses to memory banks) is decided statically (at hardware design time), and depending on the mapping scheme employed by the hardware, a request to a physical address can result in an access to a channel/rank and bank. Various interleavings are possible at each level in the memory hierarchy like caches, channels and banks. Two widely-employed interleavings are *cache line level* and *page level*. Address mapping plays an important role in determining the performance of the system as it effects both the locality and parallelism in the memory hierarchy. Figure 4.2 shows how a physical address is mapped to a channel/rank and bank based on page-level interleaving. The least significant 12-bits represent the page offset for a 4KB page. Assuming there are 4 MCs, the next 2 bits (bits 12 and 13) represent the channel id where this physical address is mapped. In a corresponding channel, assuming there are 4 ranks, the next 2 bits (bits 14 and 15) represent the rank where this physical address is mapped. Once the rank is determined, assuming there are 8 banks in a rank, the next 3 bits (bits 16, 17 and 18) are the *bank bits* and determine which bank this physical address is mapped to.

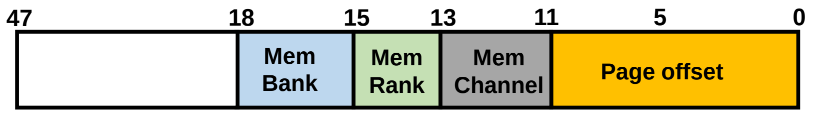


Figure 4.2: Page interleaved address mapping.

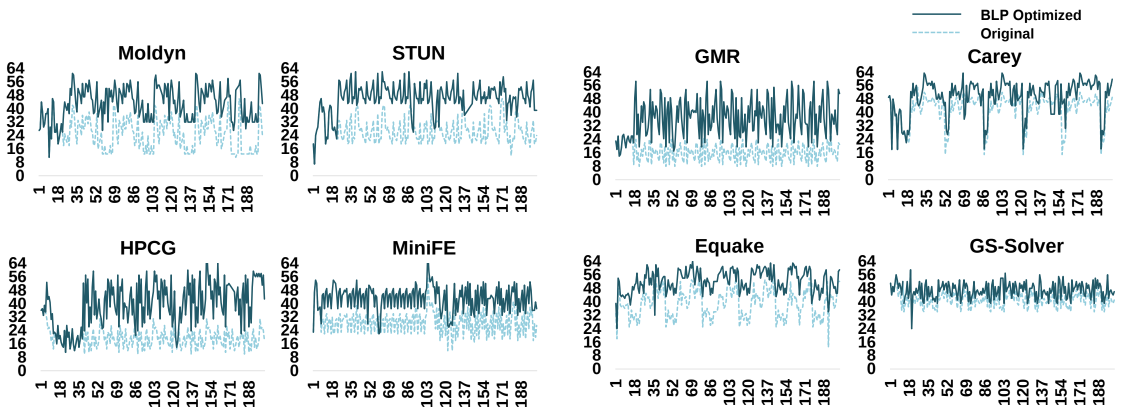
Ideally, consecutive requests to different pages in time domain should be mapped to different banks such that these independent requests are served in parallel. This is commonly referred to as **bank-level parallelism** (**BLP**). In this chapter, we define BLP as

the average number of requests being served in parallel by all the banks in the DRAM when at least one request is being served by any bank.

This definition for BLP is same as the one used in [77]. There exist various hardware-based schemes to improve BLP, and we compare our work to a few of them in this chapter.

4.3 Motivational Results

The curves marked as "Original" in Figure 4.3 give BLP values for a period of 2 billion cycles for our applications on a 12-core, 64-bank system. One can make two critical observations from these results. First, most of these applications do not perform well from a BLP angle. In fact, the average BLP values for applications HPCG and GMR are 19.7 and 16.6, respectively, as given in Figure 4.4. Second, as far as BLP is concerned, each of these applications exhibits a quite repetitive pattern. This is primarily because these index array-based irregular applications have an outermost "timing" loop that iterates either for a fixed number of iterations or until a convergence criterion is met. In fact, this repetitiveness (not just in terms of BLP,



(a) (b)

Figure 4.3: BLP values (y-axis) for our applications over a period of 2 billion cycles in a 12-core, 64-bank system.

but also in terms of access patterns and cache statistics) is the main reason why the inspector/execution paradigm (explained later) works well for irregular applications.

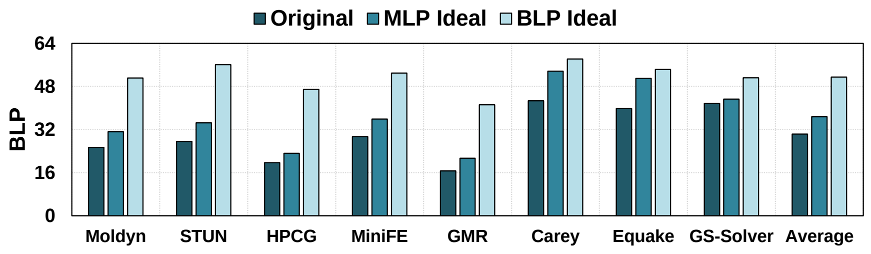


Figure 4.4: BLP results with different schemes.

To illustrate the influence of these low BLP values on performance, we present in Figure 4.5, the execution times collected using our simulator. The first bar for a benchmark in this bar-graph plots the execution time of the original application in seconds. We see that, without performing anything special regarding BLP, the execution times of our applications vary between 55.6 seconds and 124.4 seconds. At this point, one may suggest that optimizing memory level parallelism (MLP), that is, simply increasing the *burstiness* of off-chip memory requests can help us improve BLP and ultimately reduce the overall application execution times. To check the validity of this, we implemented in our simulator a strategy where the off-chip accesses originating from each core have been clustered as much as possible, subject to data dependences. Note that clustering memory requests does not necessarily mean delaying all of them. It is true that some memory requests are delayed due to clustering but also some other requests are moved to an earlier point. Actually, what this strategy (called MLP Ideal) implements in the simulator is the hardware-equivalent of the compiler technique proposed by Pai and Adve [81] that aims to increase memory-level parallelism. MLP Ideal incurs around the same number of LLC misses as the original case (within 1% in our experiments), but incurs them at different points in execution. Clearly, MLP Ideal can increase BLP, depending on the target banks of the misses clustered. The second bar for an application in Figures 4.4 and 4.5 give the resulting BLP values and execution times with MLP Ideal. On an average, maximizing MLP (instead of BLP) improves BLP by 21.2%, and reduces application execution time by 6.5%, both compared to the original case. In other words, while optimizing for MLP brings some BLP benefits, it is not very effective, and leaves a lot of performance on the table.

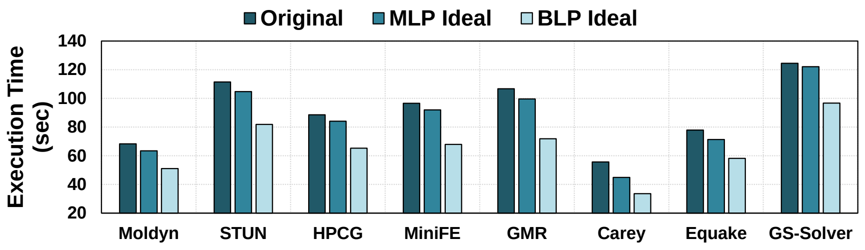


Figure 4.5: Execution time results with different schemes.

To show what the potential of an *ideal scheme* that maximizes BLP (as opposed to MLP) would be, we performed another set of experiments. It needs to be observed that, at any given period of time, there could be two reasons why an application can experience less than maximum BLP. First, there may not be enough number of off-chip memory references (e.g., if we have only 16 outstanding memory references in a period of execution, we can have a maximum BLP value of 16). Second, even if we have enough off-chip accesses, those accesses may not get distributed evenly across available memory banks. In our implementation of the ideal scheme, we ensured that, if there are sufficient number of off-chip accesses, they are always distributed across the banks evenly. Therefore, the only reason this ideal scheme could not achieve maximum BLP is the lack of sufficient number of memory accesses. The results with this ideal scheme (called BLP Ideal) are given as the last bar for each application, in

Figures 4.4 and 4.5. As compared to the original execution, this ideal scheme brings an average BLP improvement of 69.8%, resulting in an average execution time saving of 27.8%.

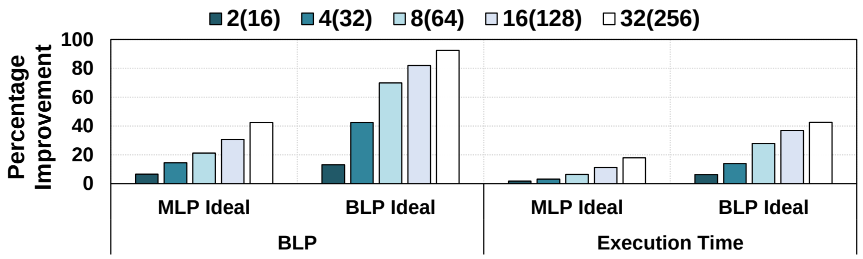


Figure 4.6: BLP and execution time improvements brought by BLP Ideal, with different bank counts on a system with 4 MCs and 8 ranks per channel.

Our last set of experiments in this section quantifies the potential of this ideal scheme in a system with a large number of banks. In Figure 4.6, a bar marked “a(b)" indicates that the system has “a" banks per rank, giving a total of b=2x4xa banks, (8(64) is the default configuration used so far). The y-axis represents the average value across all applications. We see from these results that the effectiveness of the BLP-optimal scheme increases as we increase the number of banks, which is the current trend in system design.

Overall, the results plotted in Figures 4.3, 4.4, 4.5, and 4.6 clearly show that simply maximizing MLP does not bring significant BLP improvements, and instead, maximizing BLP can bring significant performance benefits, especially with larger configurations. However, BLP Ideal sets an *upper bound* for potential execution time improvements and cannot be directly implemented. Thus, we propose a practical BLP optimization strategy that *approximates* BLP Ideal.

4.4 Technical Details

4.4.1 High Level View of Our Approach

The high-level view of our approach is illustrated in Figure 4.7 for a system with 4 cores and 4 banks. Each circle in this figure represents a *slab*, a set of loop iterations, which is the unit for scheduling computations in our framework. In Figure 4.7(a), the default execution order is shown, where accesses from different cores are clustered into the same bank (at a given period of time), resulting in a BLP of 1. For example, in the first period, requests from all cores access the first bank. Figure 4.7(b) depicts the execution order after our approach is applied. In this case, at any given time, all banks are accessed, giving a BLP of 4, which is much better than the default case.

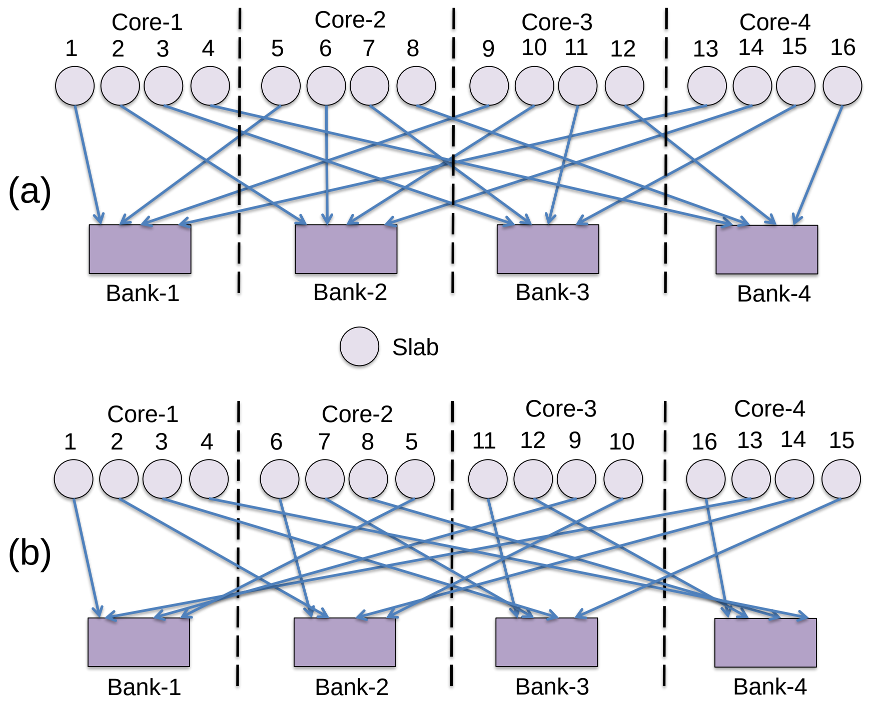


Figure 4.7: High level view of a system with 4 cores and 4 banks. Each core executes its slabs from left to right.

Our approach works at the granularity of a *parallel region*. For the purposes of this chapter, a parallel region represents a region that starts with an assignment to index arrays and ends with another assignment to them, as shown on the left side of Figure 4.8, for an example extracted from one of our application programs. The total set of iterations that will be executed by all cores in parallel region i is denoted using $C_i (1 \leq i \leq N)$, where N is the number of parallel regions. After the parallelization of C_i, the set of iterations assigned to core j is referred to as $L_{i,j}$, with $1 \leq j \leq P$, where P is the total number of cores. Each $L_{i,j}$ is divided into nearly equal sized slabs, S_{i,j,k_t}, where we execute S_{i,j,k_t} (from $L_{i,j}$) at scheduling slot (time) t.

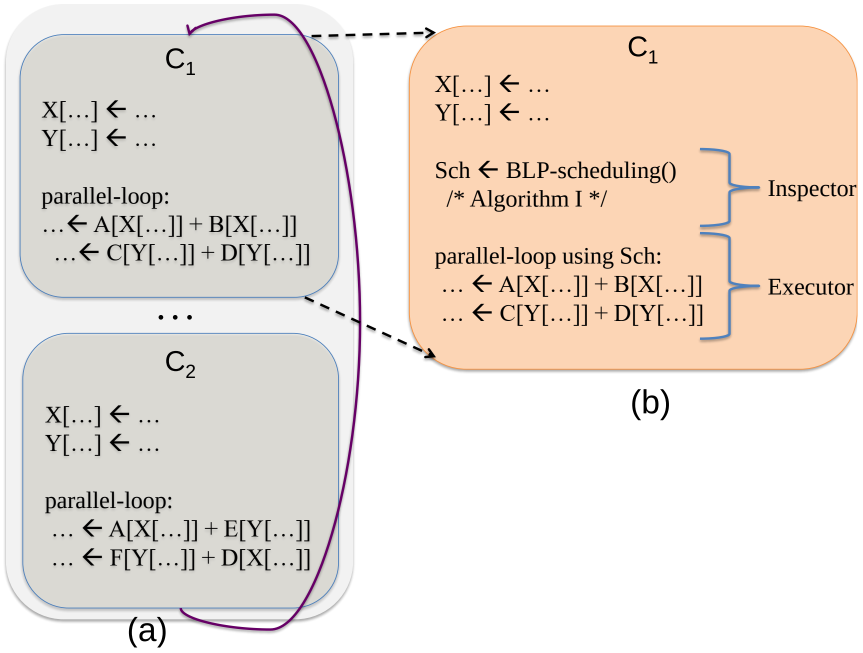


Figure 4.8: (a) A code fragment with two parallel regions and (b) Modified version of the first parallel region in (a) based on the inspector-executor paradigm.

4.4.2 Optimization Goal

The main goal behind our loop iteration scheduling algorithm is to optimize BLP. In mathematical terms, at each scheduling slot t, we need to select and schedule a slab for each core (i.e., S_{i,j,k_t} from $L_{i,j}$) such that *we cover as many memory banks as possible.* Clearly, to figure out the bank(s) accessed by a given slab, we first need to be able to predict the LLC misses, which is quite hard in the case of irregular applications. Therefore, in our default implementation, we conservatively assume that all data accesses in the parallel region will miss in the LLC, and schedule loop iterations based on this assumption. Later, we also explain how one can relax this assumption. To determine the bank to be accessed by an LLC miss, we also need support from the architecture and the OS. Operating systems have APIs that allocate a physical page for a given virtual address using page-coloring algorithm. There is also an API called page-create-va(.) in Solaris (and similar calls in other operating systems) that can accept hints from the user such that the physical pages allocated by the OS honor these hints. We modified this call to allocate physical addresses such that the kernel uses the same bank bits from the virtual address for the physical address. As a result, the bits specifying the bank (e.g., bits 16, 17 and 18 in Figure 4.2) are *not* changed during the virtual-to-physical address translation, and for a given slab, we can determine the set of bank(s) that hold the data that slab will access, and this allows the compiler to optimize for BLP using virtual addresses and expect the corresponding improvements when the hardware uses physical addresses. We observed during our experiments that the number of page faults did not increase after our optimization. That is, while our approach changes the virtual address-to-physical address mapping, doing so does not lead to any observable change in the virtual memory performance.

Each slab S_{i,j,k_t} can be associated with a bitmap, called *bank-map* Δ_{i,j,k_t}, of the form:

$$< B_1, B_2, \cdots, B_Q >,$$

where B_z $(1 \leq z \leq Q)$ is set to 1 if S_{i,j,k_t} accesses memory bank z, and 0 if it does not (Q is the total number of banks in the system). Consequently, Δ_{i,j,k_t} in a sense represents the "bank access pattern" of S_{i,j,k_t} in a compact fashion. Now, one can try to maximize the value of the following expression to optimize BLP at scheduling slot t:

$$\odot\{ \bigvee_{1 \leq j \leq P} \Delta_{i,j,k_t} \},$$

where $\vee$ denotes "bitwise OR" operation and $\odot$ is an operator that returns the number of 1s in a bit-map.

While the objective function given above can be used to maximize BLP, it does not consider row-buffer locality at all. One option to take into account row-buffer locality would be defining another type of bitmap (row-map) where each entry (position) captures whether we access a certain memory row or not. These vectors, which represent data access patterns at a memory row granularity, can then be used to develop a scheduler that can account for row-buffer locality. However, the sheer number of rows makes this approach infeasible to be implemented in practice as a part of dynamic scheme. Instead, we propose a strategy that works with the bank-maps defined earlier.

Our strategy is to maximize the value of the following target function, if doing so does not create a conflict with the BLP optimization goal discussed above:

$$\sum_{1 \le j \le P} \sum_{1 \le t \le T} \odot \{ \Delta_{i,j,k_{t-1}} \otimes \Delta_{i,j,k_t} \},$$

where $\otimes$ refers to the "bitwise Exclusive-NOR" operation. It is important to note that what this function tries to capture is to ensure that the bank-maps of two successively scheduled slabs from the same core ($\Delta_{i,j,k_{t-1}}$ and Δ_{i,j,k_t}) have the same bit values (0 or 1) in as many positions as possible. That is, this function is oriented towards achieving *bank reuse* across the successively-scheduled slabs from the same core. It is also to be noted that, while bank reuse does not necessarily guarantee memory row reuse, it increases the chances for the latter (in our experiments, we quantify the impact of our approach on row-buffer hit rate).

Overall, our approach tries to optimize BLP across the cores in a given scheduling step (horizontal dimension), while considering row-buffer locality, for each core, across successive scheduling slots (vertical dimension). The rationale behind this can be explained as follows. First, given sufficiently large slabs, careful selection of slabs from different cores (at the same scheduling slot) can be expected, in most cases, to cover all the memory banks in the system. If, for some reason, one wants to work with small slabs (each with fewer iterations) however, one needs to consider not just a single scheduling slot but multiple neighboring slots to make sure that all banks are covered. This generalized formulation will be given in the next subsection. On the other hand, the reason why we consider only intra-core bank reuse instead of inter-core bank reuse is the observation that sharing (at a memory row granularity) across cores is not as frequent as sharing within a core (especially in carefully-parallelized scientific codes where inter-core data sharing is minimized).

4.4.3 Generalization

There are two generalizations that we discuss. First, in optimizing BLP, we can consider multiple scheduling steps, and second, in considering row-buffer locality, we can consider inter-core bank reuse, in addition to intra-core bank reuse. The target function to maximize for BLP when considering q successive schedule slots instead of only the current slot t can be expressed as follows:

$$\odot\{\bigvee_{1\leq j\leq P;\ t-q\leq r\leq t} \Delta_{i,j,k_r}\}.$$

Clearly, q is a parameter that can be tuned to strike a balance between BLP and runtime overheads (due to working with small-sized slabs). The objective function that considers both intra-core and inter-core bank reuse (row-buffer locality) can be expressed as:

$$\sum_{1\leq j\leq P} \sum_{1\leq v\leq P;\ 1\leq t\leq T} \odot\{\Delta_{i,v,k_{t-1}} \otimes \Delta_{i,j,k_t}\}.$$

This function can be further enhanced to capture the bank reuse across multiple scheduling steps as well. Our experiments with this generalized scheme revealed that, considering 2 steps (instead of 1) in making scheduling decisions brought an additional 1.8% improvement (over the 1 step case) but increasing it to 3 or 4 steps did not bring any additional improvement. Consequently, in this chapter, we focus exclusively on the case where 1 scheduling step at a time is considered.

4.4.4 Algorithm and Example

To implement the objective function discussed in Section 4.4.2, our algorithm employs an iterative strategy. More specifically, to select the entries S_{i,j,k_t} in scheduling step t, our approach considers each core in turn, starting with the first one. For the first core, it selects a slab (as will be discussed shortly, bank reuse is taken into account for this). For the second core, it selects a slab such that this new slab covers as many banks as possible that have not been covered by the first slab. Similarly, for the third core, it picks up a slab that covers (if possible) the banks that have not been covered by the first two slabs, and so on. The row-buffer locality aspect on the other hand is taken into account as follows. Whenever we have multiple candidates (for a given core) to select from (i.e., candidates that cover exactly the same set of additional banks), we give priority to the one that maximizes bank reuse with the slab that has been scheduled on the same core in the *previous step*. In this way, row-buffer locality

Algorithm 4 BLP_scheduling

```
INPUT: Number of parallel regions (N); number of cores (P); number of slabs per core (M);
1: //Initialization
2: for $i$ from 1 to $N$ do
3:     for $j$ from 1 to $P$ do
4:         $L_{i,j} \leftarrow \{S_{i,j,1}, ..., S_{i,j,k}, ..., S_{i,j,M}\}$
5:     end for
6: end for
7: for $C_i$ from $C_1$ to $C_N$ do
8:     for $L_{i,j}$ from $L_{i,1}$ to $L_{i,P}$ do
9:         $t \leftarrow 0$
10:        while $L_{i,j} \not\equiv \varnothing$ do
11:            $schedule \leftarrow \varnothing$
12:            if $t \not\equiv 0$ then
13:                Search $S_{i,j,k}$ in $L_{i,j}$
14:                choose $S_{i,j,k}$ makes
15:                $\odot\{\Delta_{i,j,k_{t-1}} \otimes \Delta_{i,j,k_t}\}$ maximum
16:            else
17:                Random choose $S_{i,j,k}$ from $L_{i,j}$
18:            end if
19:            delete $S_{i,j,k}$ from $L_{i,j}$
20:            $schedule \leftarrow schedule \cup S_{i,j,k}$
21:            //Use iterative method to search for candidates
22:            for $l$ from $j+1$ to $P$ do
23:                $Candidate \leftarrow \varnothing$
24:                Search $S_{i,l,k}$ in $L_{i,l}$
25:                $Candidate \leftarrow$ all $S_{i,l,k}$ makes
26:                $\odot\{\bigvee_{1 \le r \le l} \Delta_{i,r,k_t}\}$ maximum
27:                if $t \not\equiv 0$ then
28:                    Search $S_{i,l,k}$ in Candidate
29:                    choose $S_{i,l,k}$ makes
30:                    $\odot\{\Delta_{i,l,k_{t-1}} \otimes \Delta_{i,l,k_t}\}$ maximum
31:                else
32:                    Random choose $S_{i,l,k}$ from Candidate
33:                end if
34:                delete $S_{i,l,k}$ from $L_{i,l}$
35:                $schedule \leftarrow schedule \cup S_{i,l,k}$
36:            end for
37:            schedule set is the set of slabs to schedule
38:            $t \leftarrow t+1$ //Time increased
39:        end while
40:    end for
41: end for
```

(actually, bank reuse) is considered only if doing so does not prevent us from reaching the best candidate from a BLP viewpoint.

Our approach is implemented using the *inspector-executor paradigm.* Specifically, for each parallel region, the compiler inserts the scheduler code right after the values of the index arrays are known (the index array assignments and the scheduler code together constitute the inspector). The main parallel loop that follows (known as executor) uses the schedule determined by the scheduler (see the right side of Figure 4.8). The formal algorithm for the scheduler is given as Algorithm I. The asymptotic complexity of this algorithm is $O(N * M * P^2)$, where N, P, M are respectively the total number of parallel regions, number of cores and number of slabs per core. This algorithm goes over cores one by one, and for each core, selects a slab from the remaining ones. In selecting a slab for a core j at schedule slot t, two rules are observed. First, the slab that contributes to most 1s when it is ORed with the slabs selected for cores 1 through $j-1$ in the same schedule slot (t) is selected. Second, if there are multiple such candidates, we give priority to the one that reuses most banks with the slab scheduler in the previous slot ($t-1$) on the same core. We also implemented a slightly modified version of this slab selection strategy, where cores (at a given schedule slot) are not visited in order, but based on the flexibility they have at that point. For example, if a core has only 1 potential slab that can enhance the current BLP, it is given priority over the others. This is because the others are less constrained and we may still find suitable slab candidates for them when they are visited. However, we observed in our experiments that, the difference between these two alternate implementations is less than 1%.

We want to emphasize that our approach is capable of handling a wide variety of indexed array applications. This includes applications where each index array is assigned only once in the program, as well as the applications where an index array is updated in multiple points in the program. Since our compiler analysis detects index arrays automatically, if no index array is used in the program, our optimization is simply not applied. We also believe that our implementation can be extended to work with a set of pointer codes where, once the pointer-based data structure is built, it is visited multiple times (e.g., many decision tree algorithms fall into this category). In such cases, we can collect the bank access patterns (or conservatively assume that every data access will be an LLC miss and go to main memory) after the data structure (e.g., tree) is built, and use this information in scheduling the chunks of computations that go over the data structure.

We focus on a small system with 4 cores and 4 memory banks. Each core is assumed to have been assigned 4 slabs. Figure 4.9(a) depicts the initial state of the cores at the first scheduling slot ($t = 1$). We randomly pick one slab (*slab* 1) from core 1. For core 2, to maximize the BLP, we have three choices (*slab* 1, *slab* 2 and *slab* 3). Our selection is still random at this point since no bank reuse is possible for the

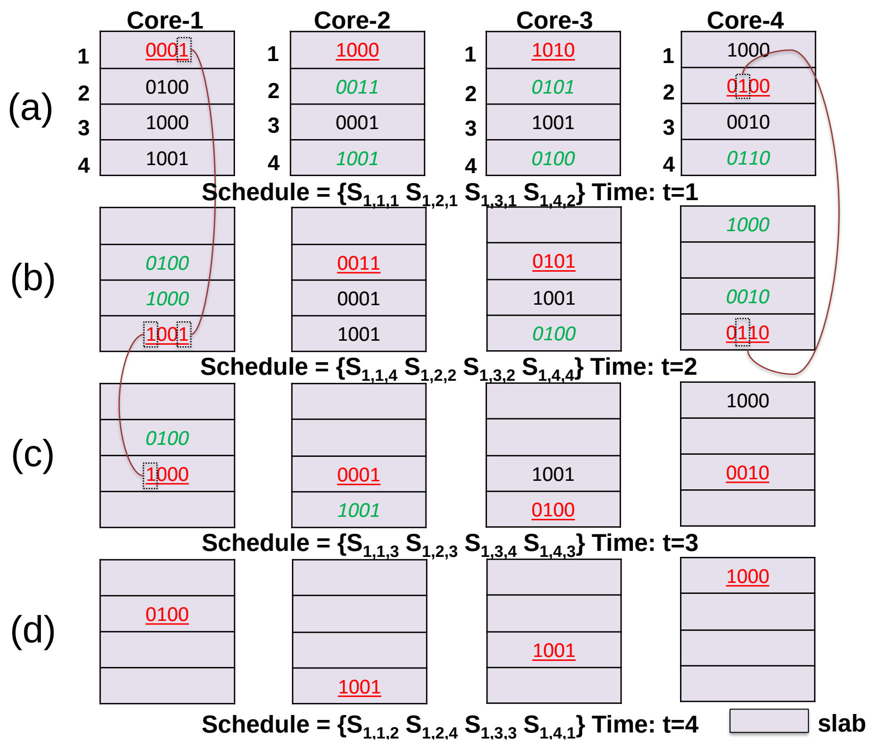

Core-1
Core-2
Core-3
Core-4
(a)
(b)
(c)
(d)
1
2
3
4
0001
0100
1000
1001
1000
0011
0001
1001
1010
0101
1001
0100
1000
0100
0010
0110
Schedule = {S1,1,1 S1,2,1 S1,3,1 S1,4,2} Time: t=1
Schedule = {S1,1,4 S1,2,2 S1,3,2 S1,4,4} Time: t=2
Schedule = {S1,1,3 S1,2,3 S1,3,4 S1,4,3} Time: t=3
Schedule = {S1,1,2 S1,2,4 S1,3,3 S1,4,1} Time: t=4
slab

Figure 4.9: An example of BLP optimized scheduling with 4 cores and 4 banks. Each core is assigned 4 slabs.

first schedule slot. At the end, we have the schedule set ($\{S_{1,1,1}, S_{1,2,1}, S_{1,3,1}, S_{1,4,2}\}$), giving the maximum BLP of 4 for this schedule slot. Figure 4.9(b) illustrates the case when time moves to the next scheduling slot ($t = 2$). Now, we have three choices for core 1 (*slab* 2, *slab* 3 and *slab* 4). Taking memory bank reuse into consideration, we pick *slab* 4 as this slab reuses the memory bank 4 from the previous schedule slot ($0001 \otimes 1001$). Similarly, for core 4 at this schedule slot ($t = 2$), we pick *slab* 4 to achieve memory bank reuse without hurting BLP. Therefore, the complete schedule for the second schedule slot is ($\{S_{1,1,4}, S_{1,2,2}, S_{1,3,2}, S_{1,4,4}\}$). Figure 4.9(c)and Figure 4.9(d) illustrate the results for the subsequent schedule slots ($t = 3$ *and* $t = 4$). Our algorithm ends when all the slabs are scheduled with maximum BLP while also exploiting bank reuse as much as possible.

4.4.5 Handling Regular Accesses

In determining the schedule (within the inspector code), the regular accesses are taken into account along with irregular accesses. Essentially, once the index arrays are assigned, we have all the information we need, and regular accesses along with irregular ones contribute to the determination of the schedule. Note also that the first job of the inspector is to determine the bank access pattern of a slab. As long as there is at least one regular or irregular reference from a slab to a bank, it is captured in the bank-map; repeated occurrences of the same irregular reference will not add anything more. However, if the same irregular reference is touched by two slabs, the bit of the corresponding bank is set in both the bank-maps. Also, if the same index array is updated multiple times, it is possible that the same reference in one place will point to a bank, and in another place to another bank. This is fully captured in our implementation.

4.4.6 Discussion

We now want to discuss a couple of important points. First, there is the question of why we consider BLP as the main optimization target and consider row-buffer locality only if doing so does not hurt BLP. The reason for this is two-fold. First, our framework operates with bank-maps and row-buffer locality optimization using them is a speculative one, i.e., its deemed benefits may or may not be realized at runtime. Second, memory schedulers employed by current architectures (e.g., FR-FCFS) compensate, to some extent, not-so-good row-buffer locality by prioritizing memory requests that hit in the row-buffers of DRAM banks over other requests, including older ones. So, even if the row-buffer locality is not optimized perfectly, the memory scheduler can still achieve some locality at runtime.

Second, while we assumed so far that the slabs in C_i can be executed in any order, our approach can be modified to work with the scenarios where we have inter-slab dependences (intra-slab dependences are taken naturally into account as the iterations in a slab are executed in their default order). To capture such dependences, we build a *dependence graph at the slab level*, where nodes correspond to slabs and an edge from one slab to another represents a data dependence between them. With this representation, our approach can be modified to consider (at each scheduling step) only the slabs that are "schedulable" and select, if dependences allow, one slab for each core to maximize BLP while considering bank-level reuse.

The third issue is regarding the validity of our conservative assumption which states that all data accesses will miss in the LLC. Clearly, this assumption does not hold in practice. By making this assumption however, capture a scenario where we put the maximum pressure on the main memory system. Clearly, depending on the actual misses at runtime, the relative success of our iteration scheduling strategy may vary. Also, in many irregular applications, a given slab is executed multiple times (once in each iteration of the outermost loop of the application). Consequently, it is possible to predict the banks that will be accessed by a given slab based on its previous executions. Current Intel processors like Xeon E5-E7 series [82] already provide uncore performance counters which can be read by a Unix performance tool like *perf* to understand the row-buffer hit rates in the DRAM banks. With all the physical address information already available in the MSHRs (Miss Status Handling Registers, which keep track of outstanding LLC misses), we assume that new ISA instructions which can capture bank accesses would be a logical extension to these performance counters. These ISA instructions could easily read the physical addresses in the MSHRs and identify the bank bits by performing a simple right shift bit-level operations and a modulo operation. With this hardware support, it would be possible to learn which bank(s) a given slab accessed in its previous executions, and accordingly, predict which banks it will access in its future executions.

Fourth, note that the scheduling problem we have is NP-hard, and our greedy algorithm may not work well in some cases because it makes a (local) slab selection decision at each step, and that decision binds the scope for future decisions (for other cores). To reach the optimal BLP, slabs for all cores should be selected considering *all cores at the same time* (instead of one-core-at-a-time). However, we decided against such a scheme because of two factors. First, this would increase the complexity of our algorithm. Second, we also formulated an ILP problem for the optimal slab selection and found that the additional execution time improvements it brought over our greedy scheme was only 2.5% on average (per parallel loop, after 7 hours of execution of the ILP solver).

Table 4.1: Benchmarks used in evaluation.

App	**Description**	**Input Size**	**MPKI**
Moldyn [84]	Generalized program for the evaluation of molecular dynamics models	336.2MB	81.3
STUN	Parallel sparse direct solver	1.25GB	66.6
HPCG [85]	High performance precondit- -ioned CG solver benchmark	210.8MB	87.4
MiniFE [86]	Finite element mini application	654.1MB	26.5
GMR	Generalized minimal residual based iterative sparse solver	308.6MB	91.2
Carey	Epidemic diffusion simulation on large social networks	1.10GB	12.5
Equake [87]	Earthquake simulation	487.7MB	14.1
GS-Solver [88]	Gauss-Seidel based iterative sparse solver	390.2MB	9.1

4.5 Experimental Evaluation

4.5.1 Setup and Applications

We implemented our scheme using *LLVM 3.5.0* [83] as a source-to-source translator. The original source code and the resulting optimized code are then compiled for simulator and actual hardware using different node compilers with the *highest optimization level* available, thereby activating all cache optimizations as well. We observed the following increases in compilation time (over the compilation time of the original applications): Moldyn:41%, STUN:26%, HPCG:36%, miniFE:39%, GMR:19%, Carey:14%, Equake:51%, and GS-Solver:67%. The longest compilation time we observed when using our approach was about 57 seconds. We evaluated our approach over a set of eight application programs described in Table 4.1. The third column shows the total input size (in MBs), and the last column gives the MPKI values of the original codes under our default simulation platform (described below). GMR, STUN and Carey are three codes written by our group.

We used both simulation-based evaluation (using GEM5 [89]) and commercial hardware-based evaluation in this work. The reason for the former is three-fold. First, we wanted to get detailed BLP statistics to measure the impact of our approach. Unfortunately, we are not aware of any way of measuring BLP in real systems. Second, we also wanted to conduct a sensitivity study where we change the values of critical system parameters/policies. Third, we wanted to compare our compiler-based scheme

Table 4.2: Major platform parameters.

Parameter	**Default value**
Cores	12 Xeon E5 (3.4GHz)
Cache Line	64 bytes (for all caches)
L1 Cache	32KB per core (private), 8-way, 4 cycle-latency
L2 Cache	256KB per core (private), 8-way, 12 cycle-latency
L3 Cache	10MB shared, 32-way, 32 cycle-latency
Data TLB	Two-level; L1: 64 entries, 4-way, L2: 512 entries, 4-way
Main Memory	4 DDR4-2933 MCs, 32GB capacity (tCL, tRCD, tRP) = (20cycles,20cycles,20cycles) 8 ranks/DIMM, 2 banks/rank, 2KB row size FR-FCFS (64 max requests/MC)

to existing hardware-based BLP optimization schemes and this comparison could only be done using a simulator. However, in addition to this simulation based study, we also collected execution time results on an Intel Ivy Bridge based multicore system. Note that, the default parameters used in our simulation closely follow those of the Intel architecture.

Table 4.2 gives the important features of the default system we modeled in our simulator. Note that the default memory scheduler in GEM5 is FR-FCFS, and later when comparing our approach to other memory schedulers, we changed this default scheduler. Our results are collected when both hardware and software prefetchers are ON (in both the simulation-based experiments and real-hardware executions). In the simulator, we implemented a state-of-the-art stride-prefetcher. In all the experiments presented below, the slab size is set to 1/50th of the iterations assigned to a core. Our sensitivity analysis with different slab sizes generated similar results, as long as the slab size chosen is large enough.

We define a metric called “coverage ratio" which captures the percentage of original executor iterations that use the new schedule after the optimization (in the ideal case, this ratio would be 100%). The coverage ratios for our benchmarks varied between 73% and 91%, averaging on 86%, indicating that our compiler was able to optimize a very large fraction of each application.

4.5.2 Simulation Results

The curves marked “BLP Optimized" in Figure 4.3 give the BLP results when our scheme is applied. Comparing these results to those of the original executions shown in the same figure, one can see that our scheme brings significant improvements

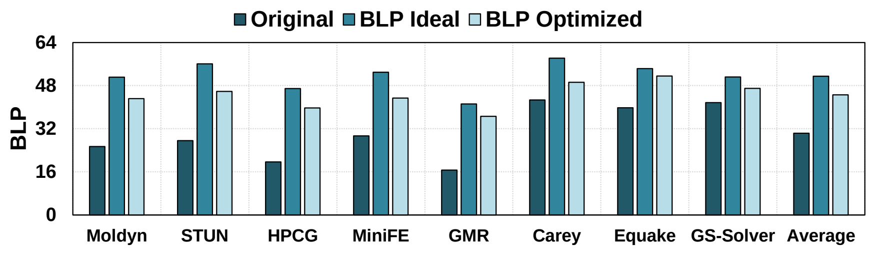


Figure 4.10: BLP results with different schemes.

in BLP. The third bar for each benchmark in Figure 4.10 gives the average BLP value with our scheme. The first two bars of the same graph reproduce results from Figure 4.4 for ease of comparison. When averaged over all application programs in our experimental suite, the proposed approach achieves an average BLP of 44.5, which is much better than the average BLP of the original benchmarks (30.3), and not too far from the BLP Ideal case (51.5).

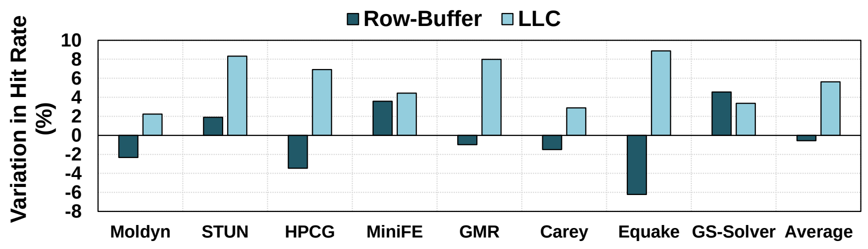


Figure 4.11: Variations in LLC and row-buffer hit rates as a result of our approach (BLP Optimized).

We now quantify the impact of our approach on execution times of our applications. Note that BLP is only one part of the big performance equation, and there are at least two important factors to consider here, in addition to BLP values, which may influence execution times: LLC behavior and row-buffer hit rates. On the positive side, recall that our approach tries to improve bank-level reuse, if doing so does not conflict with the BLP optimization goal. We can expect this to have a positive effect on both LLC performance and row-buffer hit rates. On the negative side, we have two issues to consider. First, since our approach changes the execution order of loop iterations, this can negatively affect the cache behavior. However, we do not expect this to be a major issue, as our approach works at a slab granularity and, since once a slab is scheduled all its iterations are executed in their original order, the impact on cache behavior will be quite limited. The second issue is due to the inherent conflict between BLP optimization and row-buffer locality optimization. Since our approach is primarily driven by the former, it may negatively affect the latter, though we expect the bank-reuse optimization to compensate for it. Further, the FR-FCFS memory scheduler used by the hardware is also expected to help with the negative impact of our approach on row-buffer locality. Figure 4.11 gives the variations on LLC hit rate and row-buffer hit rate when our approach is applied. We see from this plot that overall BLP Optimized improves row-buffer locality in 3 of our 8 applications, and distorts it in the remaining ones. One can also observe that the improvements in the LLC hit rates brought by our approach vary between 2.2% and 8.8%, averaging on 5.6%. It is also important to note from Table 4.1 that our applications have relatively high MPKI values, that is, they are memory intensive. With such high MPKI values, even after the optimization (and the reduction in LLC misses), there are still enough misses that allow BLP to play an important role. Further, comparing the last column of Table 4.1 and Figure 4.10, we see that our approach achieves better BLP improvements with applications that have higher MPKI values, as there are more memory accesses to schedule for different banks.

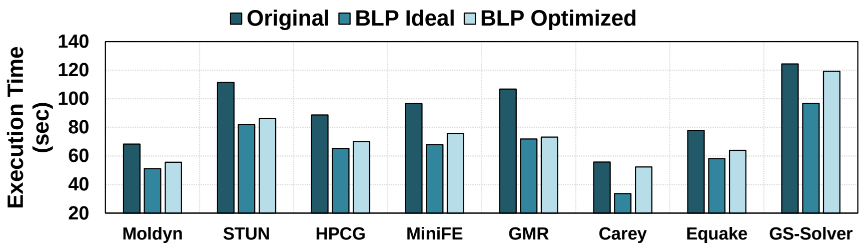


Figure 4.12: Execution time results (simulator).

The execution time results with BLP Optimized are presented in Figure 4.12, as the third bar for each benchmark. *These results capture the impact of our approach on BLP, row-buffer locality and LLC misses.* We see that our approach improves the execution times of all the applications. These improvements range between 4.1% (GS-Solver) and 22.6% (STUN), averaging on 18.3%. The relative improvements are lower in applications Carey and GS-Solver, which align well with the relatively lower BLP improvements observed in Figure 4.10. We want to emphasize that these execution time results also include all the runtime overheads incurred by our approach, which includes the execution of the code that determines the new scheduling as well as any impact on caches and on-chip network. Figure 4.13 zooms in this overhead for each benchmark, and quantifies it as a fraction of the total execution time. On an average, the contribution of the overheads amounts to 4% of the execution time. Actually, scheduling costs can be considered from two aspects. First, note that our optimization target is a parallel region, not loops. Therefore, if there is a large loop body, loop fission can be applied before our approach. Second, our approach actually uses a sliding window-based implementation. The reason is that searching all candidates to maximize BLP is not feasible in practice. Therefore, at any given time, our algorithm only considers the candidates in a window. Further, some portion of the overheads are also probably hidden during parallel execution. This is why in practice the overheads we observe are not very high.

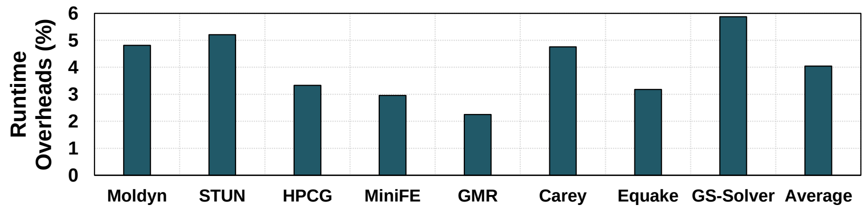


Figure 4.13: Contribution of the runtime overheads to the total execution time.

4.5.3 Results with Intel Ivy Bridge

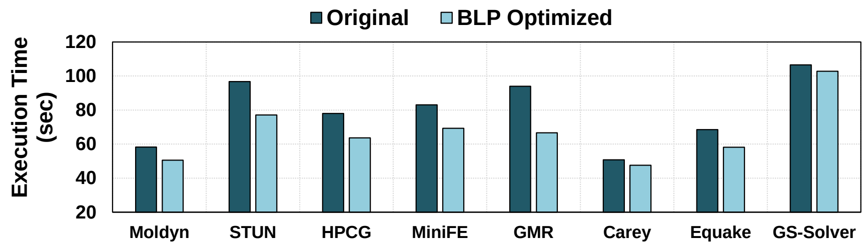
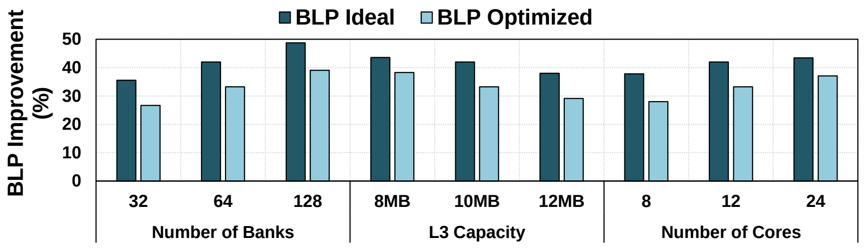
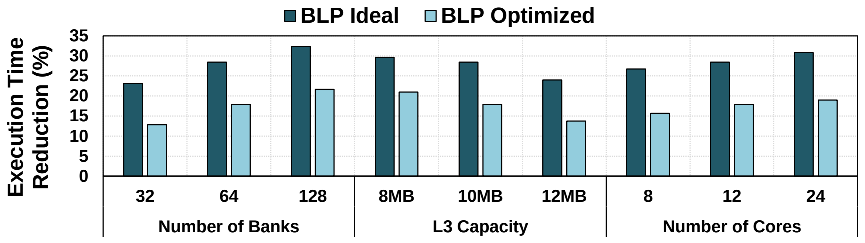


Figure 4.14: Execution time results (Ivy Bridge).

Next, we quantify the execution time improvement on our Ivy Bridge based multicore platform, which is equipped with 4 DDR3-2133 memory controllers (14 cycles for each of tCL, tRCD, and tRP). It is important to note that in the Ivy Bridge



(a)



(b)

Figure 4.15: Results from the sensitivity experiments. (a) BLP improvements, and (b) execution time reduction. In each experiment, all versions use the same hardware configuration, specified by the x-axis.

platform, there is no way to measure BLP directly. However, we are able to measure LLC misses, row-buffer hits and conflicts, and execution time. Due to space concerns, we present only execution time results but want to mention that the impact of our approach on LLC and row-buffer statistics were similar to the simulator case. In particular, compared to the original execution, our approach distorted row-buffer locality by 1% on average (generating, though, better row-buffer hit rates in three applications – STUN, miniFE, and GS-Solver), and improved LLC hit rates by 6.6% on average (improvements range from 1.8% to 9.2%). The execution time results given in Figure 4.14 indicate similar trends to those plotted in Figure 4.12. The average improvement brought by BLP Optimized is 15.7%. Clearly, there may be other factors in the real system that influence the execution times but cannot be captured by the simulator; however, our results indicate that applying our BLP optimization improves execution time significantly in the real system as well. As a point of comparison, when we modeled the same DDR3-2133 system in our simulator, we observed BLP improvements (over the original version) ranging between 18.8% and 56.3%, averaging on 30.7%. As indicated above however, there is no way to collect such BLP statistics from the real hardware.

4.5.4 Sensitivity Experiments

We now report, with the help of our simulator, the BLP and execution time results under different values of the system parameters. Each group of bars in Figure 4.15 represents the *average values* from a single sensitivity experiment, that is, the value of only one system parameter is varied, and the values of the remaining parameters are kept at their default values shown in Table 4.2. These results indicate that the effectiveness of our approach increases as we increase the number of banks, L3 (LLC) capacity, and number of cores. When we increase the number of banks (while keeping the number of memory requests the same), the number of idle banks in any given period of time increases, leading to a drop in the relative BLP (*not* in absolute BLP, as the absolute BLP increases with the increased bank count, but the relative BLP [the ratio between the observed BLP and maximum possible BLP] gets reduced). Consequently, there is more scope to optimize (more gap between the maximum BLP and observed BLP), and this in turn increases the potential impact of our BLP optimization.

One can also see that, although the effectiveness of our approach gets reduced with the increased L3 capacity, even with 12MB L3 cache it achieves 29% BLP improvement and 13.7% execution time improvement. On the other hand, it is not easy to predict the impact of increasing the number of cores on BLP. In the case of our benchmarks, we found that our optimization scheme generates better savings with the increasing core count, except for STUN and GMR. This is probably because increasing the number of cores creates more bank-level conflicts, which presents more opportunities to our approach. In addition to these three parameters, we also gauged the sensitivity of our approach to the row-buffer size, the number of memory controllers (keeping the total number of banks the same), slab size, and the size of the memory queue. Our results showed that the percentage performance improvements brought by our approach were less sensitive to these two parameters (within 2%). We also tested our irregular applications (in their default form) under both open-page and closed-page policies, and found that the former results in 4% better performance than the latter, due to primarily data locality/sharing across different threads (which may be more pronounced than in the case of commercial workloads). This observation plus the fact that we have more scope for optimization in the case of open-page policy motivated us to use the open-page policy as our baseline.

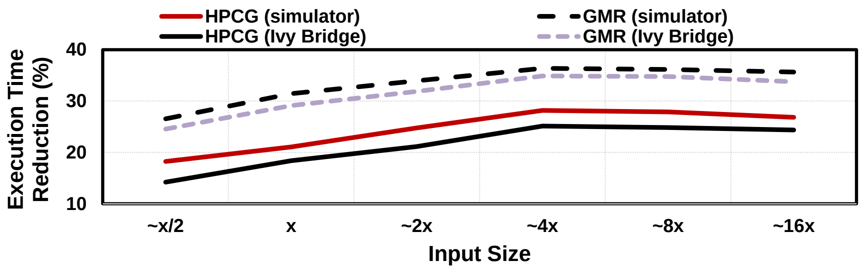


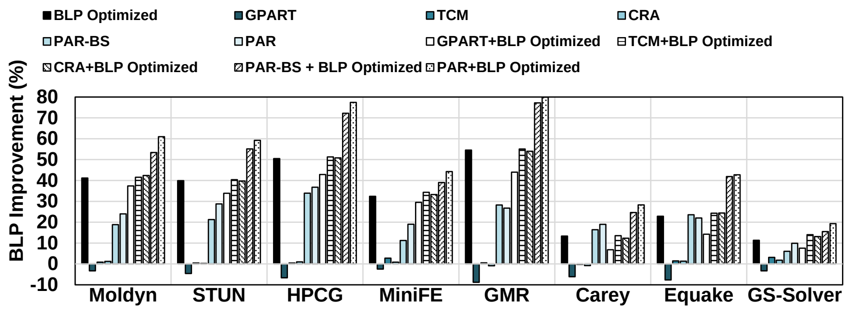
Figure 4.16: Execution time reduction for HPCG and GMR with different input sizes.

Our next experiment focuses on increasing the dataset sizes, and reports both simulation and Ivy Bridge results. In only two of our benchmarks (HPCG and GMR), we were able to change the input size safely. The results plotted in Figure 4.16[1] indicate that, as we increase the input size, the improvement brought by our approach increases but to a certain point. Beyond that point, the relative savings slightly reduce. This can be explained as follows. When the dataset size is increased, we incur more misses, which makes it even more important to exploit BLP. However, beyond a certain point, the outstanding misses start to fill all banks, and the original code starts to exhibit high levels of BLP. Since Figure 4.16 gives the relative improvements over the original version, we observe a reduction in savings.

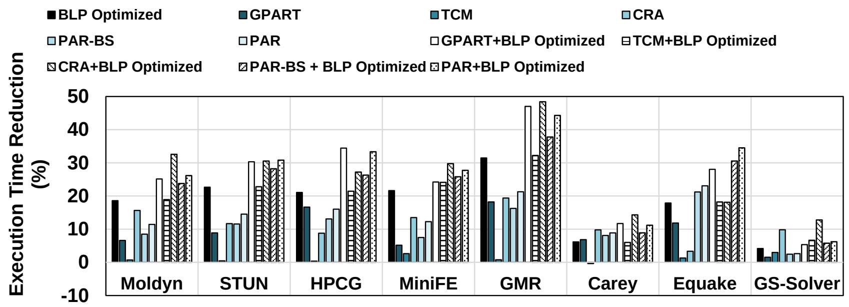
4.5.5 Comparison against Alternate Strategies

We are not aware of any compiler scheme that tries to optimize BLP for irregular applications. Pai and Adve [81] improve MLP in the context of single-core machines by clustering cache misses, and Ding et al [90] enhance BLP for multicore systems. However, both of these work with *regular loop structures with affine accesses* and do not have any runtime component, and consequently, they *cannot* handle irregular codes. In this subsection, we compare our approach, using GEM5, against five alternate schemes (one software based and four hardware based). The software

[1]On the x-axis, "x" corresponds to the default input size of the benchmark (210.8MB for HPCG and 308.6MB for GMR).



(a)



(b)

Figure 4.17: Comparison of our approach against alternate strategies: (a) BLP improvement and (b) Reduction in execution time.

scheme is GPART [91], which is a hierarchical graph clustering algorithm designed to re-organize data layout in irregular applications to improve cache performance (in our implementation, we adjusted the cluster sizes to maximize GPART's performance). The reason why we perform experiments with GPART is to gauge the impact a pure cache-locality oriented compiler scheme can have on BLP. The hardware schemes against which we compare are (1) PAR-BS [77], (2) TCM [92], (3) the critical region-aware parallel application memory scheduling scheme [93] (called CRA henceforth), and (4) the scheme described in [75] (called PAR henceforth). Note that CRA is a memory scheduling scheme designed exclusively for multithreaded workloads, whereas the remaining three hardware schemes are originally designed for multiprogrammed workloads. For each scheme we compare, we run that scheme alone as well as when it is coupled with our BLP Optimized.

The results presented in Figures 4.17(a) and (b) are normalized with respect to FR-FCFS and use the default system parameters given in Table 4.2. We see from

Figure 4.17(a) that GPART degrades BLP (with respect to the original case) in all eight applications. This is mainly because it does not do anything special for BLP and clustering data accesses (for better cache performance) tends to distort BLP, compared to the original case. Although not presented here due to space concerns, our experiments also showed that, while GPART improves the cache performance by 9.6% on average, it has little impact on row-buffer performance. These cache/row-buffer results combined with the BLP results generated, on average, 9.4% execution time improvement for GPART, as plotted in Figure 4.17(b).

Regarding the hardware schemes, we start by observing that TCM does not improve much over the FR-FCFS, primarily because TCM exploits the differences in memory intensities of different cores to improve system throughput, which makes a lot of sense in multiprogrammed workloads where one runs, for example, one application in each core. In a multithreaded application however, all threads normally have *similar* memory intensities. Consequently, except for two applications (miniFE and GS-Solver), TCM does not improve performance, and BLP Optimized generates an average BLP (resp. execution time) improvement of 32% (resp. 16.8%) over it. CRA prioritizes the threads holding locks over the others to reduce serialization; it improves over the default scheduler (as expected) in terms of the execution time, but it is orthogonal to our scheme (as it does not do anything specific for BLP). Consequently, our approach improves further over CRA; specifically, CRA and CRA + BLP Optimized generate average execution time savings of 11.5% and 26.7%, respectively, over the original execution.

PAR-BS tries to process the requests from a thread as a batch. Our approach generates better BLP results than PAR-BS in all programs except two (Carey and Equake). This is because, while PAR-BS can only take advantage of the potential BLP in memory queues, BLP Optimized can perform BLP-aware scheduling at a much larger scope (parallel region level). These trends translate to execution times, and we generate 6.9% improvement, on average, over PAR-BS, when all benchmarks are considered. When the two schemes (PAR-BS and BLP Optimized) are combined, we observe further improvements in application performance (23.4% on average over the original execution). Finally, PAR is a scheme originally designed for exploiting the potential MLP of prefetch requests. It has two components: the first one issues prefetch requests to MSHRs in a BLP-aware fashion and, the second one tries to preserve BLP exhibited by individual cores by removing interferences. We observe that, while PAR improves over FR-FCFS, our approach generates much better savings. This is mainly because PAR in a sense tries to improve BLP in a similar fashion to PAR-BS (in fact, as stated in [75], the benefits of the two schemes can partially overlap). The results in Figure 4.17(a) indicate that PAR and PAR+BLP Optimized generate average BLP (resp. execution time) improvements of 23.2% (resp. 13.7%) and 51.5% (resp. 26.7%), respectively.

To sum up, BLP Optimized outperforms, in most cases, all four hardware-based schemes tested and more importantly it can be used in conjunction with any BLP-aware scheduler (such as PAR-BS and PAR) to generate additional execution time savings. Also, it is orthogonal to schemes such as CRA (which improves aspects of execution other than BLP) and can be combined with them to obtain higher performance savings.

4.5.6 Scheduling with Dependences

Recall that, so far, if a code region has inter-slab dependences, we did not execute it in parallel. We also performed a set of experiments where such code regions are also executed in parallel, using the dependency graph discussed in Section 4.4.6. Although we do not present the detailed results due to lack of space, we want to say that 7 of our codes had at least one inter-slab dependence (miniFE did not have any), STUN having the largest number of such dependences (17 in total). Our approach generated an additional 3.3% (average) execution time improvement in this case, compared to the sequential execution of the code regions with inter-slab dependences (in STUN and Equake, the additional gains were 8.4% and 6.3%, respectively).

4.6 Conclusion Remarks

This chapter proposes and evaluates a novel loop iteration scheduling strategy to improve memory bank-level parallelism (BLP) of irregular application programs. The proposed scheduling strategy uses bank-maps to capture bank access patterns and reorganizes groups of iterations, called slabs, across cores to increase the number of concurrently-accessed banks. It also considers bank reuse, in an attempt to improve row-buffer locality, if it does not conflict with the BLP optimization. Our detailed evaluations of this scheduling strategy indicate significant improvements in terms of both BLP (46.8% on average) and execution times (18.3% on average).

Chapter 5
Co-optimizing Memory-Level Parallelism and Cache-Level Parallelism

Minimizing cache misses has been the traditional goal in optimizing cache performance using compiler based techniques. However, continuously increasing dataset sizes combined with large numbers of cache banks and memory banks connected using on-chip networks in emerging manycores/accelerators makes cache hit–miss latency optimization as important as cache miss rate minimization. In this chapter, we propose compiler support that optimizes both the latencies of last-level cache (LLC) hits and the latencies of LLC misses. Our approach tries to achieve this goal by improving the parallelism exhibited by LLC hits and LLC misses. More specifically, it tries to maximize both cache-level parallelism (CLP) and memory-level parallelism (MLP). This chapter presents different incarnations of our approach, and evaluates them using a set of 12 multithreaded applications. Our results indicate that (i) optimizing MLP first and CLP later brings, on average, 11.31% performance improvement over an approach that already minimizes the number of LLC misses, and (ii) optimizing CLP first and MLP later brings 9.43% performance improvement. In comparison, balancing MLP and CLP brings 17.32% performance improvement on average.

5.1 Introduction

Compiler researchers investigated a variety of optimizations to improve cache performance [94–102]. Most of these optimizations are geared towards minimizing the total number of cache misses, the rationale being that, lower the misses, higher the application performance. While this is certainly true and many commercial compilers already employ a large suite of optimizations that target cache miss minimizations (e.g., loop permutation, iteration space tiling, loop fusion), the impact of these techniques is becoming increasingly limited as (i) emerging applications are processing enormous amounts of data, (ii) the increases in cache capacities are lagging far behind the increases in application data volume [12, 13], and (iii) as a result, caches are becoming unable to maintain application working sets even after aggressive cache miss minimization.

As a result, a complementary approach would be embracing cache misses and trying to reduce their latencies (in addition to their counts). Recent works [92,103–114] have shown that significant amount of the overall data access latency is spent on cache miss related traffic (either as network latency to reach the last-level cache (LLC) banks/memory controllers (MCs), or as memory access itself). In other words, cache misses contribute to a large fraction of the overall data access latency. Therefore, an optimization approach that targets cache miss latencies can potentially bring significant reductions in application execution times.

Meanwhile, to enable application scalability, modern manycores are employing scalable interconnects (e.g., mesh-based network-on-chip (NoC)), instead of conventional buses. However, such NoC-based manycores lead to *non-uniform* latencies for both LLC hits and LLC misses. Typically, a data access missing in its local private cache (e.g., L1) is routed to remote LLC bank, and if it is a miss in LLC, it will be further routed to corresponding MC for an off-chip access. In this flow, multiple simultaneous accesses to a given LLC bank further increase the access latency, even when these accesses hit in the LLC bank. This is because that (i) routing all the requests to the same node can cause network contention, and (ii) multiple requests to the same LLC/memory bank can lead to cache contention.

Therefore, it is very important for an optimizing compiler that aims to maximize the performance of data access in an NoC-based manycore to minimize the latencies of both LLC hits and LLC misses (in addition to reducing the number of LLC misses, which is a traditional optimization goal). One way of reducing the latencies of both LLC hits and LLC misses is to improve their parallelism, that is, the number of LLC banks and memory banks that are concurrently accessed in a given period of time should be maximized.

In this chapter, we define cache-level parallelism (CLP) as the number of the LLC banks serving L1 misses when at least one LLC bank is serving an LLC access. Similarly, we define memory-level parallelism (MLP) as the number of memory banks serving LLC misses in parallel when at least one request is being served by a memory bank[1]. We then propose a compiler framework for reducing the latencies of both LLC hits and LLC misses, by increasing their parallelism. At a high level, this is achieved by maximizing MLP and CLP in a given period of time (i.e., execution epoch). Our contributions can be summarized as follows:

- We propose an optimization strategy that optimizes MLP for LLC misses and CLP for LLC hits *together*. Our approach employs code restructuring and computation scheduling, with the goal of reducing the latency experienced by data accesses. More specifically, for LLC hits, we want to maximize the number of cache banks concurrently accessed; and, similarly, for LLC misses, we want to maximize the number of memory banks concurrently accessed within a given period of time (in addition to the number of cache banks, as all memory accesses visit LLC banks before memory banks).
- We explain how our strategy can be used to strike a balance between MLP and CLP. Specifically, by considering the total number of accesses to each of the cache and memory banks, our compiler automatically determines the proper "trade-off" between MLP and CLP for each loop nest, that leads to the best application performance.
- We evaluate our approach using a set of 12 multithreaded applications on both a detailed manycore simulator and a commercial manycore system (Intel Knight's Landing [8]). The experimental data collected from the simulator indicate that (i) optimizing MLP first and CLP later can bring, on average, 11.31% performance

[1]We use the terms "memory-level parallelism" (MLP) and "bank-level parallelism" (BLP) interchangeably.

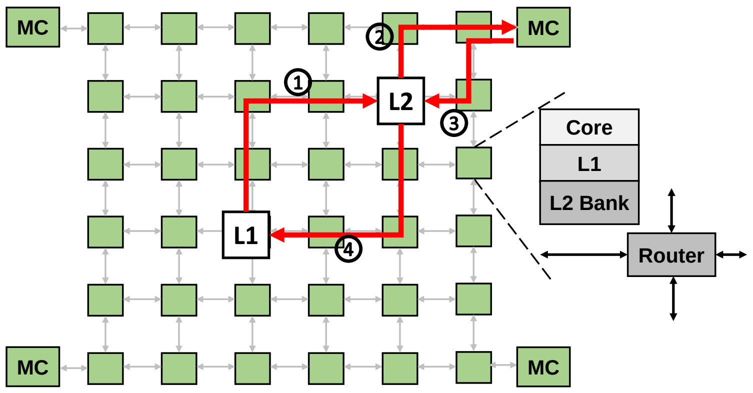


Figure 5.1: Network-on-chip (NoC) based manycore architecture template and the flow of a representative data access.

improvement over an approach that already minimizes the number of LLC misses, and (ii) optimizing CLP first and MLP later can bring 9.43% improvement. Finally, balancing MLP and CLP can bring 17.32% improvement. The corresponding improvement from our approach that balances CLP and MLP on Intel manycore is 26.15%, on average.

- Using both the simulator and the commercial manycore architecture, we present a detailed comparison of our approach against two previously-published compiler optimizations [81, 115] as well as a hardware-based memory parallelism optimization [77]. The experimental results collected clearly show that our proposed approach performs better than these alternative approaches. Specifically, it performs 12.87%, 8.31% and 6.21% better, respectively, compared to [81], [115] and [77], when using the simulator. On the commercial manycore system, our approach outperforms [81] and [115] by 20.13% and 13.27%, respectively.

To our knowledge, this is the first work that presents a compiler scheme designed to *co-optimize* MLP and CLP. Further, our approach, which primarily targets “hit and miss latencies” is *complementary* to conventional data locality optimizations (that target minimizing the “number of cache misses”) as well as techniques designed to increase compute level parallelism (e.g., loop parallelism).

5.2 Manycore Architecture

In this chapter, we target emerging Network-on-Chip (NoC) based manycore/accelerator architectures. Figure 5.1 depicts a manycore architecture template. Each node in this architecture contains a core, a private L1 cache, and a shared LLC bank. The LLC is divided in to *cache banks* and shared across all cores. The LLC in our baseline architecture refers to L2 cache, though our approach can work with cache hierarchies of any depth. We assume a static-NUCA based [116] LLC management, where the LLC is partitioned into (cache) banks and banks are distributed across nodes. Each cache line is statically mapped to a particular LLC bank based on its address. Figure 5.1 also illustrates a typical request/data flow involved in a data access. Specifically, an access that misses the private L1 cache is directed to an LLC (L2) bank (①). If it hits in the LLC bank, the requested data is sent back to the requesting node (④). However, if it misses there, an LLC miss occurs, and the request is forwarded to the corresponding MC (②). The request to a DRAM is queued in a buffer at the MC, and is issued to the DRAM by MC.

The core in each node can simultaneously issue data reference requests and those requests are received and served by LLC banks (for hits), or memory banks (for misses). Our optimization focuses on cache-level parallelism (CLP) and memory-level parallelism (MLP) in an epoch of n cycles. Typically, n can be set to a value considering the size of the ROB (reorder buffer) in the target architecture [117]. Note that, CLP captures the number of LLC banks serving L1 misses when there is at least one bank serving an L1 miss. Clearly, a higher CLP value indicates a better utilization of the LLC in the system. Similar to the CLP case, a higher MLP means better utilization of hardware resources memory banks. Each memory bank has a sense-amplifier called *row-buffer*, which is used to hold the memory row loaded. Subsequent accesses to the same row experience short latency, and are referred as row-buffer hits in an open-row policy.

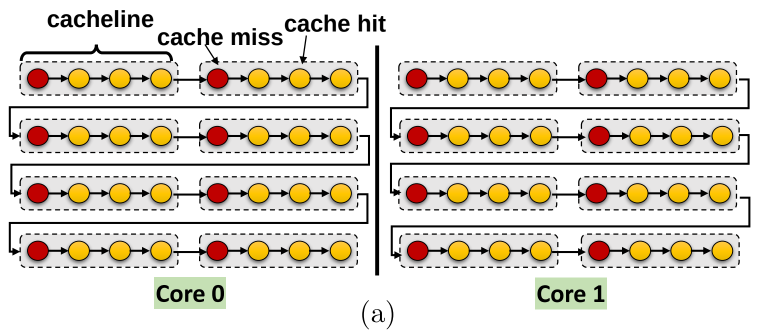
In addition to a manycore simulator, we also use Intel Knight's Landing (KNL) [8]. KNL consists of 36 nodes (referred to as tiles in Intel terminology) connected through mesh on-chip network. Each tile consists of two cores where each core features two 512-bit AVX vector units (VUs). There is a 1 MB "tile-private" L2 cache shared by two cores within a tile and cache coherence is maintained among L2 caches across different tiles. KNL has a 16GB of multi-channel dynamic random access memory (MCDRAM), which is divided into 8 channels and attached to the 8 MCDRAM MCs spread across 4 corners of the mesh on-chip network. This MCDRAM, which is separate from the DDR4 memory, can be configured into one of three different modes: (i) cache mode, where MCDRAM simply acts entirely as a conventional LLC (L3); (ii) flat mode, where MCDRAM acts entirely as an addressable memory; and (iii) hybrid mode, where 25% (or 50%) of the MCDRAM capacity is configured as LLC and the rest is configured as an addressable memory. KNL also has three different cluster modes. The base mode is referred to as the "all-to-all" mode, where the addresses are spread over the caches and MCs uniformly. On the other hand, in the "quadrant" mode, the entire mesh is divided into 4 virtual regions, and a memory access travels within the same region. Finally, in the "SNC-4" mode (also known as the sub-NUMA mode), the mesh is split into 4 non-uniform memory access (NUMA) clusters. In this case, all accesses (both cache accesses and memory accesses) travel within each NUMA cluster.

5.3 Motivation

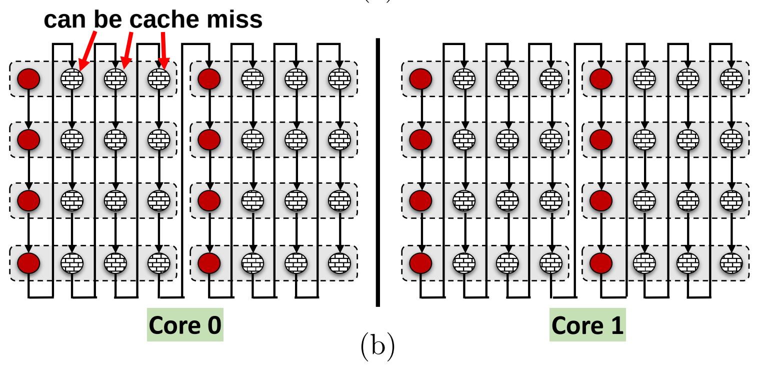
Let us consider the data access[2] pattern shown in Figure 5.2a on a two-dimensional array. We assume that the array is stored in memory in a row-major fashion (as in C language). There are two cores in the system, and each core accesses a 4×8 portion of the array (for illustrative purposes). Let us assume that this access pattern repeats itself in a (timing) loop (that is, after the last element of the array is accessed, the first element is accessed again, until a convergence criterion – captured by the timing loop condition – is met). The figure also highlights the cache lines (blocks) using gray boxes, each holding 4 array elements.

The access pattern in Figure 5.2a is very good from a data locality viewpoint, as the only misses incurred are for the accesses to the first element of each cache line (i.e., cold misses). The problem with this hit/miss pattern is that it does not exploit MLP (memory-level parallelism) well. Assuming, for example, there are 4 MCs in

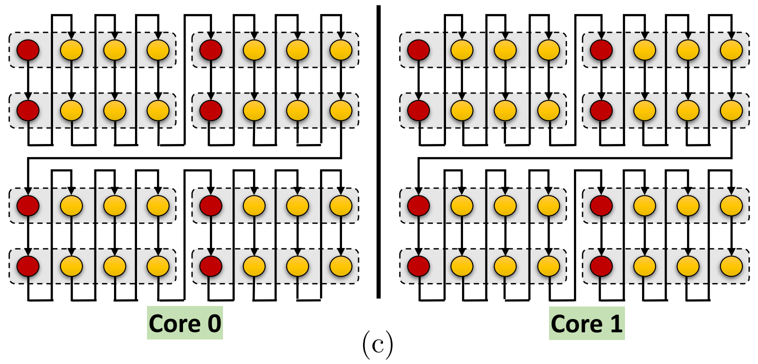
[2]We use the terms "data access" and "array reference" interchangeably.



(a)



(b)



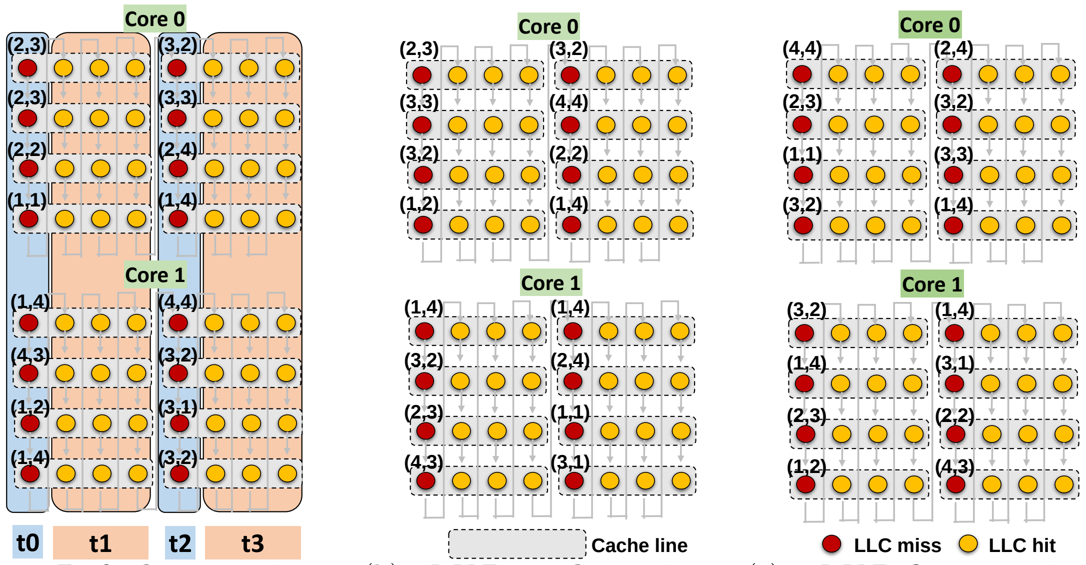
(c)

Figure 5.2: An example of data access restructuring to cluster cache misses. The ovals represents the array elements. The shaded rectangle box represents the cache line. The arrow denotes the reference order. (a) Default reference pattern captured by solid arrows. (b) Reference pattern after applying loop permutation. (c) Reference pattern after applying loop tiling on (b).

the system each controlling 4 banks, from a single core perspective, at a period of accessing consecutive 4 array elements, only one memory bank is accessed (by the access corresponding to the miss), leading to a total of 2 bank accesses at most when considering both cores. This is clearly much lower than the maximum possible of 16 memory banks.

One way of improving MLP is to *cluster* cache misses. In Figure 5.2b, each core traverses its portion of the array in a *column-major fashion*, instead of the *row-major fashion* (shown in Figure 5.2a). This new traversal order clusters the cache misses as they are now accessed in bursts. Therefore, at the initial period of 4 data accesses, each core accesses 4 banks (assuming each request/miss goes to a different bank), resulting in 8 banks being accessed when two cores are considered. However, now, one can expect additional cache misses since the new access pattern (which is column-wise) does not align with the underlying row-major layout of the array (in fact, in the worst case, after this transformations, all hits in Figure 5.2a can get converted into misses in Figure 5.2b). In other words, by going from Figure 5.2a to Figure 5.2b, we improve MLP but distort cache locality. However, this can be fixed, as suggested by [81], by tiling/strip-mining the innermost loop. The new post-tiling access pattern is illustrated in Figure 5.2c. Thus, after the back-to-back optimizations of loop permutation and tiling, we have the cache misses *clustered* and, at the same time, we maintain the original cache locality (number of cache hits).

Similar to optimizing MLP for LLC misses (red ovals in Figure 5.2), we can also optimize CLP for LLC hits (yellow ovals in Figure 5.2). There are two benefits of considering CLP together with MLP. First, a higher CLP indicates better utilization of LLC banks, and consequently reduces the LLC hits latency by overlapping (in time) different LLC accesses. And, second, a higher CLP means that requests are spread across different nodes in the network more uniformly. This potentially reduces the previously mentioned non-uniformity of cache hit latencies, and also better balances the utilizations of network links and routers (i.e., reduces network contentions as well). In this chapter, we explore three optimizations: *MLP-first*, *CLP-first*, and *Balanced*. In MLP-first, we first optimize MLP, then CLP is optimized without distorting optimized MLP. Alternately, in CLP-first, we optimize CLP as the primary target and MLP as the secondary. Finally, in Balanced, we try to strike a balance between MLP and CLP. The following discussion focuses mainly on MLP-first. The CLP-first is quite similar and therefore we omit its detailed discussion. We discuss Balanced in Section 5.5.6.



(a) **Default:** Intra-core: MLP$_{core0}$={2,3}, MLP$_{core1}$={2,2}, CLP$_{core0}$={3,3}, CLP$_{core1}$={3,3}. Inter-core: MLP$_{t0}$={3}, MLP$_{t2}$={4}, CLP$_{t1}$={4}, CLP$_{t3}$={4}.

(b) **MLP only:** Intra-core: MLP$_{core0}$={3,4}, MLP$_{core1}$={4,3}, CLP$_{core0}$={2,2}, CLP$_{core1}$={3,2}.Inter-core: MLP$_{t0}$={4}, MLP$_{t2}$={4}, CLP$_{t1}$={3}, CLP$_{t3}$={3}.

(c) **MLP-first:** Intra-core: MLP$_{core0}$={4,3}, MLP$_{core1}$={3,4}, CLP$_{core0}$={4,3}, CLP$_{core1}$={3,4}. Inter-core: MLP$_{t0}$={4}, MLP$_{t2}$={4}, CLP$_{t1}$={4}, CLP$_{t3}$={4}.

Figure 5.3: Clustering array references to improve both intra-core MLP/CLP and inter-core MLP/CLP. The read oval in each cache line represents LLC miss, whereas the subsequent yellow ovals represent LLC hits. Each cache line is associated with a memory bank and an LLC bank. The pair (memory bank, LLC bank) above a cache line indicates the corresponding memory bank ID and LLC bank ID.

5.4 High-Level Overview of Our Approach

It is to be noted that, clustering misses may not necessarily guarantee high MLP. This is because it is possible that the clustered misses still access only few memory banks. Motivated by this observation, we propose a loop iteration scheduling strategy where the clustered misses provide the maximum values of MLP from both the *inter-core* and the *intra-core* perspectives. Specifically, LLC misses issued within tiles across different cores (inter-core), and LLC misses clustered within a tile of a given core (intra-core) access as many different memory banks as possible.

Consider the example in Figure 5.3, which shows the array access order after the loop permutation and strip-mining (Section 5.3). For simplicity, let us assume that there are 4 memory banks and 4 LLC banks in the system. For a given cache line (denoted as gray box), the corresponding memory bank ID and LLC bank ID are labeled as a pair of number on top of the first data element in a given cache line.

For explanation purposes, in Figure 5.3a, we divide the scheduling process into 4 phases (labeled as $t0$ to $t3$).

In this example, among the total 16 cache lines, there are 5 in memory bank-1, 4 in memory bank-2, 5 in memory bank-3, and 2 in memory bank-4. In MLP-first, our focus is on the distinct memory banks accessed by LLC misses (red ovals). Figure 5.3a depicts the default array accesses order *without* our optimization. In phase $t0$, LLC misses from $core_0$ access two different memory banks (bank-2 and bank-1). Therefore, the intra-core MLP for $core_0$ is 2. In phase $t2$, the misses from $core_0$ access bank-3, bank-2, and bank-1, resulting in an intra-core MLP value of 3. Hence, we can denote the intra-core MLP of $core_0$ as $\mathrm{MLP}_{core0} = \{2,3\}$. Similarly, we have intra-core MLP of $core_1$ as $\mathrm{MLP}_{core1} = \{2,2\}$. Compared to intra-core MLP, inter-core MLP considers concurrent data accesses from different cores within the same execution phase. Specifically, in phase $t0$, the misses from $core_0$ and $core_1$ access bank-1, bank-2, and bank-4. As a result, we have $\mathrm{MLP}_{t0} = \{3\}$. Similarly, we calculate $\mathrm{MLP}_{t2} = \{4\}$ at phase $t2$.

We now try to optimize both intra-core and inter-core MLP. Figure 5.3b shows the new array reference order after iteration scheduling. Note that the new array referencing order is generated because of reordering the execution order of loop iterations, *not* data layout transformation. Using the same MLP calculation discussed above, the intra-core MLPs of the new array reference order are $\mathrm{MLP}_{core0}=\{3,4\}$ and $\mathrm{MLP}_{core1}=\{4,3\}$. Similarly, the inter-core MLPs are $\mathrm{MLP}_{t0}=\{4\}$ and $\mathrm{MLP}_{t2}=\{4\}$. As one can observe, the new loop iteration order in Figure 5.3b improves *both* intra-core MLP and inter-core MLP compared to the execution in Figure 5.3a.

An interesting observation is that two different iteration schedules could have exactly the *same* MLP values. For example, Figure 5.3c has the same intra-core MLP and inter-core MLP compared to Figure 5.3b, with a different loop iteration execution order. This potential allows us to optimize CLP for cache hits (denoted as yellow ovals in Figure 5.3), *without compromising MLP*. In Figure 5.3a, the second number in the pair denotes the LLC bank ID. In total, there are 2 accesses to LLC bank-1, 5 accesses to LLC bank-2, 4 accesses to LLC bank-3, and 5 accesses to LLC bank-4. All the three hits (yellow ovals) in a cache line access the same LLC bank. In the default loop execution order (Figure 5.3a), cache hits from $core_0$ access LLC bank-3, LLC bank-2, and LLC bank-1, giving an intra-core CLP of 3 in phase $t1$. Similarly, cache hits access LLC bank-2, LLC bank-3, and LLC bank-4 in phase $t3$. As a result, the intra-core CLP for $core_0$ is CLP$_{core0}$=$\{3,3\}$. We apply the same calculation for $core_1$ and obtain CLP$_{core1}$=$\{3,3\}$. We can also calculate the inter-core CLP of Figure 5.3a. Cache hits from $core_0$ and $core_1$ access all four LLC banks in phases $t1$ and $t3$, respectively, resulting in inter-core CLPs as CLP$_{t1}$=$\{4\}$ and CLP$_{t3}$=$\{4\}$.

Note that, while Figure 5.3b gives us the maximized MLP, CLP is not optimized. In fact, the CLP in Figure 5.3b is even worse compared to the CLP in Figure 5.3a. Specifically, in Figure 5.3b, the intra-core CLPs are CLP$_{core0}$=$\{2,2\}$ and CLP$_{core1}$=$\{3,2\}$, and the inter-core CLPs are CLP$_{t1}$=$\{3\}$ and CLP$_{t3}$=$\{3\}$, which are lower compared to Figure 5.3a.

Finally, in Figure 5.3c, we take the optimization of CLP into account while performing our loop iteration scheduling. This gives us the optimized MLP as well as the optimized CLP. Specifically, in this case, we have intra-core CLP$_{core0}$=$\{4,3\}$, CLP$_{core1}$=$\{3,4\}$, and inter-core CLP$_{t1}$=$\{4\}$, CLP$_{t3}$=$\{4\}$. It should be emphasized that, in our discussion so far, we have mainly focused on the MLP-first approach.

5.5 Details of the Optimizations

5.5.1 Formalization

We now define four important concepts employed by our compiler framework: *iteration block (IB)*, *iteration window (IW)*, *data block (DB)*, and *data set (DS)*. Among these four concepts, IB and IW are defined on *iteration space*, whereas DB and DS are defined on *data space*. The iteration space of an m-level nested loop can be represented by an m-dimensional vector $\vec{i} = (i_1, i_2, \cdots, i_m)^T$, delimited by loop bounds $\{(l_1, u_1), (l_2, u_2), \cdots, (l_m, u_m)\}$, where $l_k \leq i_k \leq u_k$ and $1 \leq k \leq m$. Each loop iteration is represented using an iteration vector $\vec{i}$. Similarly, the data space for an n-

dimensional array can be represented by an n-dimensional vector $\vec{j} = (j_1, j_2, \cdots, j_n)^T$ where j_k $(1 \leq k \leq n)$ is the index of array element. Each array reference is represented by a mapping from iteration space to data space. Given a loop iteration vector $\vec{i}$, the corresponding array reference (i.e., array index) is $\vec{r} = A\vec{i} + \vec{o}$, where A is the reference matrix and $\vec{o}$ is the reference offset[3]. For example, the reference to array $A[i_1 + i_2][i_2 + 2]$ in two-level nested loop is represented as $\vec{r} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 0 \\ 2 \end{pmatrix}$, where $\vec{i} = (i_1, i_2)^T$.

Iteration Block (IB): An iteration block is the granularity at which loop iterations are distributed across multiple cores for execution. Given an m-level loop nest with $\vec{i} = (i_1, i_2, \cdots, i_m)^T$, an iteration block IB is defined as:

(1) $IB = \{\vec{i_1}, \vec{i_2}, \cdots, \vec{i_q}\}$, where each $\vec{i_x}, (1 \leq x \leq q)$ is one loop iteration.

(2) For any two loop iterations $\vec{i_x} = (\underbrace{i_{x_1}, i_{x_2}, \cdots, i_{x_t}}_{t}, i_{x_{t+1}}, \cdots, i_{x_n})^T$ and $\vec{i_y} = (\underbrace{i_{y_1}, i_{y_2}, \cdots, i_{y_t}}_{t}, i_{y_{t+1}}, \cdots, i_{y_n})^T$ $(1 < x, y < q,\ 1 < t < n)$ that belong to the same IB, we have $i_{x_v} = i_{y_v}$ $(1 <= v <= t)$.

The value of t determines the outermost t-level loops can be potentially parallelized across multiple cores. In other words, it specifies the size of an IB. Choosing a proper t involves a tradeoff between parallelism and cache locality. Specifically, a small value of t indicates a parallelization of the loop iterations across different cores in big chunks (IBs), in which, consecutive data accesses to the same cache line are contained in a single chunk and assigned to one core. While this scenario is good for cache locality, there are two drawbacks of using large-sized IB. First, a large block would typically access many memory banks. As a result, we lose the flexibility of scheduling IBs towards optimizing MLP. Second, large blocks can also lead to imbalanced computation among cores and sacrifice performance due to less parallelism. On the other hand, a large value of t (small-sized IBs) can hurt cache locality. This is due to the fact that subsequent accesses to the same cache line might be distributed across different cores, resulting in extra cache misses from different cores. In our

[3]In this chapter, we focus on affine programs [118]. That is, the loop bounds and array references are assumed to be affine functions of the loop iterators. For an application program that has both affine and non-affine references, our approach optimizes only the affine ones.

framework, we choose to use proper small-sized iteration blocks so that the loop nest can be parallelized in a fine-granular and balanced fashion (Section 5.5.3). Meanwhile, loop iterations that access the same cache line are grouped into the same IB. Therefore, the temporal locality of accesses to a cache line is maintained within an IB.

Iteration Window (IW): An iteration window is a group of iteration blocks assigned to a core. Formally, an IW assigned to core c_i is denoted as $IW_{(i,j)}$ with $1 \leq i \leq C$, where C is the total number of cores, and i and j denote the core ID and IW ID, respectively. In our approach, an IW follows two rules:

(1) For any $IW_{(i,j)}$, $IW_{(i,k)}$ $(j < k)$ from core c_i, loop iterations in $IW_{(i,j)}$ are executed before loop iterations in $IW_{(i,k)}$.

(2) For any two $IW_{(m,j)}$, $IW_{(n,j)}$ from cores c_m and c_n, loop iterations from $IW_{(m,j)}$ and $IW_{(n,j)}$ are expected to execute concurrently at runtime.

Rule (1) captures the execution order of IWs within a core, whereas rule (2) provides concurrent execution of IWs from multiple cores. Each $IW_{(i,j)}$ can be expanded as a set of IBs, i.e., $\{IB_{(i,j,1)}, IB_{(i,j,2)}, \cdots, IB_{(i,j,n)}\}$. In general, we use a large-sized IWs, so that rule (2) can be satisfied at runtime. However, the size of IW cannot be arbitrary large. This is because our loop permutation changes the data access order within an IW, and we do *not* want to introduce extra cache misses within an IW. We discuss how we choose a proper size of IW in Section 5.5.3.

Data Block (DB): A data block is a group of data elements (addresses) in data space. Specifically, an DB can be expressed as a set of array elements accessed by array references $(\{\vec{r_1}, \vec{r_2}, \cdots, \vec{r_p}\})$. In our framework, we use the *cache line size* to determine the DB size (p). That is, array references in the same DB are mapped to the same cache line, and can potentially benefit from spatial locality. It is possible that multiple DBs are mapped to a single cache line, but not the other way around. Recall that data accesses to an n-dimensional array can be represented as vectors in the data space ($\vec{r_k} = (r_{k_1}, r_{k_2}, \cdots, r_{k_n})^T$, where $1 \leq k \leq p$, and r_{k_i} $(1 \leq i \leq n)$ is the array index in the ith dimension). An DB can formally be defined as:

(1) $DB = \{\vec{r_1}, \vec{r_2}, \cdots, \vec{r_p}\}$, and

(2) For any two $\vec{r_x} = (\underbrace{r_{x_1}, r_{x_2}, \cdots, r_{x_u}}_{u}, r_{x_{u+1}}, \cdots, r_{x_n})^T$ and $\vec{r_y} = (\underbrace{r_{y_1}, r_{y_2}, \cdots, r_{y_u}}_{u}, r_{y_{u+1}}, \cdots, r_{y_n})^T$ $(1 \leq x, y \leq p,\ 1 \leq u \leq n)$ in the same DB,

we have $r_{x_v} = r_{y_v}$ for any v where $1 \leq v \leq u$ and u is determined by the cache line size.

Data Set (DS): A data set is a group of DBs referenced by loop iterations from the *same* iteration block (IB). The data blocks (DBs) referenced by an IB can be inferred by applying reference matrix and offset to each iteration vector in the IB. Specifically, an DS can be defined as follows:

(1) $DS = \{DB_1, DB_2, \cdots, DB_n\}$, with $DB_x = \{\vec{r_{x1}}, \vec{r_{x2}}, \cdots, \vec{r_{xq}}\}$, where $\vec{r_{xj}} = A\vec{i_{xj}} + \vec{o}$ $(1 \leq j \leq q)$.

(2) Given an array reference $\vec{r_x}$ from DS, if $\vec{r_x} \in DB_i$, then DB_i is included in DS.

To summarize, loop iterations in a loop nest are grouped into IBs and IB is the granularity we use to parallelize and schedule a loop nest (distribute its iterations) across multiple cores. IBs within the same IW can execute in any order without compromising cache locality. Further, IWs are executed sequentially within each core. The array references made by an IB collectively constitute an DS. A DS may contain several DBs. Array references to the same cache line are grouped into the same DB.

5.5.2 Optimization Goal

With these four concepts (IB, IW, DB, and DS) in place, we are now ready to discuss our optimization target. Each DB is associated with a memory bank and an LLC bank, based on its address. We use *memory bank vector* to represent the memory bank of an DB. Given a memory bank vector $\vec{b} = (b_1, b_2, \cdots, b_n)$, where n is the total number of memory banks, bit b_i $(1 \leq i \leq n)$ is set to 1 in a bank vector, if the requested DB (cache line) is mapped to memory bank i. Similarly, for LLC banks, we define *LLC bank vectors* as $\vec{c} = (c_1, c_2, \cdots, c_l)$, where l is the total number of LLC banks. We use $\sum \vec{b}$ and $\sum \vec{c}$ to denote the total number of 1s in memory bank vector and LLC bank vector, respectively. Obviously, for $\vec{b}$ and $\vec{c}$ of a given DB, we have $\sum \vec{b}_{DB} = 1$ and $\sum \vec{c}_{DB} = 1$, indicating that the references to an DB only access *one* memory bank and *one* LLC bank.

An IB accesses a set of DBs (i.e., an DS). To capture the memory banks accessed by an IB, we apply bit-wise *or* ($\cup$) operation over all the bank vectors associated with the DBs in an DS. Specifically, the bank vector of an IB is expressed as

Algorithm 5 MLP, CLP aware iteration block scheduling (Single array, Manycore).

INPUT: Number of cores (N); Size of iteration window (W); Number of total iteration blocks (M);
OUTPUT: Iteration windows to core mapping.

```
1: function GET_MLP_OF_IB(IterationBlockPool)
2:     for each $IB_i$ in $IterationBlockPool$ do
3:         $\vec{b}_{IB_i} \leftarrow \vec{0}$
4:         $data\_blocks \leftarrow get\_data\_blocks(IB_i)$
5:         for each data block $DB_j$in $data\_blocks$ do
6:             $\vec{b}_{IB_i} \uplus get\_MLP\_Vector(DB_j)$
7:         end for
8:     end for
9:     return $(\vec{b}_{IB})$
10: end function
11: function SINGLE_CORE(W, $MLP_{IB}$, IterationBlockPool, $\vec{g}$)
12:     $iterationWindow \leftarrow \varnothing$
13:     $\vec{l} \leftarrow \vec{0}$ Intra-core MLP vector
14:     $TempSet \leftarrow \varnothing$
15:     for each $IB_i$ in $IterationBlockPool$ do
16:         $\overrightarrow{temp_g} \leftarrow \overrightarrow{MLP}_{IB_i} \uplus \vec{g}$ //calculate inter-core MLP
17:         if $\sum \overrightarrow{temp_g} > \sum \vec{g}$ then
18:             $IterationWindow \cup b_i$
19:             Update IterationBlockPool and W
20:             $\vec{l} \leftarrow \vec{l} \uplus \overrightarrow{MLP}_{IB_i}$;   $\vec{g} \leftarrow \overrightarrow{temp_g}$
21:             Break if $W == 0$
22:         else if $\sum \overrightarrow{temp_g} == \sum \vec{g}$ then
23:             $\overrightarrow{temp_l} \leftarrow \overrightarrow{MLP}_{IB_i} \uplus \vec{l}$ //calculate intra-core MLP
24:             if $\sum \overrightarrow{temp_l} > \sum \vec{l}$ then
25:                 $IterationWindow \cup b_i$
26:                 Update IterationBlockPool and W
27:                 $\vec{l} \leftarrow \overrightarrow{temp_l}$
28:                 Break if $W == 0$
29:             else if $\sum \overrightarrow{temp_l} == \sum \vec{l}$ then
30:                 $TempSet \cup IB_i$
31:             end if
32:         end if
33:         if $\sum(\vec{g} \uplus \vec{l}) \geq \beta \times MAX\_MLP$ then
34:             Add remaining IBs in IterationBlockPool to TempSet; Break
35:         end if
36:     end for
37:     if $W \neq 0$ then
38:         for each iteration block $IB_i$ in $TempSet$ do
39:             Choose $IB_i$ if it improve CLP, $remove\ IB_i\ from\ IterationBlockPool\ and\ TempSet$
40:             Break if $W == 0$
41:         end for
42:     end if
43:     return $(IterationWindow, \vec{g})$
44: end function
45: $k \leftarrow 0$
46: Generate dependency of IterationBlockPool
47: $MLP_{IB}$ = GET_MLP_OF_IB(IterationBlockPool)
48: while There are iteration blocks in IterationBlockPool do
49:     $\vec{g} \leftarrow \vec{0}$ //Inter-core MLP vector
50:     for $C_i\ from\ C_1\ to\ C_N$ do
51:         $(IW_{k,C_i}, \vec{g})$ = SINGLE_CORE($W$, $MLP_{IB}$, IterationBlockPool, $\vec{g}$)
52:         $schedule_k \cup IW_{k,C_i}$
53:     end for
54:     $k \leftarrow k + 1$ //Increase 1 schedule time unit
55: end while
```

```
(a) original code
for (j =0; j < m; j++)
    for (i = 2; i < n; i++)
        b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]
```

```
(b) naive parallelization
core 1:
for (j = 0; j < m/2; j++)
    for (i = 2; i < n; i++)
        b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]

core 2:
for (j = m/2; j < m; j++)
    for (i = 2; i < n; i++)
        b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]
```

```
(c) loop strip mining
for (iw = 0; iw < m; iw+=iw_size )
    for (j = iw; j < iw+iw_size; j++)
        for (ib = 2; ib < n; ib+=ib_size)
            for (i = ib; i < ib+ib_size; i++)
                b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]
```

```
(d) loop iteration block scheduling
core 1:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
    for (each ib in iw) /**number of iteration blocks in a
window */
        for (i = ib; i < ib+ib_size; i++)
            j' = calculate(iw, ib)
            b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]

core 2:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
    for (each ib in iw) /**number of iteration blocks in a
window */
        for (i = ib; i < ib+ib_size; i++)
            j' = calculate(iw, ib)
            b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]
```

```
(e) loop permutation
core 1:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
    for (i = ib; i < ib+ib_size; i++)
        for (each ib in iw) /**number of iteration blocks in
a   window */
            j' = calculate(iw, ib)
            b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]

core 2:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
    for (i = ib; i < ib+ib_size; i++)
        for (each ib in iw) /**number of iteration blocks in
a   window */
            j' = calculate(iw, ib)
            b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]
```

Figure 5.4: An example code fragment and its transformed versions after applying our optimizations.

$\vec{b}_{IB} = \uplus\{\vec{b}_{DB_1}, \vec{b}_{DB_2}, \cdots, \vec{b}_{DB_n}\}$.

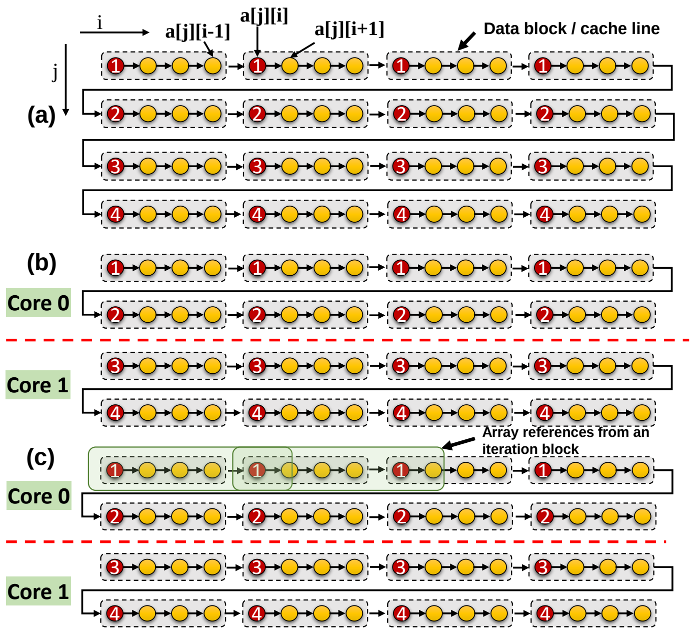
To optimize MLP, we further apply bitwise *or* ($\uplus$) on memory bank vectors among IBs. At each scheduling step[4], we try to maximize $\sum \uplus \vec{b}_{(i,j,k)}$, where i is the core id, j is the IW id, and k is the IB id. For inter-core MLP, we choose the IBs such that $\sum \uplus \vec{b}_{(i,\boldsymbol{J},k)}$ is maximum for a given IW J. Similarly, for intra-core MLP, we maximize $\sum \uplus \vec{b}_{(\boldsymbol{I},\boldsymbol{J},k)}$ for a given core I and a given IW J.

In the CLP-first approach on the other hand, we calculate CLP using the same approach discussed above, and the only difference is that we replace memory bank vector with LLC bank vector ($\vec{c}$) as our primary optimization target is CLP.

5.5.3 Loop Strip-Mining

Let us consider the two-level loop nest shown in Figure 5.4a. There are three references to array a and one reference to array b in each innermost loop iteration (ith dimension). Focusing on array a, the corresponding data access pattern is plotted in Figure 5.5a. We assume that there is a total of 4 memory banks (the number in the oval is the bank ID). Figure 5.4b and Figure 5.5b show the results of parallelizing the outer j loop between two cores without applying our approach. In our framework, we first apply loop strip-mining based on the iteration window size and the iteration block

[4]A scheduling step is a round of assigning iteration blocks to cores. More specifically, at each scheduling step, we assign each core a number of iteration blocks. This number is determined by the iteration window size.



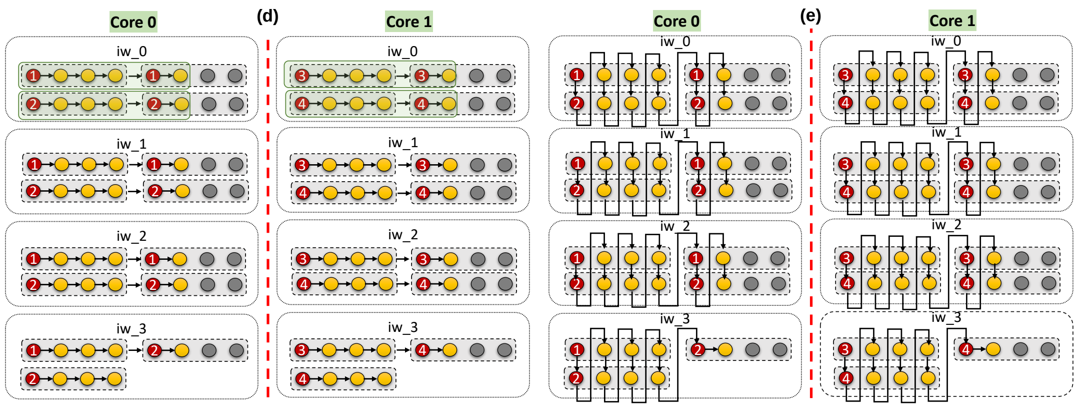


Figure 5.5: The corresponding memory access pattern of the code example in Figure 5.4. (a) Default access pattern. (b) Parallelization between two cores. (c) Forming iteration block and iteration window. (d) Iteration block scheduling. (e) Loop permutation to cluster cache misses.

size to eliminate the potential extra cache misses.

Iteration block size: Recall from Section 5.5.1 that small-sized IBs can lead to extra cache misses, whereas large-sized IBs can reduce MLP. In the example shown in Figure 5.4a, there are three data references to array *a* in each innermost loop iteration ($a[j][i-1]$, $a[j][i]$, and $a[j][i+1]$). As the iterator (i) moves forward, these three references also move forward as a "group", as illustrated in Figure 5.5a. Since these three references move across the boundaries between two neighboring DBs (highlighted with square in the figure), it is simply impossible to have each IB access only one data block. Let us assume in this case that all three data references go to the same DB from iteration 1 to $k-1$ of the inner ith loop. In the kth iteration, $a[j][k+1]$ refers to a data element in the second DB, whereas $a[j][k-1]$ and $a[j][k]$ still refer to the data elements in the first DB. Similarly, in the $(k+1)$th iteration, $a[j][k+2]$ and $a[j][k+1]$ refer to the data elements in the second DB, whereas only $a[j][k]$ refers to a data element in the first DB. Now, the question is: should we group iterations k and $k+1$ in the first IB or in the second IB? Let us assume there are two different iteration blocks (IB_1, IB_2) that are assigned to two different cores (c_1, c_2). We want iterations k and $k+1$ both in either IB_1 or IB_2 so that only one core (either c_1 or c_2) accesses both the DBs and the other core accesses only one DB. For instance, the consequence of grouping iterations k and $k+1$ in IB_1 is that IB_2 starts with iteration $k+2$ which does not access the first DB, as all of the array indices ($a[j][k+1]$, $a[j][k+2]$, $a[j][k+3]$) are in the second DB. Otherwise, c_1 needs to access 2 DBs, and c_2 also needs to access 2 DBs.

Iteration window size: IBs are grouped into iteration windows. The size of an IW is decided by the cache capacity. Specifically,

$$IW\_size = C/(D*n), \tag{5.1}$$

where C is the cache capacity in terms of the number of cache lines, D is the size of DS, and n is the number of cores. Since the size of DS captures the number of DBs (cache lines) accessed by an IB, the IW size captures the number of maximum IBs per window without hurting cache locality.

5.5.4 Loop Iteration Block Scheduling

After the sizes of IB and IW are determined, we apply loop-strip-mining (Figure 5.5c). Then, the loop nest after strip-mining is treated as consecutive iteration blocks which are input to our proposed IB scheduling (IterationBlockPool in Algorithm 1). We give the formal description of our scheduling approach in **Algorithm 1**. At high-level, Our scheme tries to achieve two objectives: 1) at each time unit of scheduling, we choose an IB that improves inter-core MLP as well as intra-core MLP, and 2) once the MLP cannot be improved any further (in the MLP-first approach), CLP is considered as the next optimization target.[5] To do that, there are three steps involved in the algorithm: 1) dependence analysis at IB granularity (line 41), 2) obtaining the MLP vector of each IB (line 42), and 3) scheduling IBs for each core (lines 45 to 47).

Our approach starts by building dependence graph (DAG) of iteration blocks. If two IBs are dependent, the necessary ordering is enforced by inserting synchronization between those two dependent iterations blocks. In subsequent IB scheduling, we always try to select independent IBs for inter-core optimization. If we cannot find independent IB, dependent IBs are chosen and correctness is guaranteed by synchronizations. We want to emphasize that, to reduce the overhead of synchronization, after scheduling all IBs, we perform a "transitive closure" based synchronization minimization strategy to remove redundant synchronizations.

Our loop iteration scheduling chooses IB from in *IterationBlockPool* (line 46). We define one scheduling cycle (time unit) as a round of assigning IBs to each core which fills an iteration window (lines 9 to 39). The scheduling ends when all the iteration blocks in *IterationBlockPool* have been scheduled. Our IB scheduling consists of three major steps: for each core and for each iteration window, (1) we first choose IBs that optimize inter-core MLP (lines 15 to 20), and then, (2) we select IBs that maximize intra-core MLP (lines 22 to 29), and finally, (3) if the iteration window still have slots and the preferred value of MLP is reached (line 30), we choose IBs that optimize CLP (lines 33 to 38). To be more specific, if a candidate IB contributes more to the inter-core MLP or intra-core MLP, we schedule that IB in the current IW. Otherwise, if the candidate provides the same MLP, we add it to a *TempSet*, which holds all the candidates that can be used for improving CLP. We use a factor β (line 30) to balance CLP and MLP (details are provided in Section 5.5.6). The complexity of this algorithm is $O(NM^2)$, where N is the number of cores and M is the total number of iteration blocks.

Figures 5.4d and 5.5d give an example code fragment and the corresponding array

[5]In the CLP-first approach on the other hand, we try choose an IB that improves CLP, and once the CLP cannot be improved any further, MLP optimization is attempted.

reference pattern, respectively, after applying our iteration block scheduling. As can be seen from Figure 5.5d, the inter-core MLP is 4 for each of the four iteration windows, and at the same time, the intra-core MLP is 2. Clearly, compared to the default access pattern depicted in Figure 5.5c (where MLP is 2 for inter-core and 1 for intra-core), *both* inter-core and intra-core MLP are improved.

5.5.5 Loop Permutation

At this point, iteration blocks are scheduled across cores and iteration windows are formed. The last step is loop permutation. Recall from our discussion in Section 5.3 that we interchange the innermost loop with the second innermost loop such that the misses are grouped together to reach an improved MLP. We apply loop permutation to loop iterations within an iteration window. Since all the data blocks accessed by the iterations within an iteration window across cores can fit in the cache (due to our selection of the size of iteration window), our permutation will t not cause any extra cache misses. Figures 5.4e and 5.5e depict the loop nest body and the array reference pattern, respectively, after applying loop permutation.

Table 5.1: An example illustrating the trade-off between MLP and CLP.

	Memory Banks				LLC Banks			
	b_0	b_1	b_2	b_3	b_0	b_1	b_2	b_3
IB_0	0	1	3	0	2	0	4	0
IB_1	1	2	0	1	2	0	4	0
IB_2	0	0	2	2	0	2	0	4
IB_3	2	2	0	0	0	3	0	3

As a summary, our approach takes iteration blocks (IterationBlockPool in Algorithm 1) as input, where each iteration block has its unique identifier (id). Then, Algorithm 1 picks up iteration blocks from IterationBlockPool and forms iteration window across the cores. As a result, each core is associated with a sequence of iteration blocks. During execution, array reference index is derived from iteration window id and iteration block id. Note that, our approach generates the transformed source code of each loop nest. In the experimental results discussed in Section 5.6 (in both the simulation and KNL experiments), we enabled all vectorization and data locality optimizations. In KNL, we used Intel compiler (icc) to generate code.

In simulation-based experiments, gcc code generator (with the highest optimization level) is used.

5.5.6 Striking a Balance between CLP and MLP

Although our discussion above mainly focuses on MLP-first which takes CLP into consideration only as secondary objective, as an alternative approach, one can also choose CLP as the primary optimization goal over MLP, or even choose to trade MLP for CLP. To explain the benefits of doing so, let us consider the access patterns of four iteration blocks, shown in Table 5.1. Each value shown in the table represents the number of accesses to a particular memory/LLC bank. Let us assume that IB_0 has already been scheduled, and we are now choosing the next IB from IB_1, IB_2, and IB_3. Based on our previous discussion, we choose IB_2 since $\sum \vec{b}_{IB_0} \cup \vec{b}_{IB_1}$ is 4, meaning that MLP is maximized to 4. However, doing so is not good from a CLP perspective, as it adds more latency to LLC hits and the value of CLP is only 2 . This is because both IB_0 and IB_1 access LLC banks b_0 and b_2, resulting in a total of 4 (2+2) and 8 (4+4) accesses to LLC banks b_0 and b_2, respectively. Since the hits in an LLC bank compete with one another, this cache contention can easily offset the potential benefits coming from optimized MLP.

As a result, one may want to explore a more "balanced approach" between MLP and CLP. We enable a tradeoff between MLP and CLP by determining the value of parameter β in **Algorithm 1** (line 30). More specifically, we employ the following optimization target:

$$Target\_Metric = \beta \times MLP + (1 - \beta) \times CLP, \tag{5.2}$$

where we have $0 \leq \beta \leq 1$. To determine an optimum value for β, we first need to augment our approach explained so far to take the number of bank accesses into consideration. More specifically, we use integer values (instead of boolean values which are used in previous discussion) for the entries in a bank vector of an IB, so that we can capture the number of accesses made to a given bank. Also, we define an operation, $\uplus$, which performs the entry-wise addition between two bank vectors, and compute the *standard deviation* (SD) for the weighted MLP vector. Note that, SD captures the distribution of accesses across different banks. In particular, a higher

SD indicates that the accesses are not balanced and some banks have long service queues populated by many accesses, whereas other banks have only a few accesses.

Let us assume that the request service latency at memory banks is η_m and the service latency at LLC banks is η_n. Let us further assume that the weighted bank MLP and CLP vectors for the current iteration block $IB_{(c)urrent}$ are $\vec{m_c}$ and $\vec{c_c}$, respectively, and we are selecting the next iteration block from IB_x and IB_y. We use $\vec{m_x}$ and $\vec{c_x}$ to denote, respectively, the MLP vector and CLP vector of IB_x. Similarly for IB_y, we have $\vec{m_y}$ and $\vec{c_y}$. We can calculate the corresponding standard deviation between IB_c and IB_x using $SD_m(IB_c, IB_x) = SD_m((\vec{m_c} \uplus \vec{m_x}) * \eta_m)$. In terms of CLP, the corresponding SD is $SD_c(IB_c, IB_x) = SD_c((\vec{c_c} \uplus \vec{c_x}) * \eta_c)$. Therefore, we have $SD(IB_c, IB_x) = SD_m(IB_c, IB_x) + SD_c(IB_c, IB_x)$. We then define $\delta = SD(IB_c, IB_x) - SD(IB_c, IB_y)$ to capture the *difference* of SDs between two iteration blocks IB_x and IB_y. If $\delta > 0$, it indicates that choosing IB_y is better since a smaller value of SD indicates a more balanced distribution of accesses to different banks. On the other hand, if $\delta = 0$, we randomly choose one iteration block from IB_x and IB_y.

Note that, to reduce potential overheads, we only perform SD analysis for the first two iteration blocks of each iteration window. After we select the first two blocks, we check the MLP vector and CLP vector of the chosen iteration block and determine the value of β. In other words, we choose the most beneficial iteration blocks (in terms of a both CLP and MLP) at the beginning of constructing an iteration window. We then use the obtained β to choose the successive iteration blocks for that iteration window. That is, our compiler automatically determines the value of β for each iteration window. Later, we present the distribution of β values determined by our approach.

5.5.7 Discussion

We now discuss the generality of our approach. If the target system employs dynamic NUCA (DNUCA) where a cache line doesn't have a fixed home bank and can reside in any cache bank in the system, our approach can be augmented to predict the locations (LLC bank) for DBs (for the next scheduling epoch). Equipped with such prediction, our approach can be used for DNUCA as well. If the LLCs are private (which is not very common), our approach will have limited impact on CLP. However, we want to emphasize that, even when the target-system employs DNUCA or private LLC, our framework will still improve MLP. Also, our approach works with different NoC topologies (e.g., mesh, butterfly, etc). As long as the information of cache/memory placement, cache management policy (i.e., SNUCA or DNUCA), and network topology are exposed to our compiler, we can apply our optimization on CLP and MLP.

5.6 Experimental Evaluation

5.6.1 Setup

We conducted both a simulation based study as well as experiments on a commercial manycore architecture. The reason why we performed simulation based experiments is two-fold. First, it is not possible to extract CLP and MLP information from a real hardware, as current performance counters, debugging tools and performance evaluation tools do not provide CLP or MLP statistics. Second, to see how our proposed compiler based approach performs under different architectures, we wanted to change some of the architectural parameters, and this could be done only in a simulation based environment. However, in addition to the simulation based experiments, we also performed experiments on Intel Knight's Landing [8], a commercial manycore/accelerator system. In KNL, each experiment is repeated 15 times, and the median-value is used in the presented-plots. The variance between the lowest and highest values was less than 2% in all experiments.

For our simulation based experiments, we used gem5 [89] infrastructure to execute 12 multithreaded applications in the full system mode. Ten of our twelve multithreaded applications are from Splash-2 [119] and the remaining two are matrix multiplication (mxm) and syr2k, a kernel that performs a rank-2k matrix-matrix operation. The input dataset sizes of these applications vary between 33.1 MB and 1.4 GB, and their LLC (L2) cache miss rates range from 16.6% to 37.2% (in simulator). Also, the number of iteration blocks (as defined in Section 5.5.1) ranges between 8,032 and

Table 5.2: System setup.

Manycore Size, Frequency	36 cores (6×6), 1 GHz, 2-issue
L1 Cache	16 KB; 8-way; 32 bytes/line
L2 Cache	512 KB/core; 16-way; 128 bytes/line
Hardware Prefetcher	stream prefetcher with 32 streams, prefetch degree of 4, prefetch distance of 64 cache lines
Coherence Protocol	MOESI
Router Overhead	3 cycles
Memory Row Size	2 KB
On-Chip Network Frequency	1 GHz
Routing Policy	X-Y routing
DRAM Controller	open-row policy using FR-FCFS scheduling policy 128-entry MSHR and memory request queue
DRAM	DDR4-2400; 250 request buffer entries 4 MCs; 1 rank/channel; 8 banks/rank
Row-Buffer Size	2 KB
Operating System	Linux 4.7
Data Distribution Across Memory Banks	2KB granularity, round-robin
Data Distribution Across LLC Banks	128 bytes granularity, round-robin

31,554.

We used LLVM [120] to implement our compiler support. Table 5.2 gives the important parameters of the manycore/memory configurations modeled in this work using gem5. In most of our simulation-based experiments, we used 36 cores and parallelized each application program such that each core executes one thread at a time (i.e., one-to-one thread-to-core mapping In all experiments, we set the scheduling epoch length 2500 cycles. We also enable both vectorization and the hardware-based prefetcher (stream-prefetcher).

In this work, we compare *eight different versions* for the execution of our application programs:

- *Default:* In this version (also called the *original version*), the iterations of a loop nest are divided into iteration blocks of equal size, and the resulting iteration blocks are assigned to available cores in a *round-robin fashion.* Unless stated otherwise, the results with all the remaining versions described below are *normalized* with respect to this version.
- *Clustering:* This version implements the approach in [81]. While it clusters the LLC misses, it does not specifically consider cache-level parallelism.
- *MLP-first:* In this version, MLP is optimized aggressively using the approach explained in Section 5.5.4. As noted there, CLP is considered, as a secondary optimization, only if doing so does not hurt MLP.

- *CLP-first:* This is the other extreme where CLP is aggressively exploited first, and MLP is considered only if doing so does not hurt CLP.
- *Balanced:* This is the approach defended in Section 5.5.6, where the compiler tries to determine the value of β parameter to *co-optimize* (balance) MLP and CLP so that maximum performance can be achieved. It is important to note that MLP-first and CLP-first are just two different incarnations of Balanced, with $\beta = 1$ and $\beta = 0$, respectively.
- *Locality-Aware-MLP:* This is a recently published compiler approach [115] that targets optimizing bank-level parallelism in a locality-conscious (row-buffer aware) manner. It does not consider CLP; however, it considers row-buffer locality.
- *PAR-BS:* This is a pure *hardware based* optimization scheme proposed by [77]. It (i) handles DRAM requests in batches to provide fairness across competing memory requests coming from different threads, and (ii) employs a parallelism-aware DRAM scheduling policy with the goal of processing requests from threads in parallel in the DRAM banks, to improve memory bank-level parallelism.
- *Ideal:* This version represents the *maximum potential savings.* It is implemented in the simulator by maximizing both MLP and CLP. It, in a sense achieves, at the same time, the MLP performance of MLP-first, and the CLP performance of CLP-first. Note that, this version is *not* practical as CLP and MLP can conflict with one another, and it is *not* always possible to maximize the both at the same time.

While we tested all these versions in our simulator (gem5), the *PAR-BS* and *Ideal* versions could not be used on Intel manycore system, as the former is a pure hardware based scheme that requires architectural modifications and the latter represents a limit study that can be evaluated only in a simulated environment. We also want to emphasize that, unless stated otherwise, all these versions have the *same degree of compute parallelism* and use the *same set of conventional data locality optimizations* (such as loop permutation and tiling that collectively minimize the number of LLC misses), and that they only differ how they map and schedule iteration blocks. The accuracy of the CME implementation employed in estimating LLC misses ranged between 79.14% and 88.36%.

5.6.2 Results with the Manycore Simulator

We present the MLP results in Figure 5.6. As expected, MLP-first generates the best MLP results. While CLP-first performs much worse than MLP-first, it is still better than the default version, as CLP-first considers MLP if doing so does not hurt CLP. Also, Clustering does not perform very well, as mere clustering of memory accesses does *not* guarantee MLP improvement (though it performs, as can be expected, better than the default version). Overall, PAR-BS performs slightly better than Clustering; but, since the throughput optimization it brings is balanced with fairness optimization, its performance is not as good as Balanced. Also, since PAR-BS is a pure hardware optimization, it is not as good as compiler based schemes that have the flexibility of performing whole program analysis. Finally, Locality-Aware-MLP generates comparable MLP results to Balanced, and Balanced outperforms all the versions tested (except MLP-first).

Figure 5.7 presents the CLP results produced by the same versions. It can be observed that Clustering, MLP-first, and Locality-Aware-MLP do not perform very well as far as CLP is concerned, though they are in general better than the default version. This is hardly surprising, as these versions do not specifically target CLP. PAR-BS does not perform any better than Default, primarily because the former mainly targets MLP, not CLP. In comparison, Balanced performs quite well in terms of CLP, and the average CLP values Balanced and CLP-first bring are about 20.59 and 24.16, respectively.

Before presenting the execution cycle results, we want to discuss the impact of these versions on cache miss statistics and row-buffer statistics (as those two metrics are generally affected by how computations are scheduled). Figure 5.8 gives the percentage increases in LLC miss rates (over the default version) when using optimized versions. We observe that, none of these versions (Clustering, MLP-first, CLP-first, Balanced, Locality-Aware-MLP, and PAR-BS) has any noticeable impact on cache miss statistics, compared the default version (the highest increase on the L2 misses over the default version was about 1.48%). This is because Clustering, MLP-first, CLP-first, and Balanced are designed, as explained earlier, to make sure that cache misses are not increased. On the other hand, Locality-Aware-MLP improves row-buffer locality (to

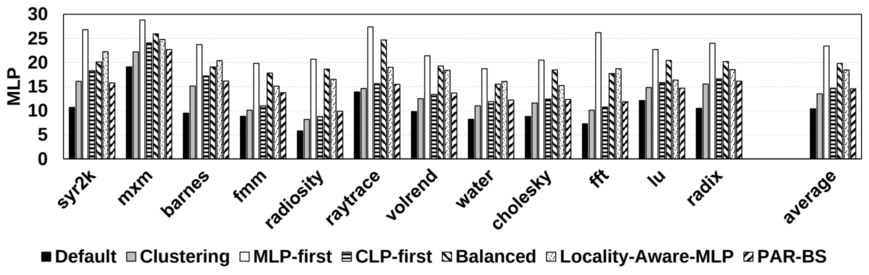


Figure 5.6: MLP Results.

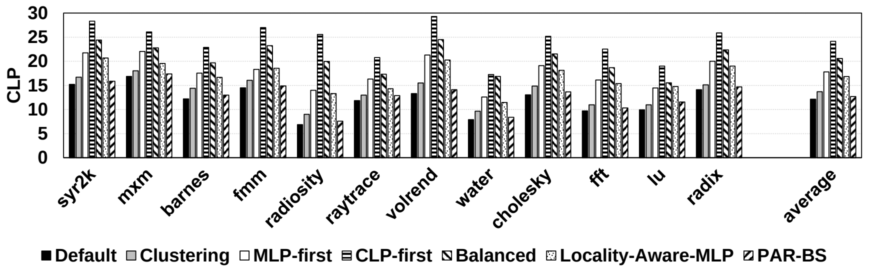


Figure 5.7: CLP Results.

be presented shortly), and that slightly improves, as a side effect, cache hits as well. Figure 5.9, on the other hand, shows the variations (increase) in row-buffer misses, with respect to the default version. These results indicate that, most of the versions tested do not cause significant variations on row-buffer misses (in fact, all observed variations are between -2% and 2%). As expected, Locality-Aware-MLP leads to some improvement on row-buffer misses.

Figure 5.10 gives, for each version, the performance improvement (parallel execution time reduction) it brings over the default version (higher, the better). Note that, in these results, for a given version, all its impact on different metrics (e.g., CLP, MLP, row buffer miss rate, LLC miss rate) as well as all other overheads it incurs are included. We observe from these results that, our defended approach (Balanced) outperforms all remaining versions in all the 12 benchmarks tested (except Ideal, of course). This is because, as explained in Section 5.5.6, Balanced tries to perform the best trade-off between CLP and MLP, instead of trying to optimize one of them very aggressively (which is the case in CLP-first and MLP-first). Clustering does not perform well, as it fails to tap the full potential of bank-level parallelism and cache-level parallelism. On the other hand, Locality-Aware-MLP performs worse than our approach, as it does not consider CLP at all. Similarly, the improvements brought by

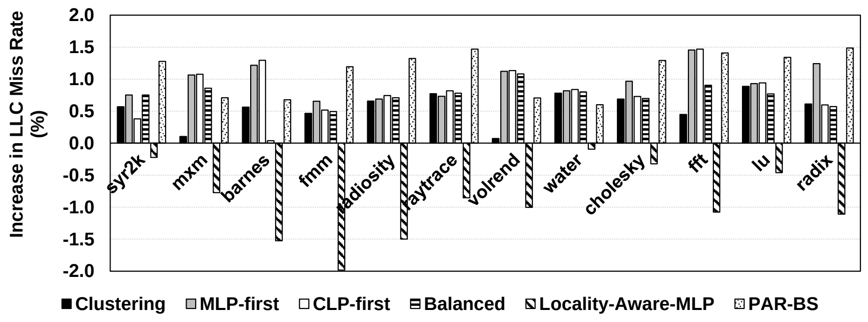


Figure 5.8: Increase in LLC miss rates (lower, the better).

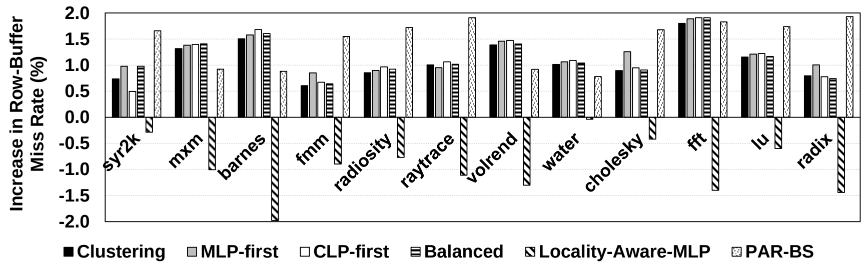


Figure 5.9: Increase in row-buffer miss rates (lower, the better).

PAR-BS are lower than those obtained using Balanced, as the former cannot optimize CLP. Further, note that, PAR-BS requires architecture level modifications, whereas our approach is a software-only solution. Overall, these results clearly underline the importance of optimizing both MLP and CLP together (in fact, Balanced brought an average performance improvement of 17.32% over the default scheme). Finally, the difference between Balanced and Ideal indicates that there is still some additional optimization opportunities that could be exploited by a more sophisticated compiler scheme.

Next, we delve into the behavior of Balanced a bit more, and explain the distribution of the compiler-determined β values across *all* the loop nests of a given application. These distribution results, plotted in Figure 5.11, indicate that, for an overwhelming majority of the loop nests in these 12 applications, the determined β values fall between 0.3 and 0.8, indicating that our approach (Balanced) really balances MLP and CLP quite well. These results also explain why Balanced performs better than CLP-First and MLP-first.

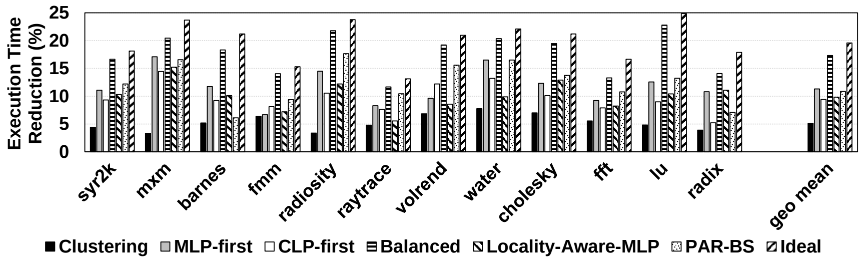


Figure 5.10: Execution time reduction (higher, the better).

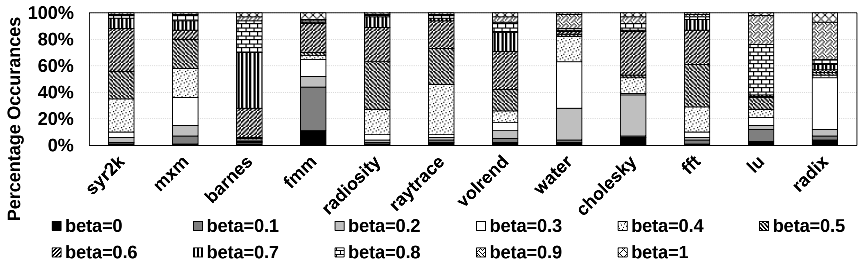


Figure 5.11: Distribution of the compiler-determined β values across all loop nest of our applications.

5.6.3 Results with Intel Knight's Landing

Recall from Section 5.2 that this architecture supports various "memory modes" and "cluster modes". We tested each of the three cluster modes (all-to-all, quadrant and sub-NUMA) under two memory modes. The first memory mode is the "cache mode" where all of the MCDRAM behave as a memory-side direct mapped cache in front of DDR4. Consequently, there is only a single visible pool of memory, and MCDRAM is simply treated as a high bandwidth (L3) cache. The second memory mode used is "hybrid mode" where some of MCDRAM space is configured as memory extension and the remaining MCDRAM space is configured as L3 cache. In this case, we profiled the applications in our experimental suite and allocated the some select data structures from the memory extension part of MCDRAM. Since the observed trends and overall conclusions with the cache and hybrid memory modes were similar, we present results from only the cache mode.

The KNL results are plotted in Figures 5.12, 5.13 and 5.14 for the "all-to-all+cache", "quadrant+cache" and "sub-NUMA+cache" configurations, respectively. For each configuration, we performed two types of experiments: one with O2 compiler flag

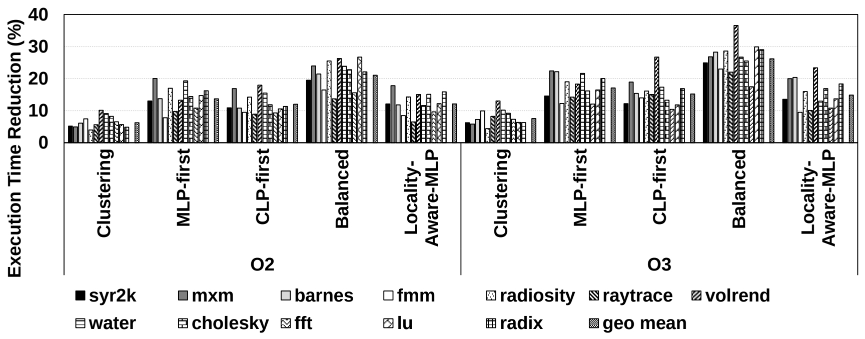


Figure 5.12: Results with “all-to-all” cluster mode and “cache” memory mode.

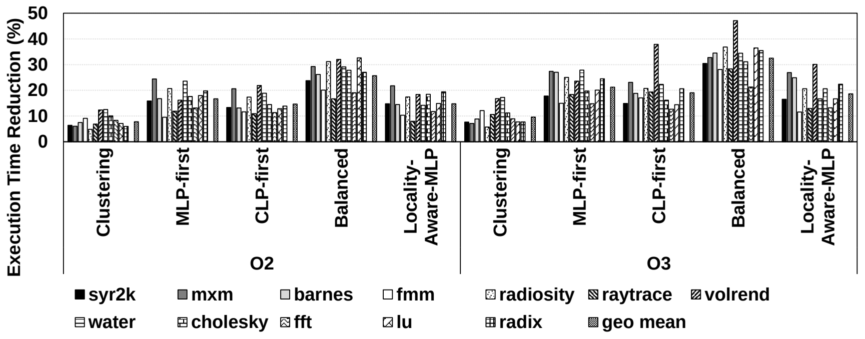


Figure 5.13: Results with “quadrant” cluster mode and “cache” memory mode.

and one with O3 compiler flag. O2 corresponds to default set of *icc* optimizations; it includes vectorization as well as some loop transformations such as loop unrolling and inlining within source file. In O3 on the other hand, the compiler activates all optimizations in O2 level; in addition, it also uses more aggressive loop optimizations such as cache blocking (tiling), loop fusion, and loop interchange.

One can make several observations from the results presented in Figures 5.12, 5.13 and 5.14 (higher, the better). First, our approach improves the performance of *all* cluster nodes in *all* application programs tested. Second, the relative performance variations we observed in our simulation based experiments are valid in Intel Knight’s Landing case as well. In particular, Balanced outperforms the remaining versions under all cluster modes, and MLP-First comes the second. Third, our approach (which is oriented towards reducing the latencies of *both* cache hits and cache misses) blends well with the traditional locality optimizations. More specifically, it can be observed that, as we move from O2 to O3, the overall execution time savings significantly improve. For example, in the case of the quadrant cluster mode, the

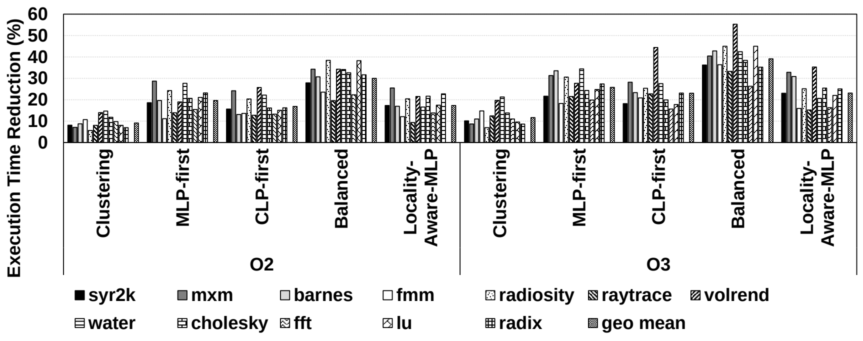


Figure 5.14: Results with "sub-NUMA" cluster mode and "cache" memory mode. average performance improvements brought (over the default version) are 25.69% and 32.54%, under O2 and O3, respectively.

5.7 Conclusion Remarks

Targeting data access parallelism, we propose three alternative optimization strategies: (i) MLP-first, which primarily optimizes memory-level parallelism for LLC misses, (ii) CLP-first, which primarily optimizes cache-level parallelism for LLC hits, and (iii) Balanced, which strikes a balance between MLP and CLP. Our simulations show that the proposed three approaches bring 11.31%, 9.43%, and 17.32% reduction in execution times. We also tested our approach on a commercial manycore architecture, and the results collected indicate 17.06%, 15.19% and 26.15% average execution time savings with MLP-first, CLP-first and Balanced, respectively.

Chapter 6 | Related Work

In this chapter, we discuss the related research efforts from two aspects: i) managing computation and data access in GPGPUs, and ii) managing data access parallelism in manycore CPUs.

6.1 Managing Computation and Data Accesses in GPGPUs

Managing irregular computation: Prior work on dynamic parallelism for GPUs has mainly dealt with the challenges of launch overhead. Wang *et al.* [28] characterize the overheads involved in dynamic parallel applications. They also compare the control-flow and memory behavior of the dynamic parallel applications against their non-dynamic parallel counterparts. Chen *et al.* [61] propose a compiler-based code transformation that replaces the child kernel launches in the parent threads with the child kernel code to reuse the already running parent threads. Therefore, they avoid the large runtime overheads involved in launching child kernels. Their code transformation also load balances the parent threads by reassigning the child tasks to different parent threads. There has been considerable amount of research done on effectively mapping computations of conventional applications to multi threads [51, 94, 104, 110, 121–125]. Yang *et al.* [121] propose a compiler framework called CUDA-NP, that starts execution with a high number of threads which are activated/deactivated by control flow during runtime, essentially distributing the work among the threads. Shen et al. [123] develop a mechanism that can find an optimal partitioning of work between CPU and GPU based on the workload characteristics using a two-step quantitative model. Kim *et al.* [124] investigate a fine-grain hardware worklist for GPGPUs which acts as the center for all the warps to pick up work. This allows the work distribution to load balance itself dynamically during the source of execution. In Chapter 2 of this dissertation, we dynamically tune the workload distribution by controlling the kernel launches, which effectively reduces not only the number of child kernels, but also the number of child CTAs. Consequently, we reduce both launch overheads and queuing latencies. Also, these overhead and latency can be hidden more effectively due to extended executions of parent threads.

Hierarchical scheduling in GPUs: There has been a substantial body of work on building efficient work-group and wavefront scheduling mechanisms for GPUs to improve cache performance, memory bandwidth utilization, DRAM performance, system performance, and energy efficiency [20, 25, 33, 46, 47, 52, 57, 58, 71, 110, 126, 127]. Lee *et al.* [52] proposed work-group and wavefront scheduling techniques which optimize the locality for applications with neighboring work-group data reuse by scheduling work-groups contiguously to CUs rather than in a round-robin fashion. Li *et al.*. [57] developed software-based techniques to improve the inter-work-group locality of an application. Jog *et al.* [71] investigated multiple scheduling techniques to reduce cache contention and improve memory-side prefetching and bank-level parallelism. Lai *et al.* [127] developed a three-stage methodology for mapping threads to cores using a formal model that captures thread characteristics as well as cache sharing behavior. Wang *et al.* [47] proposed work-group scheduling for DP applications, where they bind child WGs to its parent WG in order to exploit parent-child reuse.

Managing data locality in GPUs: There are several prior efforts focusing on improving data access performance on CPUs and GPUs [94, 104, 108, 111, 113, 122, 128]. Improved cache performance in GPUs is mainly achieved via efficient cache management policies ([24, 55, 56, 66, 129–132]), throttling the amount of parallelism ([51, 52, 57, 59]), and cache bypassing ([53, 54, 129, 133]). Oh *et al.* [55] proposed an adaptive prefetching and scheduling mechanism to improve the GPU cache efficiency. They achieve this by grouping together work-groups and monitoring the data access patterns of the wavefronts in a work-group. Koo *et al.* [56] developed an access pattern-aware cache management technique which dynamically detects the type of locality of each load instruction by monitoring a representative wavefront. Chen *et al.* [129] adaptively bypassed memory requests to the cache based on reuse distances to protect against cache contention. In Chapter 3 [102], we quantitatively characterize the data locality opportunities residing in dynamic GPU application, and propose corresponding hierarchical scheduling mechanisms to improve the data locality and application performance.

6.2 Managing Data Access Parallelism in Manycore Systems

Software approaches to improve MLP: Liu et al. [134] proposed an OS based bank-level partitioning scheme, where OS allocates pages to cores (threads) from a particular bank, thereby reducing the interference from the other applications. Pai and Adve [81] introduced the concept of clustering cache misses to improve memory level parallelism. Sung et al. [135] presented data layout transformations for structured grid codes with dynamic allocated arrays. Targeting GPUs, they use the variable length allocation syntax supported by C99 and other languages to collect the required information to perform these transformations. Pai et al. [81] proposed code transformations to increase memory parallelism by overlapping multiple read misses within the same instruction window, while preserving cache locality. Compared to their work, ours focuses on multithreaded applications running on manycores. Further, we propose iteration scheduling upon miss clustering (permutation), to improve inter-core MLP, and we consider CLP as well. In our experimental evaluations, we compared our proposed approach against [81] (which is annotated as *Clustering* in the experimental results). Ding et al. [115] proposed iteration space tile scheduling to improve BLP. Targeting regular codes with affine references, they predict last-level cache misses per tile in a loop nest in the first step, and in the second step, they identify which banks are accessed by the corresponding indices and schedule the tiles such that they increase the BLP. Their approach schedules the tiled loop iterations across cores targeting BLP optimization. It does not consider CLP. We compared our approach to this prior work in our experimental evaluations. Targeting irregular applications, Tang et al. [122] proposed an inspector-executor based loop scheduling to improve bank-level parallelism across cores and row-buffer locality from each core's perspective. We quantitatively and qualitatively compared our approach to those two works in Chapter 5 [136]. Instead of focusing row-buffer locality, we demonstrate that intra-core MLP is also important, and our approach considers both inter-core and intra-core MLP. Further, we consider CLP to improve performance by reducing cache hits latencies.

Hardware approaches to improve MLP: Several techniques [128, 137–140, 140, 141, 141, 142, 142] are proposed to improve memory parallelism. Lee et al. [75] proposed two schemes (1) MSHR issuing policy which prioritizes prefetch requests to different banks ahead of prefetch requests to the same bank to increase the BLP. (2) a BLP preserving scheme that allocates the requests in to memory controller such that the BLP across an application is preserved minimizing the interference from the other applications running on different cores. Mutlu and Moscibroda [77] proposed a scheduler which provides fairness and higher MLP. Their technique improves bank level parallelism by grouping the requests from a thread and servicing them concurrently. Kim et al. [76] proposed that considering a bank as monolithic entity results in high access latencies due to long bitlines. By further dividing the banks in to sub-arrays, the authors have proposed techniques to improve sub-array level parallelism and reduce bank serialization. Their proposal also resulted in increased row-buffer hit rate with multiple rows being maintained in the local row-buffers of the sub-arrays. Qureshi et al. [139] proposed a MLP-aware cache management to reduce the memory stalls. Compared to all these hardware efforts, we reduce the hardware design complexity by employing a "software-only" solution to improve MLP in Chapter 4 of this dissertation.

Irregular applications: There exist many compiler works that target irregular applications. Most of those works focus on data layout optimizations to improve cache locality [143–149]. The other body of work in this area focus on parallelizing irregular applications [150–153]. [91] presented a hierarchical clustering method (GPART) to improve cache locality in irregular applications. Han and Tseng [154] employed graph partitioning to improve locality in irregular applications. Zhong et al. [155] described how an affinity-based hierarchical partitioning of data can improve cache locality. Das et al. [78] are the first one to propose inspector-executor model to identify parallelism in irregular applications. Ding et al. [90] proposed trading cache hit rate for memory performance to improve performance. All these studies aimed at either improving cache performance or parallelism. Our main focus on the other hand is on BLP.

Traditional Data Locality Optimizations: The compiler literature is full of optimization techniques that target reducing the number of cache misses [95–99, 156–169]. There also exist cache bandwidth optimizations [170–172]. Our work presented in this dissertation is fundamentally different from these prior works, as it tries to optimize cache hit and miss latencies, instead of reducing the number of cache misses. Clearly, these two approaches (reducing the number of misses and reducing the miss latency) are complementary, and one would normally need to employ the both to maximize performance benefits (as already demonstrated with our O2/O3 results in KNL).

Chapter 7 Conclusion and Future Work

7.1 Conclusion

Manycore systems have been rapidly penetrated into various platforms including handheld mobiles, desktop computers, high-performance data centers and cloud. The tremendous resources, both homogeneous and heterogeneous, equipped with manycore systems provide applications huge performance potentials from parallelization. However, realizing the underlining parallelism requires detailed investigation of the entire software-hardware stack in manycore systems in order to boost application performance, especially for those applications with irregular computation and data access pattern. To this end, This dissertation takes a holistic approach and quantitatively examines the the reasons causing inefficiencies and ineffectiveness in manycore systems. The particular contributions of this dissertation can be summarized as follows.

First, using GPGPUs as a typical manycore system, this dissertation identifies that the deficiency of irregular application comes from aggressive child kernel launch without knowing the runtime states and hardware limits. It proposes controlled kernel launch which dynamically makes the child kernel launch decision based on the runtime states. Further, it proposes locality-aware hierarchical scheduling in GPGPUs which determines where to execute the launched child kernels such that the significant data reuse is captured by the caches. Together with the controlled kernel launch strategy, this dissertation answers the questions of how many, when, and where to launch child kernels.

Second, focusing on data access performance in manycore systems, this dissertation investigates the data access parallelism opportunities, which are orthogonal optimizations to transitional computation parallelism. Specifically, it proposes a software approach that re-organize and schedule computation to cores such that the memory bank level parallelism is maximized. Further, it investigate a co-optimizing strategy for both memory parallelism and cache parallelism. Those approaches discussed in this dissertation significantly improve data access parallelism and hence the application performance.

7.2 Future Research Directions

The next generation of computing systems is going to be exascale platforms where tremendous heterogeneous/specialized resources (e.g., accelerators, IoT sensors, hybrid memories) are integrated in a single system. This is driven by the fact that the proliferation of real world applications require a variety of system supports to meet certain performance/energy-efficiency/quality of service (QoS) requirements. With more and more specialized resources being integrated to the system, it makes the design more complicated and requires systematic optimizations to guarantee the delivered performance. In particular, the challenges come from *irregularity*, *heterogeneity*, *concurrency*, and *scaling*. My previous research in this dissertation demonstrates that resources in manycore systems are not effectively and efficiently utilized by the applications, and I believe these ineffectiveness and inefficiency are more severe and challenging in exascale heterogeneous platforms. This opens a huge exploration space of research opportunities and makes way for new research avenues such as heterogeneous resource management, computation distribution, data placement, and other topics.

Although the opportunities are exciting and attractive, there exist a lot of challenging/open questions in this field. For example, given heterogeneous resources, different types of resources have different properties and are beneficial to certain execution scenarios. How can applications utilize them properly and take advantage of those resources? Moreover, it is very common that multiple applications execute concurrently on the same platform. How can we efficiently manage the resources shared across different applications? How to perform collaborative management among multiple system layers (i.e., compiler, OS, runtime, and architecture) to fuse the high-level ideas with the low-level implementations without introducing significant overheads? Apart from performance and energy efficiency, how can the system ensure other metrics (e.g., security, fairness)? All such questions are non-trivial and require examination throughout the entire software-hardware stack. These questions and challenges motivate my future research, and I am confident to make significant research progress based on my previous experiences.

7.2.1 Heterogeneous Computing Systems

I think heterogeneity is default in future computing systems ranging from cloud, desktops, mobile devices to wearable devices and IoT. The fundamental question is how to use the heterogeneous resources to improve performance and meet certain requirements such as user experience, energy efficiency, portability, etc. Plenty of challenges and opportunities exist from compiler optimization to innovative architecture designs.

Given CPU-GPU system as an example of heterogeneous platform, since GPU is high-throughput oriented and provides massive parallel threads, it is good for executing parallel portion of an application program. However, the control portion which consists of lots of branches is GPU-unfriendly and is good to execute on CPU as it has higher frequency. Ideally, this partition should be done dynamically and automatically by compiler and runtime. However, programmers currently have to explicitly specify and label the portion of application program to execute either on CPU or on GPU. Such a "master-slave" model has two drawbacks. First, it is difficult or even impossible for the programmers to do an optimal program partitioning especially for applications that are input sensitive and have unpredictable runtime behaviors. Second, the redundant communication and data transfer between CPU and GPU can cause under-utilization of resources and also consume unnecessary PCIe bandwidth. I believe programmers should be freed from the burden of statically partitioning application programs, and the underlying system should be able to *automatically* and *dynamically* manage the computation and data between CPU and GPU based on program characteristics and execution status. This requires system designers to rethink the programming model, compiler and runtime systems, which will lead to many design challenges/questions. For example, can a smart compiler automatically identify code portions? Can the computation migrate dynamically from CPU to GPU or the other way around based on the execution status? When and where should such a migration happen? How are the shared resources (e.g., memory system) managed between CPU and GPU? As a first step, my ongoing work studies the unified memory system in a CPU-GPU system. Initial experiments show that the address translation is inefficient and the hardware translation structures (e.g., TLB) are under-utilized in such a system.

7.2.2 Advancing System Design for Deep Learning

Deep learning are becoming extremely hot recent years and are being widely used in various applications such as object detection, image generation. At high level, the deep learning neural network (DNN) consists of several compute intensive and memory intensive kernels which dominate the execution time. Although intra-kernel optimization has been heavily studied and various libraries and hardware accelerators have been proposed, the inter-kernel behavior receive little attention. In my ongoing study, I found that the preferred computation parallelism, data placement, data layout across compute kernels play a significant role in shaping the performance of DNN applications. Based on this observation, I intend to propose inter-kernel optimizations for large-scale DNNs to boost both training and inference of DNN applications on manycore systems. Specifically, I will research the opportunities through computation parallelization, data layout optimization and data placement.

7.2.3 Beyond Performance and Energy Efficiency

Beyond high performance and energy efficiency, other criterias are also important in certain circumstances and require support from systems. A practical example is security. Recent covert channel and side channel attacks in hardware, such as Meltdown and Spectre in commercial products, have alerted both industrial and academia researchers to the security dangers of attacks on hardware architecture features. Even traditional architecture features such as speculation and prefetching can potentially leak critical and sensitive security information to unauthorized entities. Existing architectural support for security are mainly through two approaches: i) resource partitioning, and ii) randomization, and both are conservative approaches which may compromise application performance severely due to the incurred overheads and resource under-utilization. The challenge is that can we perform fine-grain resource management to reduce the overhead while maintaining the targeted security. Moreover, most of previous efforts focus on single hardware components (e.g., cache or scheduler) separately. One disadvantage is that these approaches lose the entire picture of the system, and making a particular hardware component secure might compromise the security of other components. In my opinion, it is really necessary to view the security problem systematically and leverage the collaboration of compiler, runtime and architecture to design more effective and efficient secure systems.

Bibliography

[1] Wang, J., N. Rubin, A. Sidelnik, and S. Yalamanchili (2015) "Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *ISCA*.

[2] Kim, N. S., T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan (2003) "Leakage current: Moore's law meets static power," *Computer*.

[3] Meindl, J. D. (2003) "Beyond Moore's Law: the interconnect era," *Computing in Science Engineering*.

[4] Kuroda, T. (2001) "CMOS design challenges to power wall," in *Digest of Papers. Microprocesses and Nanotechnology 2001. 2001 International Microprocesses and Nanotechnology Conference (IEEE Cat. No.01EX468)*.

[5] Villa, O., D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally (2014) "Scaling the Power Wall: A Path to Exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[6] "Apple A11," https://en.wikipedia.org/wiki/Apple_A11, online, 2017.

[7] "Get small, go big: Meet the next-gen Snapdragon 835," https://www.qualcomm.com/news/onq/2016/11/17/get-small-go-big-meet-next-gen-snapdragon-835, online, 2016.

[8] Sodani, A., R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu (2016) "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*.

[9] "NVIDIA TESLA V100 GPU ARCHITECTURE," http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[10] Burtscher, M., R. Nasre, and K. Pingali (2012) “A quantitative study of irregular programs on GPUs,” in *2012 IEEE International Symposium on Workload Characterization (IISWC)*.

[11] O’Neil, M. A. and M. Burtscher (2014) “Microarchitectural performance characterization of irregular GPU kernels,” in *2014 IEEE International Symposium on Workload Characterization (IISWC)*.

[12] Wulf, W. A. and S. A. McKee (1995) “Hitting the Memory Wall: Implications of the Obvious,” *SIGARCH Computer Architecture News*.

[13] Mutlu, O. and T. Moscibroda (2007) “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[14] Pratx, G. and L. Xing (2011) “GPU Computing in Medical Physics: A Review,” in *Medical physics*.

[15] Stone, S. S., J. P. Haldar, S. C. Tsao, W. mei W. Hwu, B. P. Sutton, and Z.-P. Liang (2008) “Accelerating advanced MRI reconstructions on GPUs,” *J. Parallel Distributed Computing*.

[16] Schmerken, I. (2009) “Wall Street Accelerates Options Analysis with GPU Technology,” .

[17] NVIDIA (2011) “JP Morgan Speeds Risk Calculations with NVIDIA GPUs,” .

[18] Tang, X., H. An, G. Sun, and D. Fan (2013) “A Video Coding Benchmark Suite for Evaluation of Processor Capability,” in *SNPD*.

[19] Park, S. I., S. P. Ponce, J. Huang, Y. Cao, and F. Quek (2008) “Low-Cost, High-Speed Computer Vision using NVIDIA’s CUDA Architecture,” in *AIPR*.

[20] Liu, G., H. An, W. Han, X. Li, T. Sun, W. Zhou, X. Wei, and X. Tang (2012) “FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[21] Merrill, D., M. Garland, and A. Grimshaw (2012) “Scalable GPU Graph Traversal,” in *PPoPP*.

[22] Mendez-Lojo, M., M. Burtscher, and K. Pingali (2012) “A GPU Implementation of Inclusion-based Points-to Analysis,” in *PPoPP*.

[23] Kulkarni, M., K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew (2007) "Optimistic Parallelism Requires Abstractions," in *PLDI*.

[24] Adhinarayanan, V., I. Paul, J. Greathouse, W. N. Huang, A. Pattnaik, and W. chun Feng (2016) "Measuring and Modeling On-Chip Interconnect Power on Real Hardware," in *IISWC*.

[25] Jog, A., O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2016) "Exploiting Core Criticality for Enhanced Performance in GPUs," in *SIGMETRICS*.

[26] NVIDIA (2012), "Dynamic Parallelism in CUDA," .

[27] AMD (2013) "AMD APP SDK OpenCL User Guide," .

[28] Wang, J. and Y. Sudhakar (2014) "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *IISWC*.

[29] AMD (2013) "AMD APP SDK OpenCL Optimization Guide," .

[30] NVIDIA (2012), "Next Generation CUDA Compute Architecture: Kepler GK110," .

[31] Hong, S., S. K. Kim, T. Oguntebi, and K. Olukotun (2011) "Accelerating CUDA Graph Algorithms at Maximum Warp," in *PPoPP*.

[32] Che, S., B. M. Beckmann, S. K. Reinhardt, and K. Skadron (2013) "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IISWC*.

[33] Jog, A., O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. T. Kandemir, and C. R. Das (2015) "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*.

[34] NVIDIA, "CUDA C/C++ SDK Code Samples," .

[35] Wu, H., D. Li, and M. Becchi (2016) "Compiler-Assisted Workload Consolidation For Efficient Dynamic Parallelism on GPU"," in *IPDPS*.

[36] Ukidave, Y., F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli (2015) "NUPAR: A Benchmark Suite for Modern GPU Architectures," in *ICPE*.

[37] NVIDIA (2015), "CUDA C Programming Guide," .

[38] Kuhl, A. (2010) "Thermodynamic States in Explosion Fields," in *IDS*.

[39] Sanders, P. and C. Schulz (2012) "10th Dimacs Implementation Challenge-Graph Partitioning and Graph Clustering," .

[40] Diamos, G., H. Wu, J. Wang, A. Lele, and S. Yalamanchili (2013) "Relational Algorithms for Multi-bulk-synchronous Processors," in *PPoPP*.

[41] Nai, L., Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin (2015) "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions," in *SC*.

[42] Cheng, H., H. Jiang, J. Yang, Y. Xu, and Y. Shang (2015) "BitMapper: an efficient all-mapper based on bit-vector computing," in *BMC Bioinformatics*.

[43] "National Center for Biotechnology Information," http://www.ncbi.nlm.nih.gov, online, 2016.

[44] Bakhoda, A., G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt (2009) "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*.

[45] NVIDIA (2015), "Profiler User's Guide," .

[46] Rogers, T. G., M. O'Connor, and T. M. Aamodt (2012) "Cache-Conscious Wavefront Scheduling," in *MICRO*.

[47] Wang, J., N. Rubin, A. Sidelnik, and S. Yalamanchili (2016) "LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs," in *ISCA*.

[48] Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[49] Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell (2014) "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14.

[50] Vouzis, P. D. and N. V. Sahinidis (2011) "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, **27**(2), pp. 182–188.

[51] Kayiran, O., A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das (2016) "uC-States: Fine-grained GPU Datapath Power Management," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.

[52] Lee, M., S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu (2014) "Improving GPGPU resource utilization through alternative thread block scheduling," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, IEEE, pp. 260–271.

[53] Li, C., S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou (2015) "Locality-driven dynamic GPU cache bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ACM, pp. 67–77.

[54] Xie, X., Y. Liang, Y. Wang, G. Sun, and T. Wang (2015) "Coordinated static and dynamic cache bypassing for GPUs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, IEEE, pp. 76–88.

[55] Oh, Y., K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram (2016) "APRES: improving cache efficiency by exploiting load characteristics on GPUs," *ACM SIGARCH Computer Architecture News*, **44**(3), pp. 191–203.

[56] Koo, G., Y. Oh, W. W. Ro, and M. Annavaram (2017) "Access pattern-aware cache management for improving data utilization in gpu," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, pp. 307–319.

[57] Li, A., S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal (2017) "Locality-Aware CTA Clustering for Modern GPUs," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[58] Puthoor, S., X. Tang, J. Gross, and B. M. Beckmann (2018) "Oversubscribed Command Queues in GPUs," in *Proceedings of the 11th Workshop on General Purpose GPUs (GPGPU collocated with PPoPP)*.

[59] Tang, X., A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. Kandemir, and C. Das (2017) "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture (HPCA)*.

[60] Hajj, I. E., J. Gomez-Luna, C. Li, L. W. Chang, D. Milojicic, and W. m. Hwu (2016) "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[61] Chen, G. and X. Shen (2015) "Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse," in *MICRO*.

[62] Rhu, M., M. Sullivan, J. Leng, and M. Erez (2013) "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, pp. 86–98.

[63] Beamer, S., K. Asanovic, and D. Patterson (2015) "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *2015 IEEE International Symposium on Workload Characterization*.

[64] Che, S., J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron (2010) "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*.

[65] Narasiman, V., M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt (2011) "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*.

[66] Jia, W., K. A. Shaw, and M. Martonosi (2014) "MRPB: Memory request prioritization for massively parallel processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, IEEE, pp. 272–283.

[67] Ding, C. and Y. Zhong (2003) "Predicting Whole-program Locality Through Reuse Distance Analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

[68] Shen, X., Y. Zhong, and C. Ding (2004) "Locality Phase Prediction," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[69] Appleby, A. (2016), "Murmur Hash 2," https://github.com/aappleby/smhasher/blob/master/src/MurmurHash2.cpp.

[70] Kayiran, O., A. Jog, M. T. Kandemir, and C. R. Das (2013) "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*.

[71] Jog, A., O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2013) "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*.

[72] Pan, A. and V. S. Pai (2013) "Imbalanced Cache Partitioning for Balanced Data-parallel Programs," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*.

[73] Keckler, S. W. (2014) "Rethinking Caches for Throughput Processors: Technical Perspective," *Commun. ACM*.

[74] McKinley, K. S. (2014) "Author Retrospective for Optimizing for Parallelism and Data Locality," in *Proceedings of the 25th International Conference on Supercomputing*.

[75] Lee, C. J., V. Narasiman, O. Mutlu, and Y. N. Patt (2009) "Improving Memory Bank-level Parallelism in the Presence of Prefetching," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[76] Kim, Y., V. Seshadri, D. Lee, J. Liu, and O. Mutlu (2012) "A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM," in *Proceedings of the 39th International Symposium on Computer Architecture*.

[77] Mutlu, O. and T. Moscibroda (2008) "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," *SIGARCH Comput. Archit. News*.

[78] Das, R., M. Uysal, J. Saltz, and Y.-S. Hwang (1994) "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures," *J. Parallel Distrib. Comput.*

[79] Rixner, S. (2004) "Memory Controller Optimizations for Web Servers," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*.

[80] Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens (2000) "Memory Access Scheduling," *SIGARCH Comput. Archit. News*.

[81] Pai, V. S. and S. Adve (1999) "Code Transformations to Improve Memory Parallelism," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*.

[82] Intel Uncore Performance Monitoring Guide, I. (2014) "Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual," .

[83] Lattner, C. and V. Adve (2004) "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization.*

[84] Craik, D. J., A. Kumar, and G. C. Levy (1983) "MOLDYN: a generalized program for the evaluation of molecular dynamics models using nuclear magnetic resonance spin-relaxation data," *Journal of Chemical Information and Computer Sciences.*

[85] Dongarra, J. and M. A. Heroux (2013) "HPCG: Toward a New Metric for Ranking High Performance Computing Systems," .

[86] "MiniFE," https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks.

[87] Aslot, V., M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady (2001) "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," .

[88] Hhonghao, H., Y. Dongjin, H. Yi, and X. Jinqiu (2009) "Preconditioned Gauss-Seidel Iterative Method for Linear Systems," in *Information Technology and Applications. IFITA '09.*

[89] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood (2011) "The Gem5 Simulator," *SIGARCH Comput. Archit. News.*

[90] Ding, W., M. Kandemir, D. Guttman, A. Jog, C. R. Das, and P. Yedlapalli (2014) "Trading Cache Hit Rate for Memory Performance," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation.*

[91] Han, H. and C.-W. Tseng (2006) "Exploiting Locality for Irregular Scientific Codes," *IEEE Trans. Parallel Distrib. Syst.*

[92] Kim, Y., M. Papamichael, O. Mutlu, and M. Harchol-Balter (2010) "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[93] Ebrahimi, E., R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt (2011) "Parallel Application Memory Scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.*

[94] Ding, W., X. Tang, M. Kandemir, Y. Zhang, and E. Kultursay (2015) "Optimizing Off-chip Accesses in Multicores," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[95] Leung, S.-T. and J. Zahorjan (1995) *Optimizing data locality by array restructuring*, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

[96] Kodukula, I., N. Ahmed, and K. Pingali (1997) "Data-centric Multi-level Blocking," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI).*

[97] Wolf, M. E. and M. S. Lam (1991) "A Data Locality Optimizing Algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI).*

[98] Carr, S., K. S. McKinley, and C.-W. Tseng (1994) "Compiler Optimizations for Improving Data Locality," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[99] Kandemir, M., J. Ramanujam, A. Choudhary, and P. Banerjee (2001) "A layout-conscious iteration space transformation technique," in *IEEE Transactions on Computers.*

[100] McKinley, K. S. and O. Temam (1996) "A Quantitative Analysis of Loop Nest Locality," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[101] Sanchez, F. J., A. Gonzalez, and M. Velero (1997) "Static locality analysis for cache management," in *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[102] Tang, X., A. Pattnaik, O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das (2019) "Quantifying Data Locality in Dynamic Parallelism in GPUs," in *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS).*

[103] Sharifi, A., E. Kultursay, M. Kandemir, and C. R. Das (2012) "Addressing End-to-End Memory Access Latency in NoC-Based Multicores," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.

[104] Kandemir, M., H. Zhao, X. Tang, and M. Karakoy (2015) "Memory Row Reuse Distance and Its Role in Optimizing Application Performance," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[105] Kim, Y., D. Han, O. Mutlu, and M. Harchol-Balter (2010) "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *The Sixteenth International Symposium on High-Performance Computer Architecture*.

[106] Hashemi, M., Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt (2016) "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *Proceedings of the 43rd International Symposium on Computer Architecture*.

[107] Skarlatos, D., N. S. Kim, and J. Torrellas (2017) "Pageforge: A Near-memory Content-aware Page-merging Architecture," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.

[108] Tang, X., M. T. Kandemir, H. Zhao, M. Jung, and M. Karakoy (2019) "Computing with Near Data," in *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[109] Pattnaik, A., X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das (2019) "Opportunistic Computing in GPU Architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*.

[110] Pattnaik, A., X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das (2016) "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.

[111] Kislal, O., J. Kotra, X. Tang, M. T. Kandemir, and M. Jung (2018) "Enhancing Computation-to-core Assignment with Physical Location Information," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[112] Kislal, O., J. Kotra, X. Tang, M. Taylan Kandemir, and M. Jung (2017) "POSTER: Location-Aware Computation Mapping for Manycore Processors." in *Proceedings of the 2017 International Conference on Parallel Architectures and Compilation.*

[113] Tang, X., O. Kislal, M. Kandemir, and M. Karakoy (2017) "Data Movement Aware Computation Partitioning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[114] Shrifi, A., W. Ding, D. Guttman, H. Zhao, X. Tang, M. Kandemir, and C. Das (2017) "DEMM: a Dynamic Energy-saving mechanism for Multicore Memories," in *Proceedings of the 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).*

[115] Ding, W., D. Guttman, and M. Kandemir (2014) "Compiler Support for Optimizing Memory Bank-Level Parallelism," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture.*

[116] Kim, C., D. Burger, and S. W. Keckler (2002) "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[117] Steen, S. V. D. and L. Eeckhout (2018) "Modeling Superscalar Processor Memory-Level Parallelism," *IEEE Computer Architecture Letters.*

[118] Feautrier, P. (1992) "Some efficient solutions to the affine scheduling problem. I. One-dimensional time," *International Journal of Parallel Programming.*

[119] Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta (1995) "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of International Symposium on Computer Architecture (ISCA).*

[120] Bondhugula, U., J. Ramanujam, and et al. (2008) " PLuTo: A practical and fully automatic polyhedral program optimization system," in *Proceedings of Programming Language Design And Implementation (PLDI).*

[121] Yang, Y. and H. Zhou (2014) "CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications," in *PPoPP.*

[122] Tang, X., M. Kandemir, P. Yedlapalli, and J. Kotra (2016) "Improving Bank-Level Parallelism for Irregular Applications," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[123] Shen, J., A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips (2014) "Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms," in *ICS*.

[124] Kim, J. Y. and C. Batten (2014) "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *MICRO*.

[125] Puthoor, S., A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers (2016) "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture," in *GPGPU*.

[126] Jog, A., O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2013) "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*.

[127] Lai, B. C. C., H. K. Kuo, and J. Y. Jou (2015) "A Cache Hierarchy Aware Thread Mapping Methodology for GPGPUs," *IEEE Transactions on Computers*.

[128] Ryoo, J., O. Kislal, X. Tang, and M. T. Kandemir (2018) "Quantifying and Optimizing Data Access Parallelism on Manycores," in *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[129] Chen, X., S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu (2014) "Adaptive cache bypass and insertion for many-core accelerators," in *Proceedings of International Workshop on Manycore Embedded Systems*, ACM, p. 1.

[130] Duong, N., D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum (2012) "Improving cache management policies using dynamic reuse distances," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, IEEE, pp. 389–400.

[131] Jia, W., K. A. Shaw, and M. Martonosi (2012) "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proceedings of the 26th ACM international conference on Supercomputing*, ACM, pp. 15–24.

[132] Baghsorkhi, S. S., I. Gelado, M. Delahaye, and W.-m. W. Hwu (2012) "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," in *ACM SIGPLAN Notices*, vol. 47, ACM, pp. 23–34.

[133] Xie, X., Y. Liang, G. Sun, and D. Chen (2013) "An efficient compiler framework for cache bypassing on GPUs," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, IEEE, pp. 516–523.

[134] Liu, L., Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu (2012) "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

[135] Sung, I.-J., J. A. Stratton, and W.-M. W. Hwu (2010) "Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*.

[136] Tang, X., M. Taylan Kandemir, M. Karakoy, and M. Arunachalam (2019) "Co-Optimizing Memory-Level Parallelism and Cache-Level Parallelism," in *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

[137] Chou, Y., B. Fahs, and S. Abraham (2004) "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*.

[138] Eyerman, S. and L. Eeckhout (2007) "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*.

[139] Qureshi, M. K., D. N. Lynch, O. Mutlu, and Y. N. Patt (2006) "A Case for MLP-Aware Cache Replacement," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*.

[140] Jain, A. and C. Lin (2013) "Linearizing Irregular Memory Accesses for Improved Correlated Prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*.

[141] Hur, I. and C. Lin (2004) "Adaptive History-Based Memory Schedulers," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*.

[142] Yedlapalli, P., J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam (2013) "Meeting midway: Improving CMP performance with memory-side prefetching," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*.

[143] Petrank, E. and D. Rawitz (2002) "The Hardness of Cache Conscious Data Placement," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[144] Zhang, C., C. Ding, M. Ogihara, Y. Zhong, and Y. Wu (2006) "A Hierarchical Model of Data Locality," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

[145] Ding, C. and K. Kennedy (1999) "Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time," in *Proceedings of the ACM 1999 Conference on Programming Language Design and Implementation.*

[146] Mellor, J., D. Whalley, and K. Kennedy (2001) "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings," in *International Journal of Parallel Programming.*

[147] Mitchell, N., L. Carter, and J. Ferrante (1999) "Localizing non-affine array references," in *the Proceedings of Parallel Architectures and Compilation Techniques (PACT).*

[148] Strout, M. M., L. Carter, and J. Ferrante (2003) "Compile-time Composition of Run-time Data and Iteration Reorderings," in *Proceedings of the ACM 2003 Conference on Programming Language Design and Implementation.*

[149] Jo, Y. and M. Kulkarni (2012) "Automatically Enhancing Locality for Tree Traversals with Traversal Splicing," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications.*

[150] Kulkarni, M., M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval (2009) "How Much Parallelism is There in Irregular Applications?" in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

[151] Yedlapalli, P., E. Kultursay, and M. T. Kandemir (2011) "Cooperative Parallelization," in *Proceedings of the International Conference on Computer-Aided Design.*

[152] Codrescu, L., D. Wills, and J. Meindl (2001) "Architecture of the Atlas chip-multiprocessor: dynamically parallelizing irregular applications," *Computers, IEEE Transactions on.*

[153] Yu, H. and L. Rauchwerger (2000) "Adaptive Reduction Parallelization Techniques," in *Proceedings of the 14th International Conference on Supercomputing.*

[154] Han, H. and C.-W. Tseng (2001) "Improving Locality for Adaptive Irregular Scientific Codes," in *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science.

[155] Zhong, Y., M. Orlovich, X. Shen, and C. Ding (2004) "Array Regrouping and Structure Splitting Using Whole-program Reference Affinity," in *Proceedings of the ACM 2004 Conference on Programming Language Design and Implementation.*

[156] Song, Y. and Z. Li (1999) "New Tiling Techniques to Improve Cache Temporal Locality," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI).*

[157] Kandemir, M., A. Choudhary, J. Ramanujam, and P. Banerjee (1999) "A matrix-based approach to global locality optimization," in *Journal of Parallel and Distributed Computing.*

[158] Verdoolaege, S., M. Bruynooghe, G. Janssens, and F. Catthoor (2003) "Multi-dimensional incremental loop fusion for data locality," in *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. (ASAP).*

[159] Cierniak, M. and W. Li (1995) "Unifying Data and Control Transformations for Distributed Shared-memory Machines," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI).*

[160] O'Boyle, M. and P. Knijnenburg (2002) "Integrating Loop and Data Transformations for Global Optimization," *J. Parallel Distribute Computer.*

[161] Lim, A. W., G. I. Cheong, and M. S. Lam (1999) "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication," in *ICS.*

[162] Anderson, J. M. and M. S. Lam (1993) "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *PLDI.*

[163] Hall, M. H., S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam (1995) "Detecting Coarse-grain Parallelism Using an Interprocedural Parallelizing Compiler," in *Supercomputing.*

[164] Verghese, B., S. Devine, A. Gupta, and M. Rosenblum (1996) "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *ASPLOS.*

[165] Dashti, M., A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth (2013) "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *ASPLOS*.

[166] Li, W. (1994) *Compiling for NUMA parallel machines*, *Tech. rep.*, Cornell University.

[167] Wolf, M. E. and M. S. Lam (1991) "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*.

[168] Maydan, D. E., S. P. Amarasinghe, and M. S. Lam (1993) "Array-data Flow Analysis and Its Use in Array Privatization," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[169] Karakoy, M., O. Kislal, X. Tang, M. T. Kandemir, and M. Arunachalam (2019) "Architecture-Aware Approximate Computing," in *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[170] Jeong, M. K., D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez (2012) "Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12.

[171] Juan, T., J. J. Navarro, and O. Temam (1997) "Data Caches for Superscalar Processors," in *Proceedings of the 11th International Conference on Supercomputing*, ICS '97.

[172] Sohi, G. S. and M. Franklin (1991) "High-bandwidth Data Memory Systems for Superscalar Processors," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV.

Vita

Xulong Tang

Xulong Tang is a Ph.D. candidate at The Pennsylvania State University where he works with his advisors Dr. Mahmut T. Kandemir and Dr. Chita R. Das. He received his M.S. from the University of Science and Technology of China, USTC (2013) and his B.S. from Harbin Institute of Technology (China, 2010), both in computer science. His research interests lie in the fields of high-performance computing and parallel computer architectures and systems. He has published extensively in these areas in venues including MICRO, HPCA, PLDI, SIGMETRICS, and PACT.