PERFORMANCE MODELING FOR ARCHITECTURAL AND PROGRAM ANALYSIS

by

Yu Jung Lo

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2015

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Yu Jung Lo**

has been approved by the following supervisory committee members:

Mary W. Hall , Chair **March 19, 2015**
Date Approved

Samuel Williams , Member **March 19, 2015**
Date Approved

Rajeev Balasubramonian , Member **March 19, 2015**
Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

To address the need of understanding and optimizing the performance of complex applications and achieving sustained application performance across different architectures, we need performance models and tools that could quantify the theoretical performance and the resultant gap between theoretical and observed performance. This thesis proposes a benchmark-driven Roofline Model Toolkit to provide theoretical and achievable performance, and their resultant gap for multicore, manycore, and accelerated architectures.

Roofline micro benchmarks are specialized to quantify the behavior of different architectural features. Compared to previous work on performance characterization, these micro benchmarks focus on capturing the performance of each level of the memory hierarchy, along with thread-level parallelism(TLP), instruction-level parallelism(ILP), and explicit Single Instruction, Multiple Data(SIMD) parallelism, measured in the context of the compilers and runtime environment on the target architecture. We also developed benchmarks to explore detailed memory subsystems behaviors and evaluate parallelization overhead. Beyond on-chip performance, we measure sustained Peripheral Component Interconnect Express(PCIe) throughput with four Graphics Processing Unit(GPU) memory managed mechanisms.

By combining results from the architecture characterization with the Roofline Model based solely on architectural specification, this work offers insights for performance prediction of current and future architectures and their software systems. To that end, we instrument three applications and plot their resultant performance on the corresponding Roofline Model when run on a Blue Gene/Q architecture.

To those who had given me dreams to look forward to.

CONTENTS

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I want to thank my thesis advisor, Mary Hall, for her valuable guidance, patience, and long-term career and life advice. She is the one who had given me dreams to look forward to. Without her, the dream of further contributing my knowledge to the field of Computer Science would not be realized.

Second, I'd like to thank Samuel Williams, my idol in the field of Computer Science, for his valuable guidance and patience. Without him, this thesis work would not be possible.

I want to extend my gratitude to my committee member, Rajeev Balasubramonian, for agreeing to sit on my thesis committee and giving me valuable feedback to polish my thesis work.

I'd like to thank scientists from Lawrence Berkeley National Laboratory — Leonid Oliker, Terry Ligocki, Brian Van Straalen, and Matthew Cordery, for their valuable collaboration.

Finally, I'd like to thank my parents and my brother, Jesse, for their love and support.

Authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Partial support for this work was provided through the Scientific Discovery through Advanced Computing (SciDAC) program funded by the U.S. Department of Energy Office of Advanced Scientific Computing Research under award number DE-SC0006947. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231. This research used resources of the Argonne Leadership Computing Facility, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725

CHAPTER 1

INTRODUCTION

The growing complexity of high-performance computing architectures makes it difficult for users to achieve sustained application performance across different architectures. Worse, quantifying the theoretical performance and the resultant gap between theoretical and observed performance is becoming increasingly difficult. Without automating performance data acquisition, performance modeling will continue to require extensive involvement by one of very few performance experts. As such, performance models and tools that facilitate this process are crucial. Such performance models need not be complicated, but should be practical and intuitive. A model should provide upper and lower bounds on performance for a given computation on a particular target architecture and be suggestive of where optimization would be profitable. Additionally, the model should provide an indication of the fundamental bottlenecks and inherent challenges associated with improving a specific kernel's performance on the target architecture.

An exemplar of such modeling capability is the Roofline Model [37, 36, 3]. The Roofline Model combines arithmetic intensity, memory performance, and floating-point performance together into a two-dimensional graph using bound and bottleneck analysis. However, it is time consuming and difficult and requires a human to manually analyze an architecture to determine the machine characteristics needed for populating the Roofline Model. Worse, even if the machine characteristics can be manually estimated, the real issue is achievable performance. These theoretical maximums give a developer no guidance as to what optimization is necessary to achieve maximum performance. As such, we wish to automate that process.

This thesis presents a benchmark-driven Roofline Model Toolkit to automatically and empirically determine the machine characteristics that are needed to generate the Roofline graph and do Roofline analysis. The core of the Roofline Model Toolkit is a processor architecture characterization engine, and a collection of portable instrumented micro benchmarks. These micro benchmarks are implemented with a Message Passing Interface(MPI) to decompose a domain across multiple processors, and OpenMP to express thread-level parallelism. In addition, this thesis extends the Roofline formalism to address the emerging challenges associated with accelerators such as Graphics Processing Units(GPUs). To that end, we constructed five benchmarks designed to drive empirical Roofline-based analysis. The first two present the conventional memory hierarchy bandwidth and floating-point computation aspects of the Roofline Model. The third estimates the parallelization overheads. The fourth benchmark is a novel and visually intuitive approach to quantify the performance relationship between spatial and temporal locality on a CPU or GPU. The last benchmark is designed to quantify the effective Peripheral Component Interconnect Express(PCIe) bandwidth and effects of different caching strategies on a GPU. We will evaluate these benchmarks on four platforms — Edison (Intel Ivy Bridge CPU), Mira (IBM Blue Gene/Q), Babbage (coprocessor only, Intel MIC Knights Corner), and Titan (GPU only, Nvidia Tesla K20x), and use the resultant Empirical Roofline Model to analyze three HPC benchmarks — HPGMG-FV, GTC, and miniDFT.

1.1 Roofline Performance Model

Roofline is an insightful visual performance model that combines arithmetic intensity, memory performance, and floating-point performance into a two-dimensional graph. This graph bounds the performance of individual algorithms or entire applications running on multicore, manycore, or accelerated architectures. Figure 1.1 represents a conventional Roofline Model. In this section, we define the performance metrics that are used to form a Roofline Model.

1.1.1 Performance Metrics

Performance metrics measure the different aspects of an algorithm or application performance. One can quantify the communication performance by defining the measurement of the data transfer rate as bytes transferred per second (B/s) or billions of bytes per second (GB/s). One can also quantify the computational performance by counting the number of operations. A common performance metric used to determine computational performance is floating-point operations per second (FLOPs) or billions of floating-point operations per second (GFLOPs).

1.1.2 Arithmetic Intensity

Arithmetic Intensity is defined as the ratio of the number of floating-point operations performed to the number of bytes per memory transferred. Take a simple triad as an

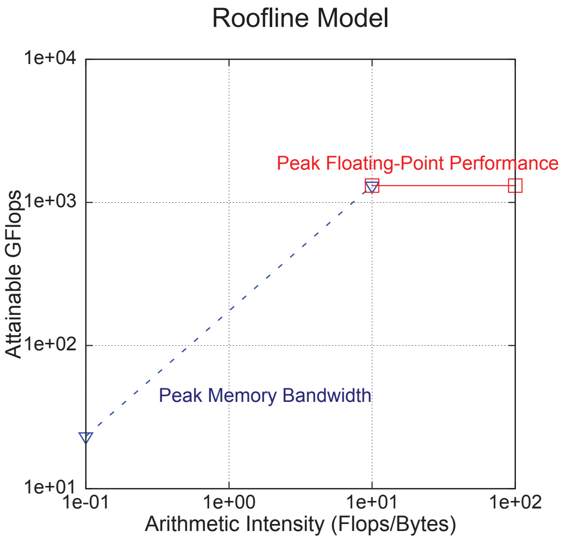


Figure 1.1. A conventional Roofline Model: the two lines intersect at the machine balance.

example. A triad kernel (a[i] = b[i] + scalar * c[i]) requires arithmetic intensity of 0.0625 (2N Flops / 32N bytes) which is lower than the conventional wisdom design point of 1 Flop per byte. One could estimate the theoretical arithmetic intensity of a certain application by hand or by automatic and static analysis at compile time. Another approach is to use performance counters to acquire the performance of the application on a given machine at runtime. Runtime data collection is the easiest way to calculate total memory traffic. However, the limitations of performance counters might not give us detailed and accurate runtime data. Figure 1.2 presents the Arithmetic Intensity of HPC kernels.

1.1.3 Basic Roofline Model

Equation 1.1 bounds attainable architectural performance. Figure 1.1 plots Equation 1.1 on a log10-log10 scale.

$$GFlops = min \begin{cases} \text{Peak GFlops} \\ \text{Memory Bandwidth} \times \text{Arithmetic Intensity} \end{cases} \tag{1.1}$$

This formula drives the two performance limits — peak floating-point performance, and peak memory bandwidth. The two ceilings could be used to determine application performance characteristics and to guide which optimization strategies to perform. Beyond estimating application performance bottlenecks, this model could be used to evaluate the parallelism and cache locality issues within a single Roofline Model.

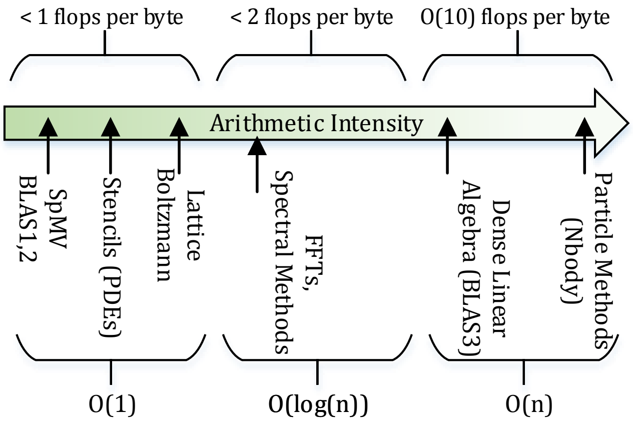


Figure 1.2. Arithmetic Intensity in HPC.

1.2 Empirical Roofline Toolkit Overview

The Empirical Roofline Toolkit is a downloadable set of tools that anyone can use to help application development and optimization by getting machine and application performance/characteristics, and a completed Roofline Model graph to do Roofline analysis (a beta-release of the Empirical Roofline Tool, ERT, is in Lawrence Berkeley National Laboratory [19]). A high-level description of toolkit structure and functional components is shown in Figure 1.3. The three primary parts of this toolkit are described here.

- **Architecture Characterization Engine:**

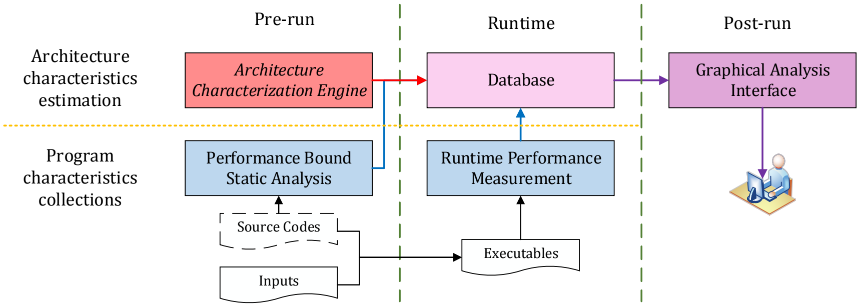


Figure 1.3. Overview of the Empirical Roofline Toolkit.

The architecture characterization engine, a collection of portable instrumented micro benchmarks running on the target architecture, can be used to determine machine characteristics and performance. Each micro benchmark is specialized to highlight the multicore, manycore, and accelerated architecture features. This thesis focuses on this part. Our benchmark-driven Roofline Model Toolkit will be integrated into this engine.

- **Program Performance Analysis:**

There are two primary components for the program performance analysis — performance bound static analysis and runtime performance measurement. The goal here is to describe the position within the Roofline landscape correlated with the program call stack. For performance bound static analysis, there is an existing tool — PBound (a tool based on the ROSE compiler infrastructure), which was created at Argonne [28]. PBound can gather performance-relevant data by examining the source code of an application, and can be used for computing realistic, parameterized performance estimates based on source analysis and transformation.

In combination with empirical performance data obtained as described here and architecture characteristics produced by the architecture characterization engine in part 1, this approach can construct a predictive performance model that can be used to gauge an algorithm's performance given different types of potential future architecture configurations and to gain valuable insight into different aspects of software optimization strategies.

- **Query Database and Graphical User Interface:**

The database query system will be augmented to support all kinds of queries required for the Roofline analysis. For advanced users, rapid evaluation and diagnosis of an application can be done programmatically by using the API to interact with the performance database. These queries can be shared across many applications. There will be a graphical user interface to this query system. The interface can be used to present the resultant performance model graphically and relate performance back to their source code. This will allow analysis artifacts to be stored and retrieved from the database, helping a user evaluate and diagnose application performance on the target machine.

1.3 Thesis Contributions

The following are the primary contributions of the thesis.

- We have created Roofline Benchmarks that can be used to capture the attainable bandwidth and floating-point performance of each level of the memory hierarchy, along with thread-, instruction- and data-level parallelism. We have also created benchmarks to evaluate parallelism, locality issues, and software managed cache technologies in CUDA to provide in-depth characterization of the memory subsystems.

- We have developed an architectural characterization engine. This can be used both to provide the achievable performance and its resultant gap between reality and theory on current architectures, and also to predict performance on potential new architectures, enabling informed algorithm design and implementation.

- Finally, we use the toolkit to benchmark four leading HPC systems: Edison, Mira, Babbage, and Titan. Edison's Empirical Roofline Model, with sufficient parallelism, is very close to its theoretical model. Conversely, we see substantial differences between theory and reality on other architectures.

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 discusses related work. Chapter 3 discusses two basic benchmark designs, experimental setup, and Empirical Roofline Model construction. The benchmark discussion includes our design principle and functionality description. The experimental setup includes the description of four leading HPC platforms as well as details of programming models, compilers, and execution methodology. Chapter 4 extends the basic Empirical Roofline Model by adding performance ceilings as well as describing advanced benchmarks. These benchmarks can be used to evaluate various performance issues and perform sensitivity analysis in which we examine the impact on effective bandwidth and in-core performance. Chapter 5 discusses GPU benchmarking to provide an in-depth understanding of a GPU's memory subsystems. Chapter 6 integrates the performance results from three HPC applications and analyzes them using the Empirical Roofline Model. Finally, Chapter 7 concludes this thesis with a summary of contributions and ideas for possible future work.

CHAPTER 2

RELATED WORK

Performance modeling and benchmarking has put enormous effort in the area of performance analysis and optimization. This research area can be divided into two key subareas — architecture performance analysis and prediction, in which one wishes to understand achievable performance on an existing machine and predict the performance of a future architecture, and application performance analysis and prediction, in which one attempts to predict and analyze the application performance on a target architecture. The work presented in this thesis is focused on understanding a machine's achievable performance, and analyzing an application within the resultant architectural performance model.

The architecture community has developed software simulators to predict performance for potential architectures before a real machine is built. However, simulators only provide theoretical numbers that offer no insights into architecture capability and productivity. To rectify this, one could use accurate performance counters to provide information as to how well the machine and application is performing [29]. However, performance collection from a real machine only provides a flood of data, and performance data accuracy should be verified. Hence, in this thesis, we analyze hardware performance by creating a series of benchmarks to address the need of quantifying achievable performance and reliable performance data.

The following section presents related research on performance modeling and benchmarking, and compares the differences with our approaches.

2.1 Performance Modeling

A performance model can provide a path to understand the past and current trend as well as predict the future trend to show you whether you are trending toward or away from the performance objects. Parallel random-access machine (PRAM), a shared memory abstract machine, is an exemplar of parallel computation models [30, 24]. PRAM can be used to model sequential and parallel algorithmic performance. Algorithm cost is estimated by using two parameters, $O(time)$ and $O(time \times \text{number of processor})$. However, this model neglects memory access latency, synchronization, or communication issues, and cannot reflect the trends underlying parallel computers. To rectify this, Culler, et al. proposed a parallel machine model, LogP [14], for quick prototyping and developing fast, portable parallel algorithms and offering guidelines to architecture designers. LogP is described by four parameters: L (communication latency), o (overhead of sending and receiving message), g (processor to processor bandwidth), and P (the number of processing units). However, LogP is still not simple and intuitive for users. Williams, et al. proposed a higher-level abstraction, the Roofline Model [37, 36, 3]. This visually-intuitive performance model can be used to bound the performance of various numerical methods and operations running on multicore, manycore, or accelerated architectures. This thesis is based on the Roofline Model.

2.2 Performance Analysis Tool

Performance modeling cannot be accurate without actual performance statistics. To measure and analyze program performance, one could use HPCToolkit that collects accurate measurements of a program's work and resource consumption [22]. Alternatively, one could choose Tuning and Analysis Utilities (TAU) that is capable of gathering performance information through instrumentation of functions, statements, and event-based sampling [34]. Both support access to hardware performance counters using PAPI (Performance Application Programming Interface [29]). One can also choose Vampir to diagnose the application performance [35]. Vampir is able to collect aggregated performance information on total processors and threads or per process/thread. This can also be used to profile and trace an application. These tools help to highlight inefficient parts of code. User may find these tools are useful to understand their program performance.

Although these tools can identify which part of an application is inefficient, they cannot provide insights into different aspects of software optimization strategies. As such, combined with the bounds gained from the theoretical Roofline Model, this thesis proposes a benchmark-driven Roofline Model Toolkit to determine machine characteristics and address the need of understanding application performance on target architectures.

2.3 Benchmarking

STREAM (included in the HPC Challenge Benchmark (HPCC)) is a simple and popular benchmark that is used to measure data movement [31]. Typically, it measures the data movement between DRAM and the cache hierarchy and is structured as a streaming memory accesses. The STREAM benchmark has become the de-facto solution for benchmarking the ultimate DRAM bandwidth of a multicore processor. STREAM uses OpenMP threads and will perform a series of experiments designed to quantify the memory subsystem's performance as a function of common array operations, such as copy, add, and triad. Unfortunately, all these operations write to the destination array without reading it. As such, the hidden data movement necessitated by a write-allocate operation effectively impedes the bandwidth. Today's instruction set architectures (ISA) often provide a means of bypassing this write allocate operation. Unfortunately, it is rare for a compiler to generate this operation appropriately on real applications. As such, we are motivated to augment stream with read-only (sum or dot product) or read-modify-write (increment) benchmarks in order to cleanly quantify this hidden data movement.

When DRAM bandwidth no longer dominates the computation, cache bandwidth often becomes the bottleneck to on-node application performance. CacheBench part of LLCBench (Low Level Architectural Characterization Benchmark Suite) can be used to understand the capacities and bandwidths of the cache hierarchy [26]. Unfortunately, CacheBench is not threaded with OpenMP or parallelized with MPI. As such, it cannot measure contention at any level of the cache hierarchy (including DRAM like STREAM), nor can it provide the attainable machine limit at each level of memory hierarchy. Rather than taking this purely empirical approach, one can, with sufficient documentation, create an analytical model of the cache hierarchy using the Execution Cache Memory model [21]. This analytic model is based on the assumptions that single core execution time is composed of in-core execution and data transfers in the memory hierarchy. Then it scales single-thread performance by adding cores until the first bottleneck is reached to model latency effect.

Both the STREAM and LLCBench benchmarks are structered as pure data streaming accesses and hardware stream prefetchers can often speculate the cache line to hide memory latency. The performance of these prefetchers is highly dependent on architectures. It has been observed that bandwidth is highly correlated with the number of elements accessed contiguously [25]. Short "stanzas" of memory access see substantially degraded performance. Stanza Triad was created to quantify this effect [15]. Unfortunately, it is not threaded. As such, it cannot identify when one has transitioned from a concurrency-limited regime to a throughput-limited regime when running on multicore processors.

To provide a more in-depth understanding of memory subsystem performance, several benchmarks for measuring the memory performance of HPC systems along dimensions of spatial and temporal locality have been proposed. Apex-MAP is a synthetic benchmark that directly tackles the locality concept by performing data movement operations in accordance with parameterized degrees of spatial and temporal locality [32, 33, 1]. The guiding principle in the design of Apex-MAP is to characterize the performance behavior of a scientific application by using a small set of performance factors — regularity, data set size, spatial locality, and temporal locality. However, this benchmark does not consider the influence of synchronization in an application.

Synchronization can be the key factor of overhead in all parallel architectures. The EPCC OpenMP micro benchmark suite is perhaps the de-facto solution for benchmarking OpenMP overheads [18, 5, 6]. However, the OpenMP barrier time estimation does not accurately eliminate the effect of creating an OpenMP parallel region, and EPCC is only designed to quantify the OpenMP parallelism issues in a CPU. As such, we are motivated to design a parallelization overhead benchmark that could quantify the synchronization and parallel region launching time in multicore, manycore, and accelerated architectures. One can also use the CLOMP benchmark to characterize the aspect of OpenMP implementations [4]. CLOMP complements the EPCC benchmark suite to provide simple measurements of OpenMP overheads in the context of real application scenarios.

Accelerated architectures have been typically used as an accelerator with dedicated memory attached to a conventional system through a PCIe or similar bus. The performance might be dominated by the data transfer between GPU and CPU over the PCIe. One could measure the PCIe bandwidth by using the OpenCL PCIe bandwidth test, or other scientific applications [7]. However, these benchmarks do not analyze the importance of locality on accelerated architectures like GPUs, and do not quantify the benefits of the emerging software managed cache technologies in CUDA.

Perhaps the most similar work to ours is encapsulated in the benchmarks used to drive the Energy Roofline Model [8]. In that work, a series of experiments were constructed that vary arithmetic intensity to understand the architectural response in terms of both performance and power. When combined with a cache benchmark, one can infer the energy requirements for various computational and data movement operations. Whereas their goal was focused heavily on power and energy, we are focused on performance.

CHAPTER 3

EMPIRICAL ROOFLINE MODEL

In this chapter, we create bandwidth and floating-point benchmarks as primary motivators to construct a basic Empirical Roofline Performance Model. Section 3.1 and Section 3.2 discuss the bandwidth benchmark and floating-point benchmark design philosophy. Then we discuss the four leading HPC platforms evaluated in this thesis. Benchmarking results and analysis are presented in Section 3.4. Then Section 3.5 synthesizes bandwidth and floating-point performance results into a two-dimensional performance graph using bound and bottleneck analysis — a basic Empirical Roofline Performance Model. Finally, we summarize the chapter in Section 3.6.

3.1 Memory and Cache Bandwidth

The peak performance and the throughput of a processor have increased significantly over the last couple decades. However, bandwidth and data movement are still the paramount performance bottleneck on scientific applications. Unfortunately, as discussed in the related work, most existing benchmarks fail to proxy the contention, locality, or execution environment associated with real applications. To rectify this, we have created a Roofline Bandwidth Benchmark that uses a hybrid MPI+OpenMP model to explore bandwidths across all scales. We also created a CUDA bandwidth benchmark to address the need of getting achievable bandwidth in the sophisticated GPU's memory subsystem.

3.1.1 Bandwidth Code

We use a hybrid MPI+OpenMP model to create this benchmark. Thus, developers could proxy a flat MPI model or run in hybrid mode to understand the performance on NUMA architectures. Rather than using the work-share construct, the Roofline Bandwidth driver (Fig. 3.1 (bottom right)) creates a single parallel region and statically assigns threads to a range of array indices. All initialization, synchronization, and computation take place within this parallel region.

Like CacheBench, the bandwidth computation kernel (Fig. 3.1 (upper right)) is designed to quantify the available bandwidth at each level of the memory hierarchy using a simple unit-stride streaming memory access pattern. However, unlike CacheBench, it includes the effects of contention arising from thread parallelism and finite Network-on-Chip(NoC) bandwidth. In that regime, it is similar to STREAM code (Fig. 3.1 (left)) which uses the OpenMP work-share constructs to split loop iterations across multiple threads. Unfortunately, STREAM code writes to the destination array without reading it. As such, the hidden data movement necessitated by a write-allocate operation effectively impedes the performance. To that end, we used a simple array increment kernel that can read and write to the same destination.

The benchmark may thus be used to quantify the capacity of each level of the memory hierarchy as well as the bandwidths between levels. Moreover, by adjusting the parameters, one can estimate the overhead for an MPI or OpenMP barrier. As the benchmark is MPI+OpenMP, one can explore these bandwidths and overheads across all scales.

```
void STREAM(TYPE scalar){
  ssize_t j;
  #pragma omp parallel for
  for (j = 0; j <SIZE; j++)
    B[j] = scalar * A[j];
}



int main() {
  scalar = 3.0;
  for (k = 0; k < TIMES; k++) {
    // start timer here
    STREAM(scalar);
    // stop timer here
  }
}
```

```
void KERNEL(uint64_t size, uint64_t trials,
            double * __restrict__ A){
  double alpha = 0.5;
  uint64_t i, j;
  for (j = 0; j < trials; ++j) {
    for (i = 0; i < size; ++i) {
      A[i] = A[i] + alpha;
    }
    alpha = alpha * (1-1e-8);
}}

int main() {
  ...
  #pragma omp parallel private(id)
  {
    uint64_t n, t;
    initialize(&A[nid]);
    for (n = 16; n < SIZE; n *= 1.1) {
      for (t = 1; t < TRIALS; t *= 2) {
        // start timer here
        KERNEL(n, t, &A[nid]);
        // stop timer here
        #pragma omp barrier
        #pragma omp master
        {
        MPI_Barrier(MPI_COMM_WORLD);
        }
        double bytes = 2 * sizeof(double) *
                       (double)n * (double)t;
}}}}
```

Figure 3.1. Bandwidth code comparison. (left) STREAM facsimile. (right) Roofline Bandwidth Benchmark.

3.1.2 Specialized CUDA Bandwidth Kernel

For many developers, the most difficult part of creating a high-performance application that leverages GPUs is managing the bandwidth bottleneck of different types of memory. To help programmers better understand the GPU's memory subsystem, we have created a CUDA's bandwidth benchmark to provide the achievable bandwidth of different types of GPU memory.

We found it illustrative to run three slightly different kernels designed to quantify the effects of explicit and implicit reuse within the GPUs memory hierarchy. Both Kernel `global_trialInside` (`global_tInside` on Fig. 3.2) and Kernel `global_trialOutside` (`global_tOutside`) use global memory, but with the trials loop inside and outside the CUDA kernel call, respectively. Kernel `sharedMem` copies global memory data to shared memory, performs a trials loop inside the kernel, and copies back to global memory. The CUDA kernels block-stride loop over a one-dimensional array to guarantee memory coalescing.

3.2 Floating-Point Compute Capability

Although many applications are limited by memory bandwidth, on-chip computation and in-core performance are also important aspects of performance on scientific applications. The common features of high-performance architectures are deep pipelines, significant instruction-level and data-level parallelism, and sophisticated branch prediction schemes. Compilers claim to be capable of performing low-level optimizations on these architectures. Hence, we are motivated to create a floating-point benchmark to attain the flops limit with

```
dim3 gpuThreads(64);
dim3 gpuBlocks(224);

#if defined (GLOBAL_TRIAL_INSIDE)
  global_trialInside <<<gpuBlocks, gpuThreads>>> (nsize, trials, d_buf);
#elif defined(GLOBAL_TRIAL_OUTSIDE)
  for (uint64_t t = 0; t < trials; ++t) {
    global_trialOutside <<<gpuBlocks, gpuThreads>>> (nsize, d_buf, alpha);
    alpha = alpha * (1  1e-8);
  }
#else
  sharedMem <<<gpuBlocks, gpuThreads>>> (nsize, trials, d_buf);
#endif
```

Figure 3.2. CUDA bandwidth computation kernel.

compilers' aggressive optimization.

The floating-point computation kernel modified from the bandwidth benchmark uses a processor macro to vary the degree of the polynomial. Doing so allows one to change the balance between loads/stores and floating-point operations from L1-limited to flops-limited. Figure 3.3 presents an example of this benchmark. As one can see, the degree of parallelism per thread in this routine is O(nsize). An in-order processor would deliver performance limited by the floating-point latency if the compiler failed to sufficiently unroll and reorder. Hence, a compiler could unroll this loop to hide floating-point latency and express instruction-level parallelism to avoid bubbles in the pipeline and/or SIMDize the unrolled code to exploit data-level parallelism. Alternately, an out-of-order processor, with a sufficiently deep reorder buffer, could find the inherent instruction-level parallelism and attain high performance.

3.3 Experimental Setup

The diversity of existing and emerging hardware and programming models makes construction of generalized benchmarks particularly difficult. This section provides the background materials on four leading HPC platforms and programming models used throughout this work. In Section 3.3.1, we discuss the four fundamentally different architectures — a conventional superscalar out-of-order Intel Xeon multicore processor (Edison), a low-power dual-issue in-order IBM Blue Gene/Q multicore processor (Mira), a high-performance in-order Intel Xeon Phi manycore processor (Babbage), and a high-performance NVIDIA Kepler K20x GPU accelerated system (Titan) used for benchmarking. Section 3.3.2 and

```
void KERNEL(uint64_t size, uint64_t trials, double * __restrict__ A) {
  for (j = 0; j < trials; ++j) {
    for ( i = 0; i < nsize; ++i) {
      double beta = 0.8;
      #if FLOPPERITER == 2
        beta = beta * A[i] + alpha;
      #elif FLOPPERITER == 4
        beta = beta * A[i] + alpha;
        beta = beta * A[i] + alpha;
      #elif FLOPPERITER == 8
      ...
      #endif
      A[i] = beta;
    }
    alpha = alpha * (1 - 1e-8);
}
```

Figure 3.3. Roofline Floating-Point Benchmark

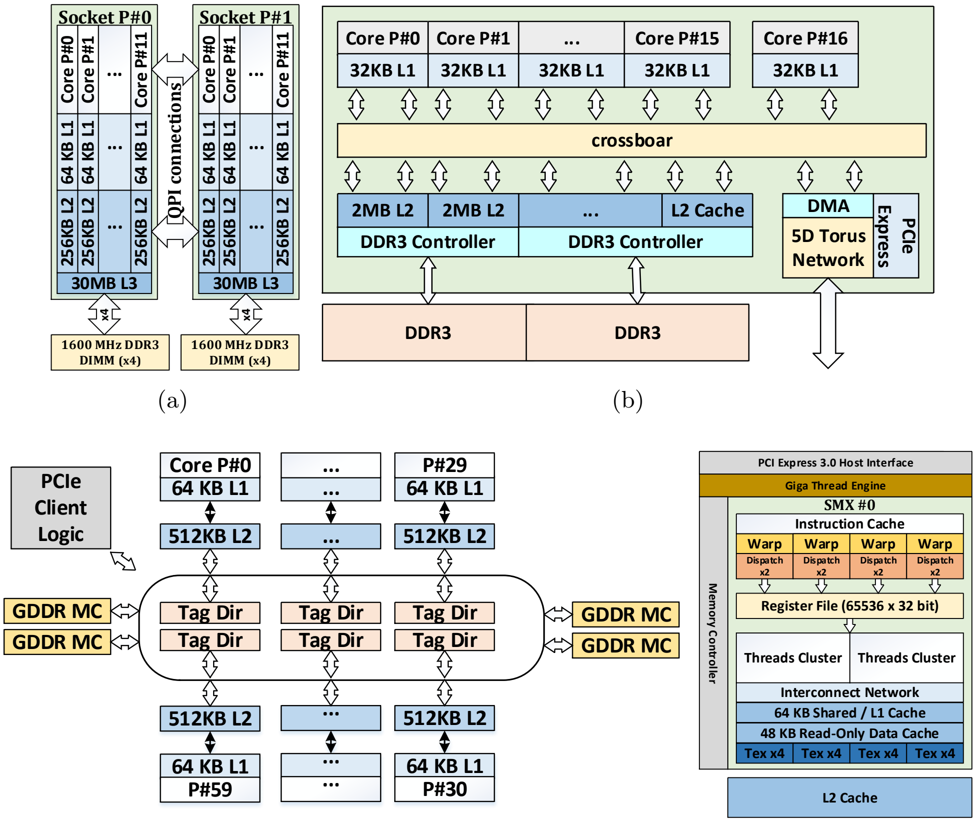
Section 3.3.3 provide some background on programming models and compilation options, and execution on our selected platforms.

3.3.1 Architectural Platforms

In this thesis, we used four HPC platforms as a testbed for our benchmarking experiments. We believe that benchmarking these four leading HPC systems can help the scientific community that uses these systems throughout their work. Figure 3.4 illustrates the memory subsystems for the four platforms.

- **Edison (Intel Ivy Bridge Multicore Processor):**

Edison is an MPP at NERSC [17]. Each node includes two 12-core Xeon E5 2695-V2 processors nominally clocked at 2.4GHz (TurboBoost can increase this substantially). Each core is a superscalar, out-of-order, 2-way HyperThreaded core capable of performing two



(a)

(b)

(c)

(d)

Figure 3.4. Architecture layouts for four platforms. (a) Edison. (b) Mira. (c) Babbage (MIC only). (d) Titan (GPU only).

four-way AVX SIMD instructions (add and multiply) per cycle in addition to loads and stores. Each core has a private 32KB L1 data cache and a private 256KB L2 cache. The 12 cores on a chip share a 30MB L3 cache and a memory controller connected to four DDR3-1600 DIMMs. The 2 sockets are connected by quick path interconnections. Extensive stream prefetchers are designed to saturate bandwidth at each level of the cache hierarchy. Theoretically, the superscalar and out-of-order nature of this processor should reduce the need for optimized software and compiler optimization. Among the four architectures, Edison is the only out-of-order 2-socket processor and delivers the highest clock rate.

- **Mira (IBM Blue Gene/Q Multicore Processor):**

Mira is an IBM Blue Gene/Q system installed at Argonne National Lab [11]. Each node includes one 16-core BGQ SOC. Each of the 16 A2 cores is a four-way SMT dual-issue in-order core capable of performing one ALU/Load/Store instruction and one four-way FMA per cycle. However, to attain this throughput rate, one must run at least two threads per core. Each core has a private 16KB data cache and the 16 cores share a 32MB L2 cache connected by a crossbar. Ideally, the SMT nature of this architecture should hide much of the effects of large instruction and cache latencies. However, the dual-issue nature of the processor can impede performance when integer instructions are a significant fraction of the dynamic instruction mix.

- **Babbage (Intel Xeon Phi Knights Corner (KNC) Manycore coprocessor):**

Babbage is a Knights Corner (KNC) Manycore Integrated Core (MIC) testbed at NERSC [2, 10]. The KNC processor includes 60 dual-issue in-order four-way HyperThreaded cores. Each core includes a 32KB L1 data cache, a 512KB L2 cache, and an 8-way vector unit. L2 caches are conneted by a bidirectional ring interconnect. Although L2 caches are coherent, the ring NoC topology coupled with the coherency mechanism may impede performance. Unlike the aforementioned multicore processors, this manycore processor uses very high-speed GDDR memory which provides a theoretical pin bandwidth of over 350GB/s. To proxy the future Knights Landing (KNL) MIC processor that will form the heart of the NERSC8 Supercomputer Cori [9], we conduct all experiments in “native” mode. As such, the host processor, the host memory, and the PCIe connection are not exercised.

- **Titan (Nvidia K20 GPU):**

Titan is a Cray accelerated MPP system at the Oak Ridge National Lab. Each node includes a 16-core AMD Interlagos CPU processor and one NVIDIA K20x GPU [12]. Each

GPU includes 14 streaming multiprocessors (SMX) each of which can schedule 256 32-thread warps and issue them four at a time to their 192 CUDA cores. Each SMX has a 256KB register file, a 64KB SRAM that can be partitioned into L1 cache, and shared memory (scratchpad) segments. Each chip includes a 1.5MB L2 cache shared among the SMX and is connected to high-speed GDDR5 memory with a pin bandwidth of 232GB/s. Unfortunately, software on the production system Titan tends to lag behind NVIDIA releases. As such, we used a similar K20xm within the Dirac testbed at NERSC [16] to evaluate the CUDA unified virtual address and Unified (managed) Memory. For our purposes, the K20x and K20xm GPUs are identical.

Table 3.1 summarizes the key architectural characteristics of these platforms. Please note that the peak GFlop/s and bandwidths shown are theoretical.

3.3.2 Programming Models and Compilation

In this section, we provide the compiler flags that were used on different platforms (Table 3.2). Note that the table only provides the best performing compiler and compiler options for each architecture in our benchmark experiments. Nominally, all our codes are (MPI+)OpenMP or (MPI+)CUDA. Although for the most part compilation is straightforward, there are some variations across the three compilers.

First, Edison and Babbage both use the Intel C compiler. However, as MIC is run in native mode, it requires the "-mmic" option while Edison is compiled with "-xAVX". The Intel and IBM compilers enable OpenMP differently. On the Intel platforms, one uses "-openmp" while on XL/C, one uses "-qsmp=omp:noauto". To instruct the compilers that there is no aliasing, we use the "-fno-fnalias" and "-qalias=ansi:allptrs" flags on the Intel and IBM compilers, respectively. Finally, it should be noted that depending on the

Table 3.1. Architectural characteristics of four evaluation platforms. ^{1}One GPU per node. ^{2}CUDA cores. ^{3} without TurboBoost.

Platform	Edison	Mira	Babbage	Titan
MPU	Intel Xeon E5-2695v2	IBM BGQ	Xeon Phi KNC	Nvidia K20x
Clock rate (GHz)	2.4	1.6	1.053	0.732
Processors per Node	2	1	1	1^1
Cores per Processor	12	16	60	2688^2
Total Threads	48	64	240	28672
Peak GFlops	460.8^3	204.8	1011	1310
L1 Bandwidth (GB/s)	1843	819.2	4043	1310
DRAM Pin Bandwidth (GB/s)	102.6	42.66	352	232.46

Table 3.2. Compilation flags for each platform.

Platform	**Compiler**	**Flags**
Edison	Intel C	-O3 -xAVX -openmp -fno-alias -fno-fnalias
Mira	IBM XL/C	-O5 -qsimd=auto -qalias=ansi:allptrs -qsmp=omp:noauto
Babbage	Intel C	-O3 -mmic -fno-alias -fno-fnalias -liomp5
Titan	Nvidia CC	-O3 -arch=sm_35 -lcudart

benchmark and platform, we either use CUDA 5 (Titan) or CUDA 6 (Dirac). The NVIDIA compiler requires that one specify the "-arch=sm_35 " flag to build the benchmark for the K20x series.

3.3.3 Benchmark Execution

Unlike simple desktop systems, the MPP supercomputers at NERSC, ALCF, and OLCF might launch jobs from one node and run them on another set of nodes. As such, the benchmark application launch routines vary somewhat from one platform to the next. Table 3.3 shows the relevant options used in our experiments.

On Edison, the Cray system at NERSC, one uses the aprun command to run programs on the compute nodes. To that end, we run the benchmark using two MPI tasks and bind each to one NUMA node with strict memory containment via the "-S 1 -ss -cc numa_node" options. On Mira, we evaluate both a fully threaded and a hybrid mode of 4 processes of 16 threads. We recommend "BG_THREADLAYOUT=1" to balance these threads within cores (scatter affinity) if the total MPI process * OpenMP threads is smaller than 64. On Babbage, which uses the Intel MPI implementation, one uses the "-ppn" option to control the number of MPI tasks per card and the "-n" option to control the total number of MPI tasks. Unlike Edison where aprun controls affinity, one must use the "KMP_AFFINITY" environment variable on Babbage. We set it to "balanced" to distribute threads across the chip if the total MPI process * OpenMP threads is smaller than 240. On Titan, we once again use the aprun options. However, as we do not use the CPU cores, there was no need to control CPU thread affinity or NUMA bindings.

3.4 Benchmarking Results

This section discusses results gained from the two benchmarks running on the four platforms described in Section 3.3.

Table 3.3. Execution mode for each platform.

Platform	Application Execution command
Edison	aprun -n 2 -d 12 -N 2 -S 1 -ss -cc numa_node [benchmark]
Mira	qsub -n 1 –proccount 1 –mode c1 –env BG_SMP_FAST_WAKEUP=YES: BG_THREADLAYOUT=1: OMP_PROC_BIND=TRUE: OMP_NUM_THREADS=64: OMP_WAIT_POLICY=active [benchmark]
Babbage	mpirun.mic -n 1 -ppn 1 [benchmark]
Titan	aprun -n 1 [benchmark]

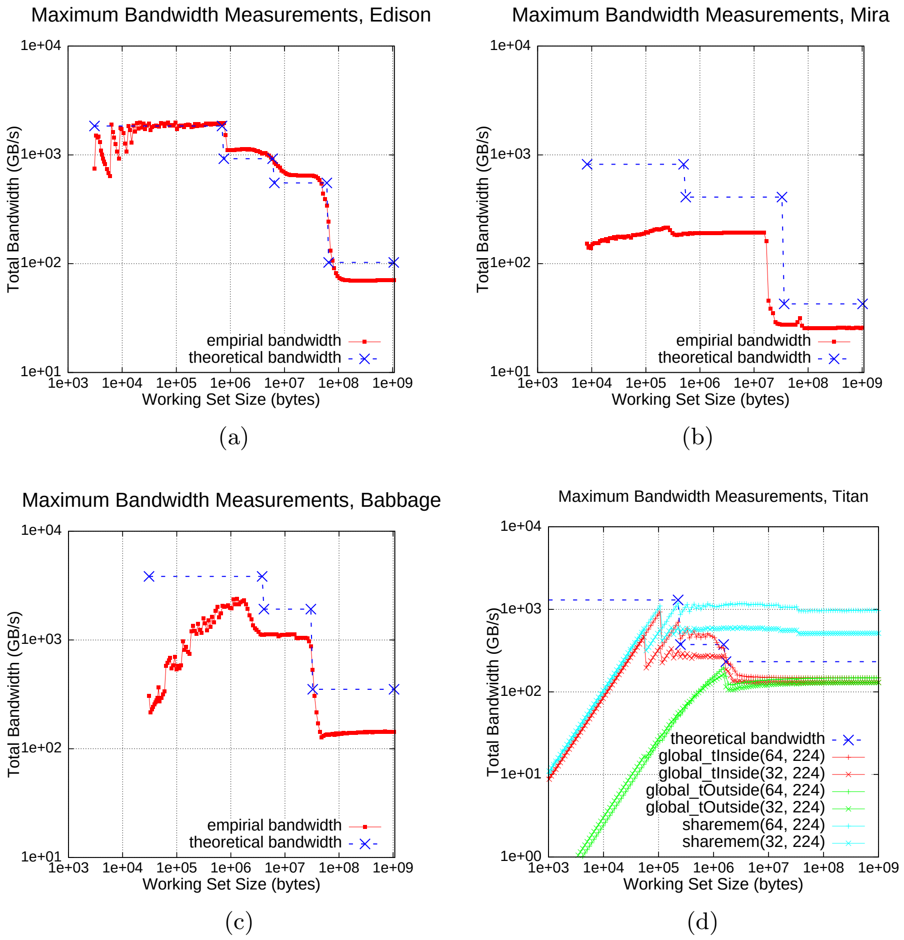
3.4.1 Bandwidth Results

Figure 3.5 presents the results of our Roofline Bandwidth Benchmark running on our four platforms. Note, the x-axis represents the total working set summed across all concurrent units of execution. The blue line marks the theoretical bandwidth and capacities for each level of the memory hierarchy. On the CPU architectures, the red line presents resultant bandwidth.

On Edison, we run two processes per node (launch two MPI tasks), while all other machines run with a single process. We observe that the hardware is capable of achieving its theoretical performance at each level of the cache hierarchy. The only exception is that Edison fails to attain the DRAM pin bandwidth. This is not surprising since few machines can have such high bandwidth and ever attain the pin bandwidth. Moreover, although the read-modify-write memory access pattern performs at higher bandwidth than read-only memory, it may be still suboptimal for this architecture. We can also observe that the transitions are at expected cache capacities. The smooth transitions in bandwidth at cache capacities suggest that the cache replacement policy may not be a true LRU or FIFO but a pseudo variant.

The performance on Mira was consistently below the theoretical bandwidth limits and the transitions seemed to indicate reduced effective cache capacities. The cache replacement policy obtained from BGQ's technical specification is LRU. The particularly surprising low L1 bandwidth may indicate the presence of a write-through or store-through L1 architecture.

On the highly-multithreaded MIC (Babbage), we found it was necessary to operate on working sets exceeding 1 MB (over 4KB per thread) in order to obtain good performance. As the architecture can load 64 bytes per cycle, it is not unreasonable to think 64 loads were necessary to amortize any loop overheads within the benchmark. For smaller working sets, performance was degraded, indicating an underutilization of resources. Generally



(a)

(b)

(c)

(d)

Figure 3.5. Roofline Bandwidth Benchmark results on our four platforms. Please note the log-log scale. On the GPU, the syntax is Kernel(# threads per thread block, # of thread blocks per kernel). (a) Edison. (b) Mira. (c) Babbage (MIC only). (d) Titan (GPU only).

speaking, the benchmark correctly identified the L1 and L2 cache capacities, but the attained bandwidths were far less than the theoretical number. Low L2 bandwidth can be attributed to the lack of an L2 stream prefetcher like on Edison and Mira. If the compiler fails to insert software prefetches perfectly, memory latency will be exposed. Conversely, low DRAM bandwidth is a known issue on this machine and requires hardware solutions to rectify. On the highly-multithreaded MIC (Babbage), we found it was necessary to operate on working sets exceeding 1 MB (over 4KB per thread) in order to obtain good performance. As the architecture can load 64 bytes per cycle, it is not unreasonable to think 64 loads were necessary to amortize any loop overheads within the benchmark. For smaller working sets, performance was degraded, indicating an underutilization of resources.

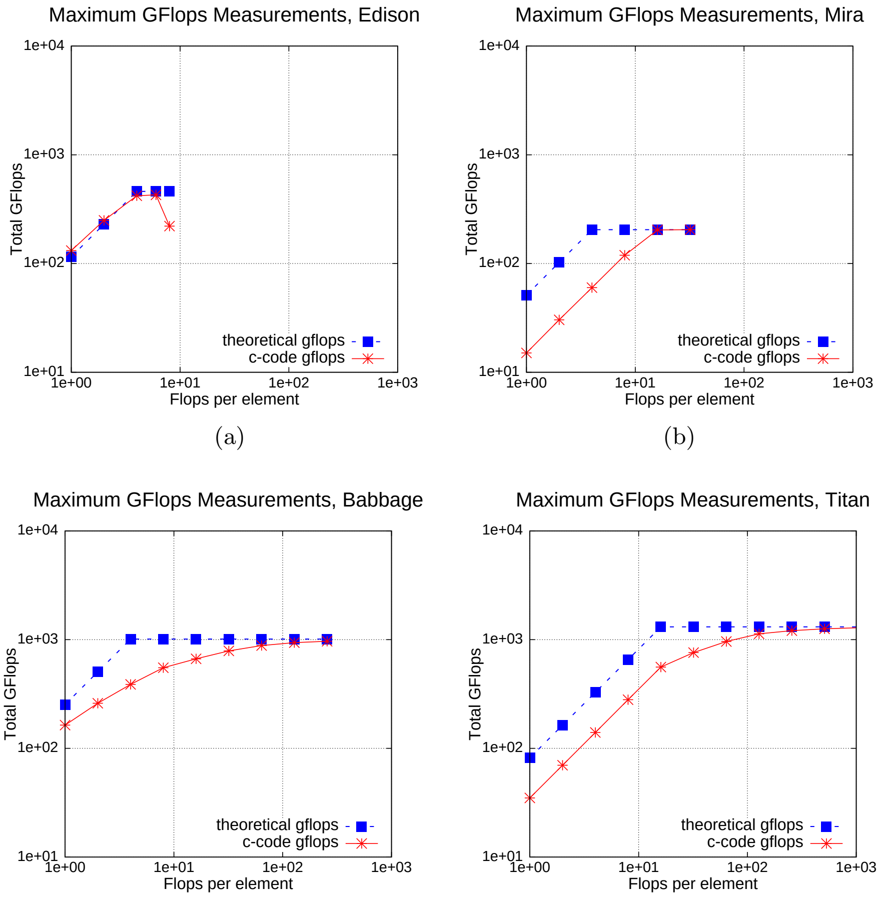
Generally speaking, the benchmark correctly identified the L1 and L2 cache capacities, but the attained bandwidths were far less than the theoretical number. Low L2 bandwidth can be attributed to the lack of an L2 stream prefetcher like on Edison and Mira. If the compiler fails to insert software prefetches perfectly, memory latency will be exposed. Conversely, low DRAM bandwidth is a known issue on this machine and requires hardware solutions to rectify.

On Titan, "global_trialOutside" is perhaps the most similar to the CPU implementations. The entire working set is parallelized across thread blocks and the summation (reuse) occurs at the CUDA kernel level. That is, there is one kernel call per iteration of the geometric sum. We explore performance as a function of the thread block size (32 or 64) with a constant 224 thread blocks. As on Babbage, we see substantial underutilization coupled with large CUDA kernel overheads at small working set sizes, but performance eventually saturates at the DRAM limit, although this is well below the theoretical pin bandwidth. "global_trialInside" restructures the summation loop to increase locality within a thread block and as such, exercises the L1 cache for the per thread-block working set (note, there are 7168 or 14336 threads in all). We see much better performance at the small scale (fewer CUDA kernel calls) and performance can hit the L1 and L2 limits before settling at the DRAM limit. Finally, "sharedMem" restructures the loop once again and exploits shared memory in a blocked manner. As such, it can reach the theoretical performance limit of about 1.3TB/s for shared memory.

Overall, the trends in bandwidth performance on manycore and accelerators are a little disturbing. That is, the only way to get high performance is with massive parallelism on large working sets. For real applications, this observation will make it difficult to use accelerators or manycore processors to solve existing problems faster. Rather, one will be able to run larger problems in comparable time. Nevertheless, this benchmark can be used to help guide programmers as to when it will be viable to migrate to a manycore or accelerated architecture.

3.4.2 Floating-Point Performance

Figure 3.6 presents the floating-point performance as a function of L1 Arithmetic Intensity expressed as Flops per Element (FPE) — essentially the degrees of the FPE. The blue line marks the microarchitecture performance model that takes into account the issue rate of loads/stores compared to floating-point instructions given the mix demanded by the kernel. We run the GFlops benchmark with different degree of FPE until reaching the theoretical peak performance.



(a) (b) (c) (d)

Figure 3.6. Basic GFlops C code compared to theoretical GFlops on four platforms. (a) Edison. (b) Mira. (c) Babbage (MIC only). (d) Titan (GPU only).

Figure 3.6 demonstrates that Edison can quickly reach its peak performance and that performance tracks well with the theoretical model. Generally speaking, at low FPE, performance is diminished due to the fact that the core can perform 8 flops per cycle, but can only sustain loading and storing 2 elements per cycle. Interestingly, the performance of the reference C code falls at high FPE. This is presumably a limit of the reorder buffer and the desire to continually find 5 independent floating-point instructions.

Mira's performance on compiled code is shifted to the right. Generally speaking, this suggests that additional instructions are consuming the same issue slots as loads or stores. On the dual issue A2 architecture, this could very well be integer or branch instructions. This effect was not present on Edison as it is a superscalar processor and can issue integer or branch instructions from ports other than those used for floating-point or load/store.

With sufficient FPE, performance is pegged to peak.

Babbage shows a third behavior — asymptotically approaching peak performance. This behavior suggests that additional instructions (e.g. integer or branch) are consuming the same issue slot as floating-point instructions. As such, performance behaves like $FPE/(FPE+k)$ where k is the number of extra instructions impeding performance.

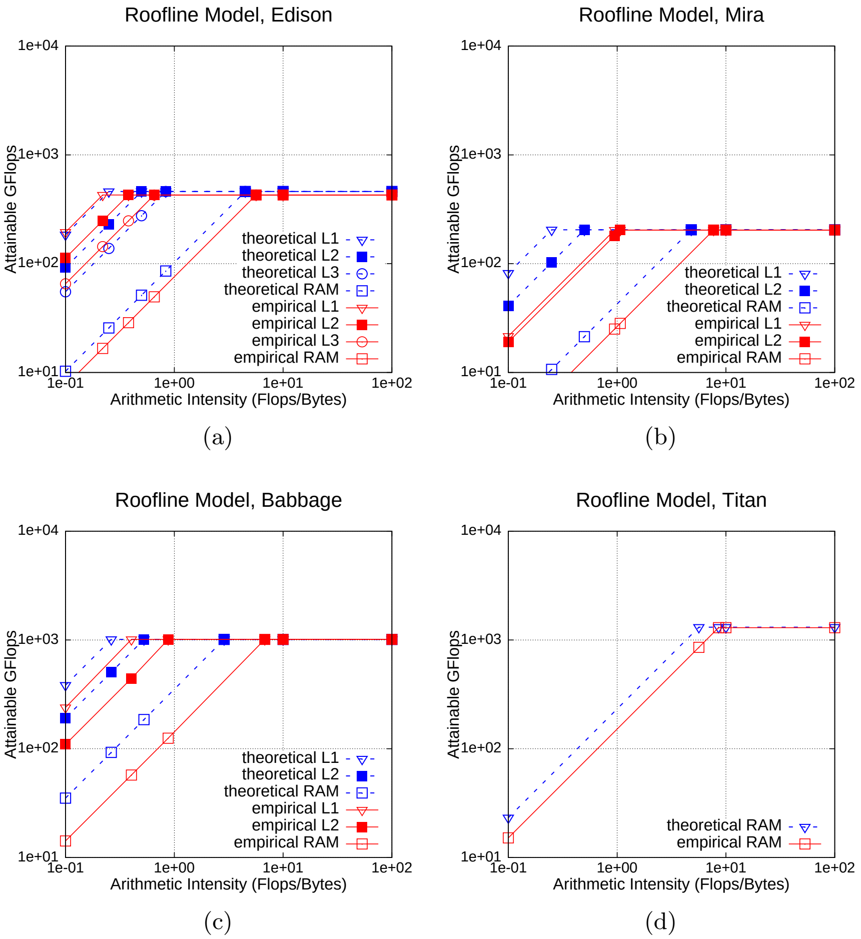
Finally, we constructed a similar CUDA C benchmark to run on the GPU. The theoretical bound is based on the assumption that each load/store unit can sustain loading 4 bytes per cycle (128 bytes per SMX) from memory. We observe that the GPU's performance seems to embody characteristics of both BGQ and MIC. That is, one lacks the issue bandwidth to fully drive the core and the SMX cannot sustain loading/storing 128 bytes per cycle from memory.

3.5 Empirical Roofline Model Construction

Now that we have benchmarked the bandwidth and compute characteristics on each of our four platforms, we may construct Empirical Roofline Models for each. Figure 3.7 shows the resultant models using L1/L2/L3 and DRAM bandwidths as well as the theoretical Roofline Model for each platform. An ideal architecture is one that can fully exploit the technology on which it is built. We see that in general, Edison's empirical performance is very close to its theoretical limits. Conversely, on Mira and Babbage, we see substantial differences between theory and reality. The extreme multithreading paradigm allows the GPU to deliver a high fraction of its theoretical bandwidth when running on the device.

3.6 Summary

In this chapter, we discussed two basic benchmark designs and four leading HPC platforms. As discussed in Section 3.1 and 3.2, our benchmarks are able to proxy contention, locality, and execution environment associated with real applications by using OpenMP+MPI model. The bandwidth benchmark is designed to quantify available bandwidth at each level of memory hierarchy; the floating-point benchmark is designed to find the highest achievable performance by varying the degree of FPE. In Section 3.4, we benchmarked the four leading platforms: Edison, Mira, Babbage, and Titan, that have been described in Section 3.3. Results show that Edison is able to come close to its theoretical performance while we see large overheads at small working set size on Babbage and Titan. Overall, the only way to get high performance on manycore architectures and accelerators is with massive parallelism on large working sets. That is, one can run larger problems in comparable time. These benchmarks can be used to help guide when it will be viable to migrate to a manycore or



(a)

(b)

(c)

(d)

Figure 3.7. Roofline Models for four platforms. (a) Edison. (b) Mira. (c) Babbage (MIC only). (d) Titan (GPU only).

accelerated architecture. Finally, in Section 3.5, we synthesized the results to construct an Empirical Roofline model compared with theoretical Roofline. We see Edison's Empirical Roofline is very close to its theoretical one. However, we see substantial difference between theory and reality on the other architectures.

CHAPTER 4

EMPIRICAL ROOFLINE MODEL: PERFORMANCE CEILINGS

In this chapter, we expand the Empirical Roofline Model by adding bandwidth and computational performance ceilings. We perform various sensitivity analysis in which we satisfy less and less in-core parallelism and memory concurrency by creating advanced benchmarks to provide in-depth knowledge into memory subsystems.

4.1 Architecture Compute Capability

When performance is on the balance between compute-bound and memory-bound performance, proper exploitation of instruction-, data-, and thread-level parallelism can ensure the code is not artificially flop-limited. Unfortunately, there are relatively few benchmarks that accurately measure the importance of these facets of parallelism on modern many-core and accelerated architectures. To address this deficiency, we constructed a SIMDize floating-point benchmark.

4.1.1 SIMDize Floating-Point Benchmark

Figure 4.1 presents the original floating-point kernel and other explicitly optimized implementations. An in-order processor could unroll the loop and SIMDize the code to exploit data-level parallelism and/or express instruction-level parallelism. Conversely, an out-of-order processor could reorder the instruction stream, but it can never automatically SIMDize the instruction stream. Hence, we are motivated to design an optimized floating-point benchmark to quantify the disparity between the performance that can be obtained by the architecture on compiled code and the true performance capability of the architecture. We implemented three explicitly unrolled and SIMDized (via intrinsics) implementations of the floating point benchmarks — AVX, AVX-512, QPX. We use 2 flops per element and unrolling by 8 as examples here.

```
void KERNEL(uint64_t size,
  uint64_t trials, double * __restrict__ A){
  for (j = 0; j < trials; ++j) {
    for ( i = 0; i < nsize; ++i) {
      double beta = 0.8;
      #if FLOPPERITER == 2
        beta = beta * A[i] + alpha;
      #elif FLOPPERITER == 4
        ...
      #endif
      A[i] = beta;
  }
  alpha = alpha * (1 - 1e-8);
}}
```

```
void AVX_KERNEL(uint64_t size,
  uint64_t trials, double * __restrict__ A){
  for (j = 0 ; j < ntrials; ++j) {
    for (i = 0 ; i < nsize ; i += 8) {
      bv1 = _mm256_set1_pd(0.8);
      v1  = _mm256_load_pd(&A[i]);
      bv1 = _mm256_mul_pd(bv1, v1);
      bv1 = _mm256_add_pd(bv1, v1);
      _mm256_store_pd(&A[i], bv1);
      // repeat above operations for A[i+4]
    }
    alpha = alpha * (1e-8);
    av = _mm256_set1_pd(alpha);
}}
```

```
void QPX_KERNEL(uint64_t size,
  uint64_t trials, double * __restrict__ A){
  for (j = 0 ; j < ntrials ; ++j){
    for (i = 0 ; i < nsize ; i += 8){
      bv1 = vec_splats(0.8);
      v1  = vec_ld(0L, &A[i]);
      bv1 = vec_madd(bv1,v1,av);
      vec_st(bv1, 0L, &A[i]);
     // repeat above operations for A[i+4]
    }
    alpha =  alpha * (1e-8);
    vec_splats(alpha);
}}
```

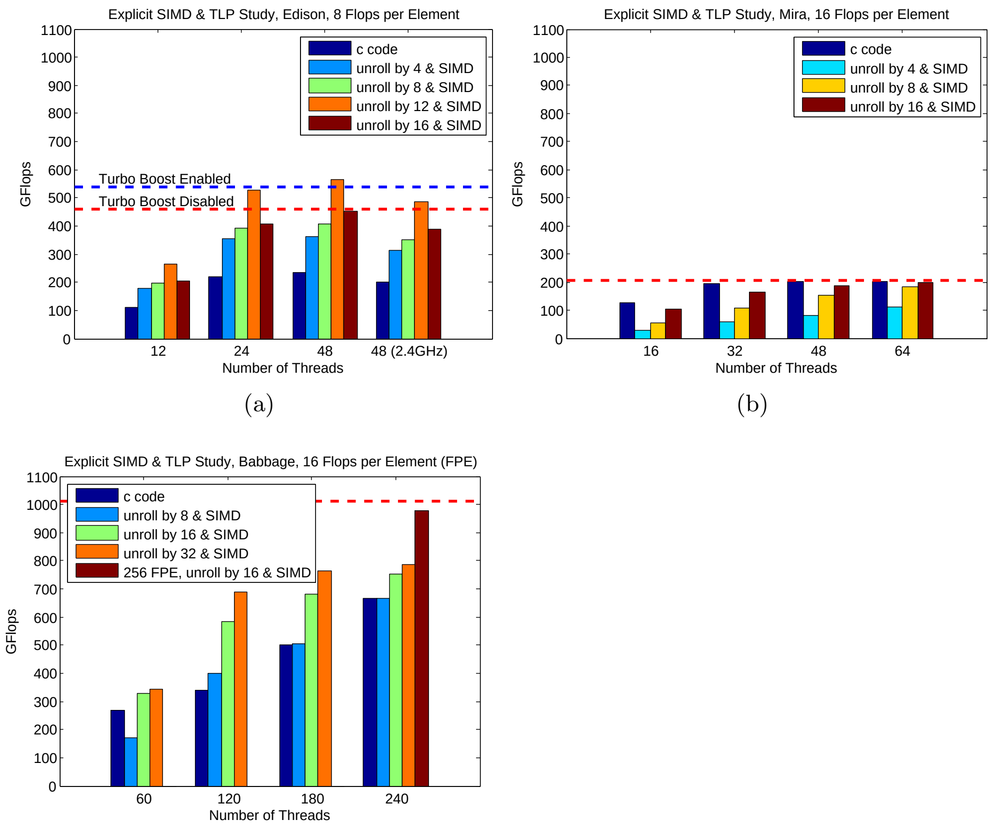
```
void AVX512_KERNEL(uint64_t size,
  uint64_t trials, double * __restrict__ A){
  for (j = 0 ; j < ntrials ; ++j) {
    for (i = 0 ; i < nsize ; i += 8) {
      bv1 = _mm512_set1_pd(0.8);
      v1  = _mm512_load_pd(&A[i]);
      bv1 = _mm512_fmadd_pd(bv1,v1,av);
      _mm512_store_pd(&A[i], bv1);
    }
    alpha = alpha * (1e-8);
    av = _mm512_set1_pd(alpha);
}}
```

Figure 4.1. GFlops code: (up left) Reference C version. (up right) AVX version optimized for Edison. (bottom left) QPX version optimized for Mira. (bottom right) AVX-512 version optimized for Babbage.

4.1.2 Optimization Results

On today's processors, thread- and data-level parallelism must be explicit in the code generated by a compiler. As auto-parallelizing and auto-vectorizing compilers are rarely infallible, these forms of parallelism must often be explicit in the source code as well. Figure 4.2 presents the performance of these implementations on Edison, Mira, and Babbage as a function of thread-level parallelism and unrolling (explicit instruction-level parallelism). Note, each implementation used a different number of flops per element (FPE).

We observe that Edison attains a little less than half the advertised peak with compiled C code. However, when using an optimized implementation, performance improves significantly and can actually exceed the nominal peak performance of 460 GFlop/s. The faster-than-light effect is due to the fact that TurboBoost is enabled on this machine. With a maximum frequency of 2.8GHz with 12 cores, the true peak performance is about 537 GFlop/s — quite close to the observed performance. To verify this, we use the `aprun --p-state` option to peg the frequency at the advertised 2.4GHz and performance is as



(a)

(b)

(c)

Figure 4.2. Performance disparity between compiled code and optimized code in which thread-, instruction-, and data-level parallelism have been made explicit. (a) Edison. (b) Mira. (c) Babbage (MIC only).

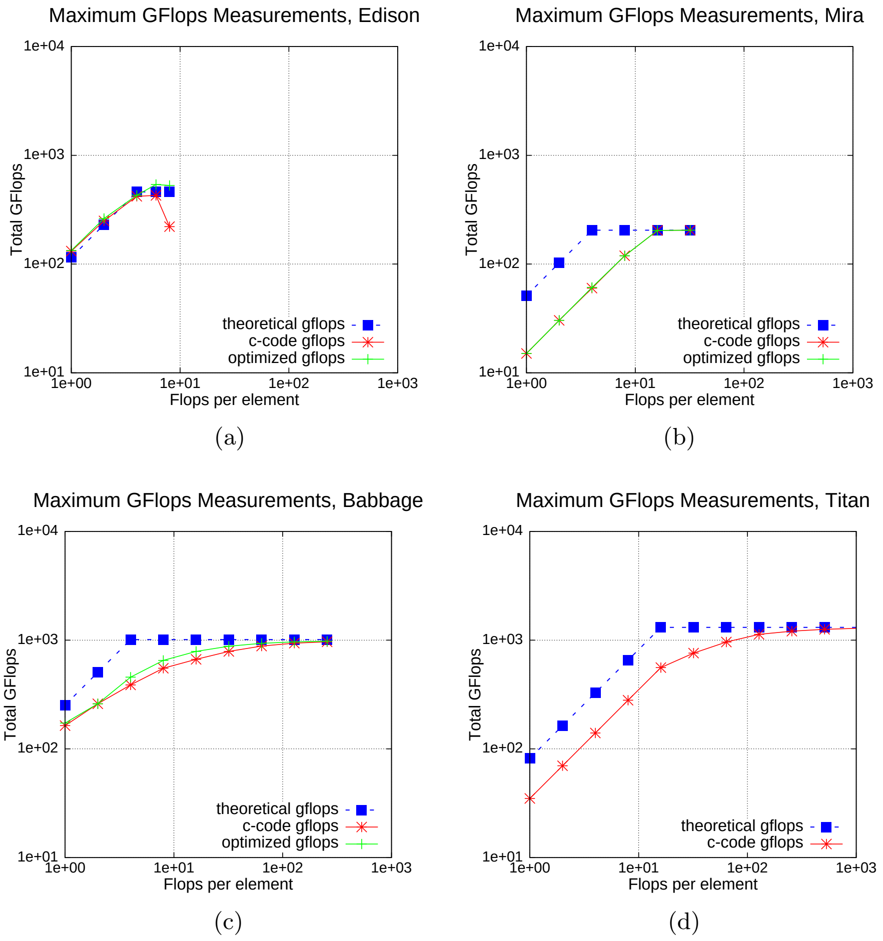
expected. Although the machine is sensitive to instruction-level parallelism (unrolling), it generally does not require HyperThreading to attain good performance.

Running a similar set of experiments on Mira (BGQ), we see a very different outcome. First, compiled code delivers very good performance. This indicates that the XL/C compiler was able to effectively SIMDize and unroll the code sufficiently to hide the floating-point latency. Using explicitly unrolled code, we observe that significant unrolling (2-4 SIMD instructions per thread) is required to reach peak performance. Unlike Edison, Mira clearly requires two threads to attain peak performance.

Finally, Babbage presents a mix of characteristics similar to both Edison and Mira. The compiler clearly fails to make full use of the architecture on even this simple kernel. With sufficient unrolling (4 SIMD instructions per thread), performance begins to saturate after two threads. Only with extremely high intensity (256 flops per element) does performance approach peak.

4.1.3 L1 Arithmetic Intensity Performance

Even when one can maintain a working set in the L1 cache , performance will be dependent on the dynamic instruction mix and the issue capability of the core. In this section, we leverage the Roofline Floating-Point Benchmark to quantify performance as a function of L1 Arithmetic Intensity expressed as Flops per Element (FPE) — essentially the degree of the polynomial. For each architecture, we run both the reference C code quantifying the ability of the architecture as well as the best performing SIMDized and unrolled implementation. Figure 4.3 presents the true performance capability of the architecture (in green) compared



(a) (b) (c) (d)

Figure 4.3. Basic GFlops code and optimized SIMDized unrolling GFlops code compared to theoretical GFlops on four platforms. (a) Edison. (b) Mira. (c) Babbage (MIC only). (d) Titan (GPU only).

with Figure 3.6.

We observe that on Edison, with appropriate parallelism, performance can sustain a high FPE. On Mira, we see the performance trend of optimized code is almost overlapped with the compiled code. On Babbage, although the trend of performance of optimized code behaves like the reference C code, the performance is rapidly approaching its peak.

4.2 Computation Ceilings

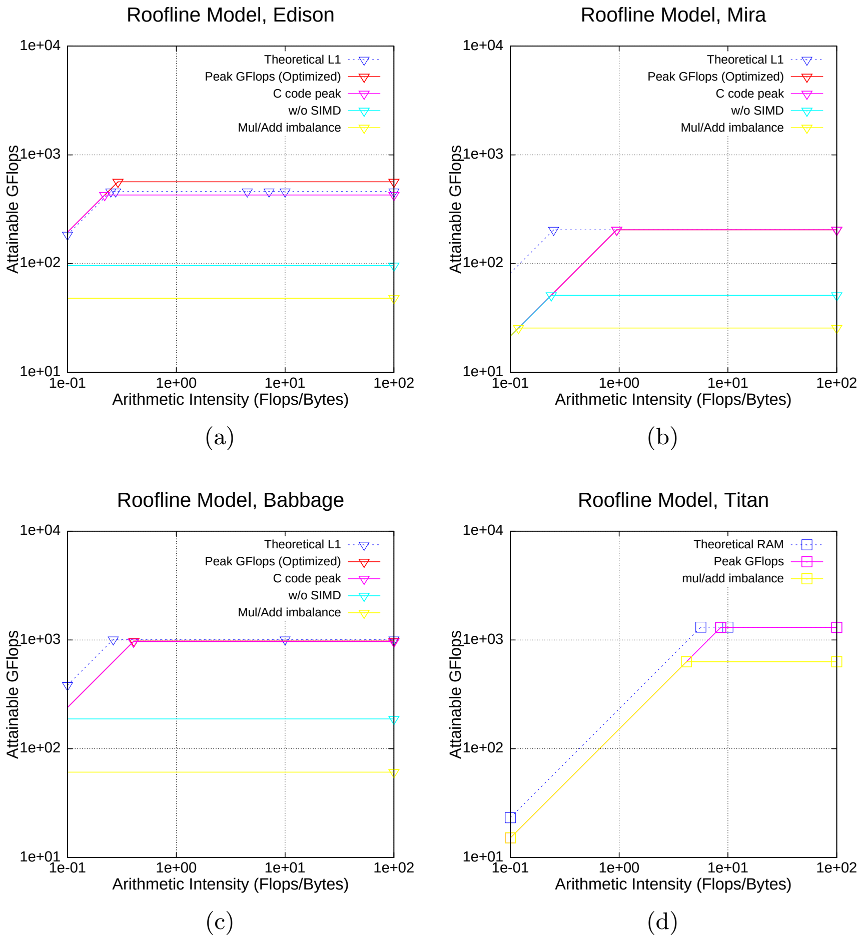
Now we leverage the GFlops Benchmark again to perform a sensitivity analysis in which we examine the impact on performance as we satisfy less and less of the in-core parallelism (data-parallelism, and floating-point functional units). Every time we remove one of these forms of parallelism, performance is diminished. To explore the impact of data-level parallelism, we could disable the SIMD flag at compile-time. For GPU architectures, there is no compiler option to simply disable SIMD. To examine floating-point functional units, we perform only multiplications with the same flops. We do not examine ILP impact on the GFLops Benchmark. However, one could create another micro benchmark like the code snippet shown on Figure 4.4 to study the ILP impact.

Figure 4.5 presents an expanded Empirical Roofline Model where all the computation performance ceilings are shown. The blue line marks the theoretical Roofline. The peak performance that can be obtained by the architecture on compiled code and the true performance capability of the architecture are marked in pink and red, respectively. The cyan line marks performance without using SIMD units, and finally, the yellow line marks performance without balancing add and multiply operations. As we can see, without exploiting SIMD units sufficiently by either the compiler or the programmer, the performance loss is around 78% on Edison, Mira, and Babbage. If kernels cannot always exploit fused multiply-add or achieve balance between multiplies and adds, performance would decrease by 50% at most. In conclusion, performance diminishes if in-core parallelism is not expressed across machines. The performance loss can be up to 91% compared to achievable

```
#pragma unroll UNROLLNUM
for (int i = 0; i < ITER; ++i)
{
  a = a * b + c;
}
```

```
#pragma unroll UNROLLNUM
for (int i = 0; i < ITER; ++i)
{
  a = a * b + c;
  d = d * b + c;
  e = e * b + c;
}
```

Figure 4.4. A suggestive ILP micro benchmark. (left) No ILP. (right) ILP = 3.



(a) (b) (c) (d)

Figure 4.5. Empirical Roofline Models with computation performance ceilings. (a) Edison. (b) Mira. (c) Babbage (MIC only). (d) Titan (GPU only).

peak performance. This in-core performance sensitivity analysis can provide insights into optimization strategies.

4.3 Parallelization Overheads

The time required to create a parallel region or synchronize threads can become an impediment for performance. As we observe in the bandwidth results, parallel overheads dominate for small working set sizes. The goal for this benchmark is to quantify the time required for these operations in OpenMP and MPI on multicore and manycore architectures.

4.3.1 OpenMP/MPI Overhead Benchmark

Given a serial program and a parallel program, let Ts be the execution time of the serial program and Tp be the execution time of the parallel program by using p OpenMP threads. Hence, we could define the "Overhead" of a parallel program as $Tp - Ts/p$.

To measure the overhead of OpenMP directives, like EPCC [18, 5, 6], we compare the time taken for a section of code executed sequentially, to the time taken for the same code executed in parallel.

The EPCC benchmark measures the overhead of an OpenMP barrier by measuring the time taken to perform a routine `testbar` on total OpenMP threads `outerreps` times (Fig. 4.6 (left)). The `testbar` routine performs a small kernel `delay` on a single thread `innerreps` times. By applying the above "Overhead" methodology, we could get the OpenMP barrier overhead.

However, unlike EPCC, our OpenMP benchmark eliminates the time to launch a parallel region. It can therefore more accurately estimate the OpenMP barrier overhead (Fig. 4.6 (right)). We also create a similar benchmark to measure MPI barrier overhead.

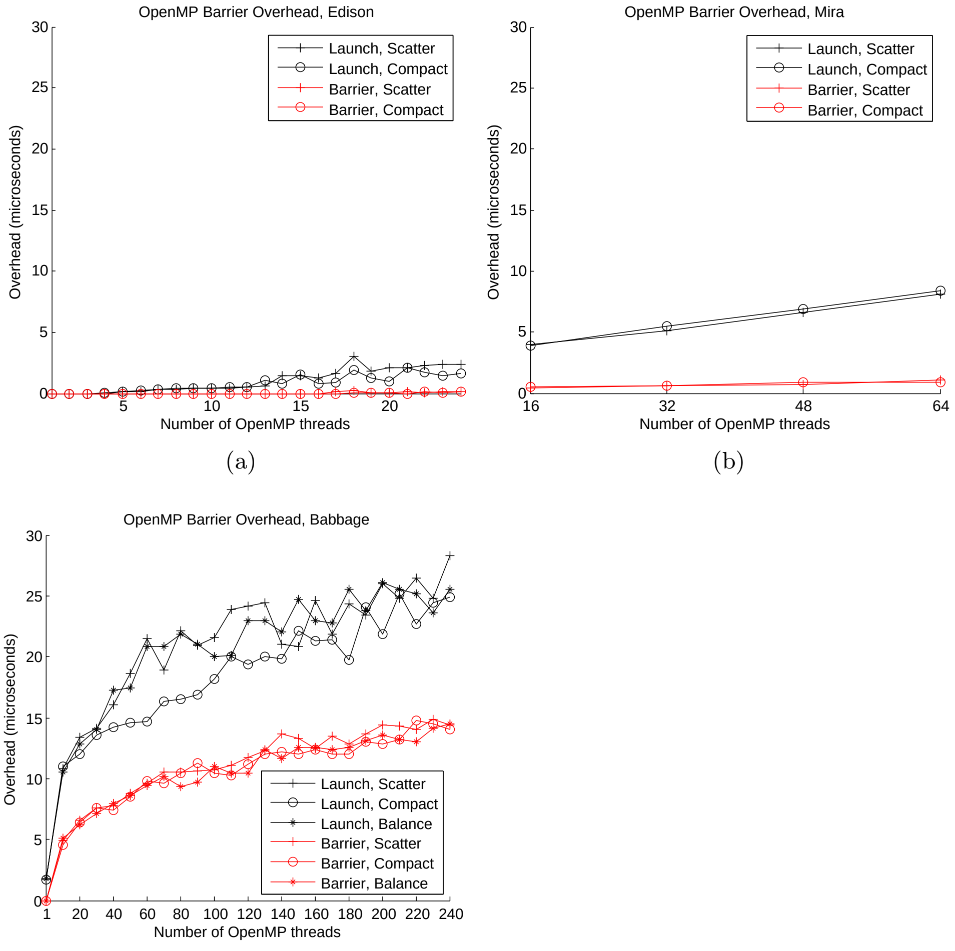
```
void testbar() {
  int j;
  #pragma omp parallel private(j)
  {
    for (j = 0; j < innerreps; j++) {
      delay(delaylength);
      #pragma omp barrier
    }
  }
}




void benchmark() {
  for (k = 0; k <= outerreps; k++) {
    start = getclock();
    testbar();
    times[k] = (getclock() - start) *
          1.0e6 / (double) innerreps;
  }
}
```

```
void barrierOve(uint64_t nsize, int nthreads,
                              double *time) {
  for (i = 0; i < INNER; ++i) {
    #pragma omp parallel
    {
      int id = omp_get_thread_num();
      #pragma omp barrier
      if (id == 0)
        start = getTime();
      work(nsize);
      #pragma omp barrier
      if (id == 0)
        *time += (getTime() - start) * 1e6;
    }
  }
  *time = *time / (double)INNER;
}

int main() {
  for (i = 0; i < OUTER; ++i) {
    barrierOve(nsize, nthreads, &barrier_time);
    total_barrier_time += barrier_time;
  }
}
```

Figure 4.6. OpenMP Overhead Benchmark comparison. (left) EPCC facsimile. (right) Roofline OpenMP Overhead Benchmark.

4.3.2 Results

Figure 4.7 presents the OpenMP overhead on multicore and manycore architectures by comparing different thread affinities. The black line marks the OpenMP parallel region launch overhead and the red line marks the OpenMP barrier synchronization overheads with different thread affinity. Thread affinity can restrict execution of certain threads to a subset of the physical processing units. Depending upon the topology of the architecture, thread affinity can have a dramatic effect on the execution speed of a program. First, *scatter* affinity evenly distributes the threads in a round-robin fashion across the entire system, assuring that threads do not share local cache. Second, *compact* affinity bounds consecutive threads as close as possible so that communication overhead, cache line invalidation overhead, and



(a)

(b)

(c)

Figure 4.7. OpenMP parallel region launch and barrier overhead on Edison, Mira, and Babbage. (a) Edison. (b) Mira. (c) Babbage (MIC only).

page thrashing are minimized. Thirdly, *balanced* affinity (a uniquely useful mode available to the MIC coprocessor) evenly distributes threads among the cores. However, unlike scatter, it will spread out threads to all cores before assigning multiple threads to a given core and keep thread numbers close to one another. The way to place threads can increase the probability that threads on the same core are using nearby data.

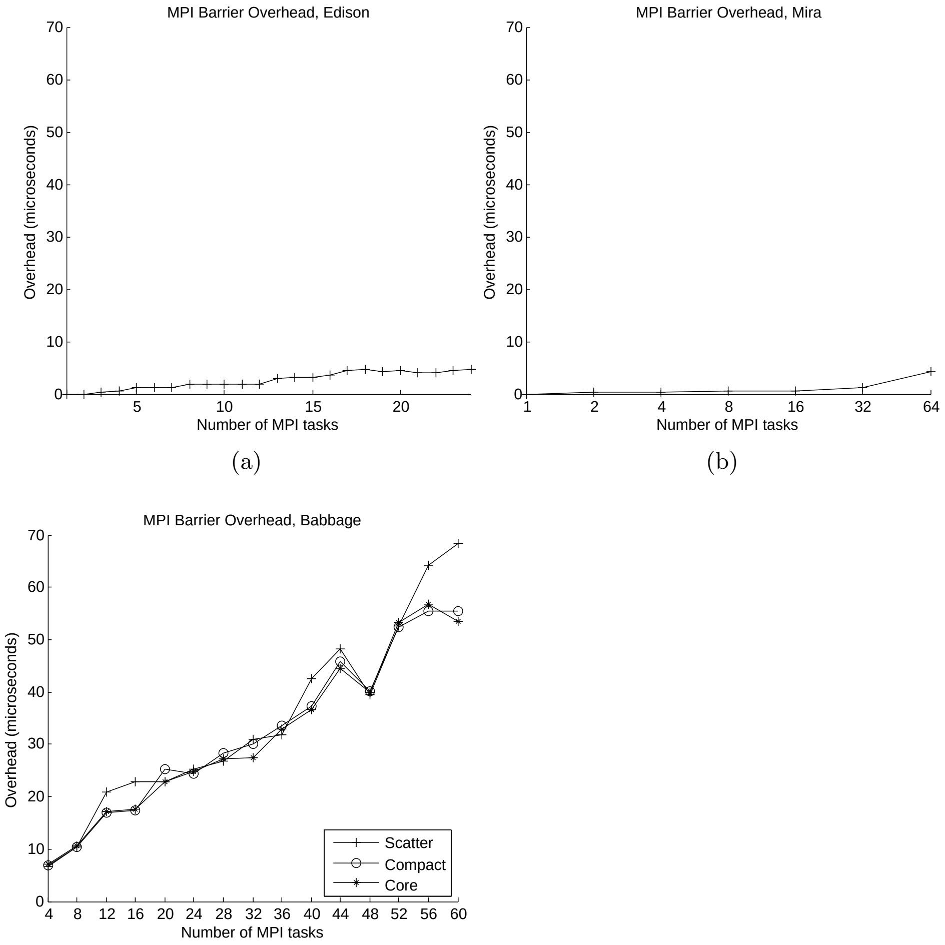
Among the three architectures, we can observe that overheads of creating a parallel region is around two times more than the overhead of barrier synchronization, and the time to create a parallel region is more sensitive to the number of threads than the time to synchronize. On Edison, the overhead is quite small and performance that uses *compact* affinity to place threads is slightly better than *scatter* affinity.

On Mira, the overhead is around two times Edison's overhead. We see that using different affinity does not affect performance significantly. Conversely, on Babbage, we see affinity affects performance dramatically. Due to the compensatory hardware threading and L2 cache layout, affinity is important on the MIC. We could observe that *scatter* affinity has the highest overheads and *compact* affinity has the lowest overheads. However, if we can not use the full 240 threads on Babbage, using *compact* affinity is unlikely to produce benefits on the MIC architecture as it tends to leave cores completely unused for thread counts below chip maximum. Conversely, one could use *balanced* affinity to fully utilize cores and cache efficiently.

Figure 4.8 presents the MPI overhead on Edison, Mira, and Babbage. We see the MPI barrier synchronization overhead is quite small on Edison and Mira. On Babbage, we see a similar performance trend when using different affinity but the MPI overhead is two times higher than the OpenMP overhead.

4.4 Memory Locality Performance Characterization

Reducing data communication over memory has been an optimization issue for scientific applications. Most current cache memory architectures work efficiently if subsequent memory accesses exhibit good locality of references. Hence, we are motivated to design a benchmark to quantify the performance of high-performance computing architectures along dimensions of spatial and temporal memory locality. This benchmark can be used to explore a software-hardware co-design to be able to leverage the inherent locality of programs and reduce data motion in the system.



(a)

(b)

(c)

Figure 4.8. MPI barrier overhead on Edison, Mira, and Babbage. (a) Edison. (b) Mira. (c) Babbage (MIC only).

4.4.1 Memory Locality Benchmark

To quantify the impact of spatial and temporal memory locality, we define two working sets — a total working set and an active working set, both shared among all threads and processes on a node. That is, one divides the *AWS* into *AWS_per_thr* and *TWS* into *TWS_per_thr*, and creates appropriate pointers *nid*. Each thread loops over the small chunk of an active working set *REUSE* times. Then the *start* loop jumps by *AWS_per_thr* step and stops by condition $start < TWS\_per\_thr$. By doing this, we can sample through a two-dimensional heatmap of active working sets (spatial memory locality) and reuse (temporal memory locality). By reusing the active working set, we could increase the temporal locality of references.

We implemented two versions of synchronization. The first synchronizes when the reusing loop is done (Fig. 4.9 (left)). The second synchronizes at every reuse point (Fig. 4.9 (left)). By doing this, we can see how fine-grained/coarse-grained synchronization influences effective bandwidths within an application.

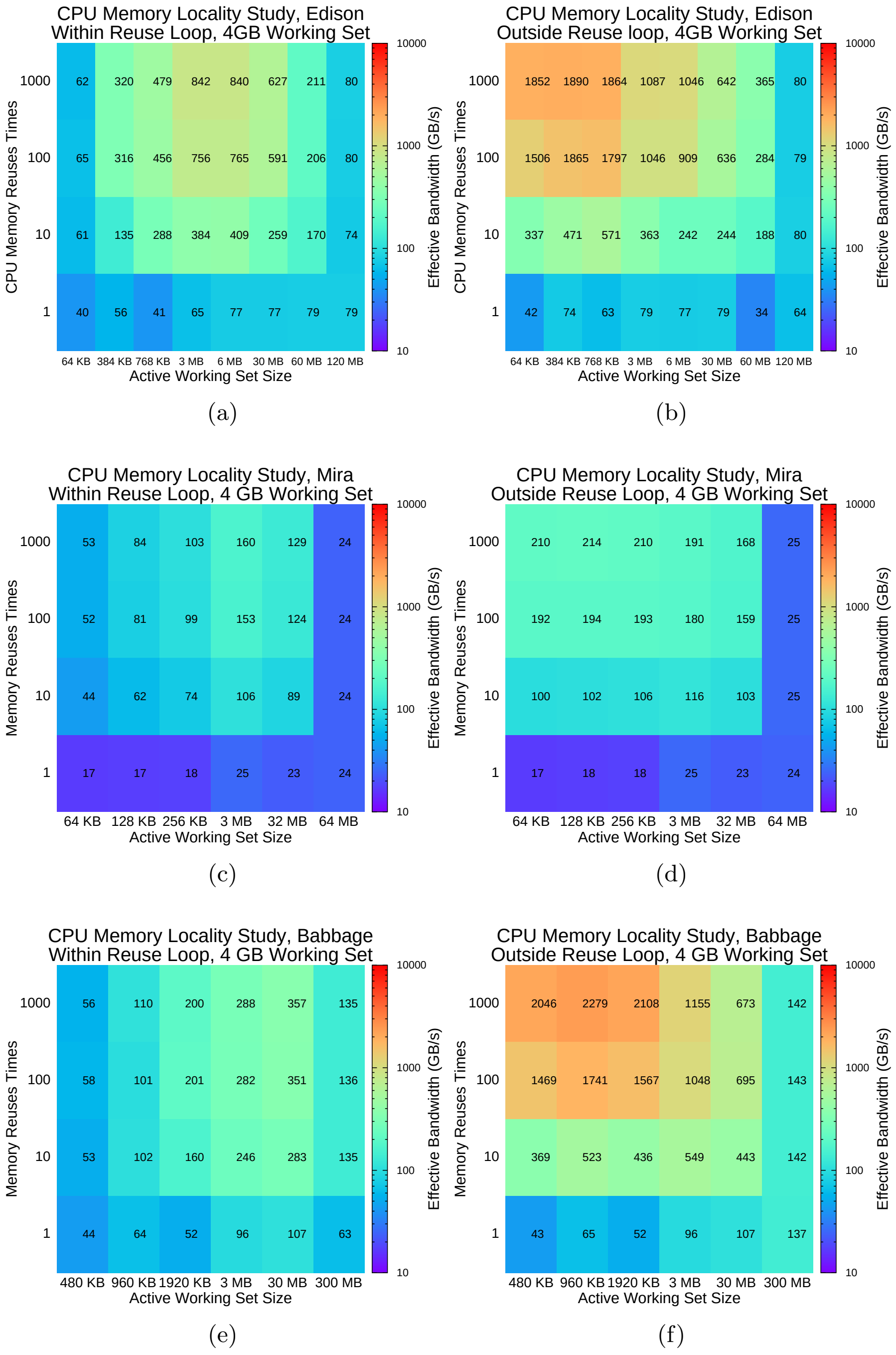
4.4.2 Results

We characterize spatial locality by the size of contiguous memory locations accessed in succession (active working set size). Temporal locality quantifies the amount of reuse of the same memory location during application execution. Figure 4.10 presents the effective bandwidth as a function of active working set size and amount of temporal reuse. We observe that on Edison when active working set size fits into L1 cache capacity, the machine can attain peak performance with high enough reuse (high temporal locality) and coarse-grained synchronization (Fig. 4.10(a)). Conversely, with fine-grained synchronization, we see the highest performance occurs when the active working set size fits into L2 cache (Fig. 4.10(b)). We can see a similar outcome on Mira and Babbage. Comparing Edison and Babbage (Fig. 4.10(a)(e)), with fine-grained synchronization, we observe that Edison's performance is better than Babbage if the active working set size fits into cache capacity. That is, with fine-grained locality, only when the active working set size fits into ultimate memory capacity, we will gain performance benefits from manycore architectures. Hence, one wishing

```
#pragma omp parallel private(id)
{
  nid = TWS_per_thr * id;
  double * __restrict__ A = &buf[nid];
  // start timer here...
  for (start = 0; start < TWS_per_thr;
       start += AWS_per_thr) {
    alpha = 0.5;
    for (r = 0; r < REUSE; ++r) {
      for (i = 0; i < AWS_per_thr; ++i) {
        A[start + i] = A[start + i] + alpha;
      }
      alpha = alpha * (1 - 1e-8);
    }
    #pragma omp barrier
    #pragma omp master
    {
      MPI_Barrier(MPI_COMM_WORLD);
    }
  }
  // stop timer here...
}
```

```
#pragma omp parallel private(id)
{
  nid = TWS_per_thr * id;
  double * __restricti__ A = &buf[nid];
  // start timer here...
  for (start = 0; start < TWS_per_thr;
       start += AWS_per_thr) {
    alpha = 0.5;
    for (r = 0; r < REUSE; ++r) {
      for (i = 0; i < AWS_per_thr; ++i) {
        A[start + i] = A[start + i] + alpha;
      }
      #pragma omp barrier
      #pragma omp master
      {
        MPI_Barrier(MPI_COMM_WORLD);
      }
      alpha = alpha * (1 - 1e-8);
  }}
  // stop timer here...
}
```

Figure 4.9. Memory Locality Benchmark code: (left) Synchronize outside the REUSE loop. (right) Synchronize within the REUSE loop.



(a) (b)

(c) (d)

(e) (f)

Figure 4.10. Effective bandwidth on Edison, Mira, and Babbage by comparing locality affect and fine-grained/coarse-grained synchronization. (a) Edison, sync at every REUSE point. (b) Edison, sync outside the REUSE loop. (c) Mira, sync at every REUSE point. (d) Mira, sync outside the REUSE loop. (e) Babbage, sync at every REUSE point. (f) Babbage, sync outside the REUSE loop.

to use a manycore architecture to accelerate their applications should use coarse-grained synchronization to avoid large parallelism overhead.

One could combine the results gained from the parallelization benchmarks in Section 4.3 and bandwidth benchmark in Section 3.1 to calculate theoretical bandwidth by using Equation 4.1.

$$\begin{gathered}
\text{Bandwidth} = \frac{AWS \times REUSE}{\text{A} + (reuse - 1) \times \text{B} + \text{barrier overhead}} \\
\text{if L2 cache capacity} \leq AWS \leq \text{L3 cache capacity then} \\
\text{A} = \frac{AWS}{\text{DRAM bandwidth}} \text{ and B} = \frac{AWS}{\text{L3 cache bandwidth}}
\end{gathered} \tag{4.1}$$

Figure 4.11 presents theoretical bandwidth on Mira by using Equation 4.1. The result shows that the theoretical bandwidth is higher than the resultant bandwidth gained from the locality benchmark. The effect may be due to the memory latency effect. This also indicates that the locality benchmark contains more information than the individual cache bandwidth and the OpenMP overhead benchmarks.

Overall, this benchmark can quantify the impact of spatial and temporal locality, and also guide developers as to what optimization is necessary to move an application from multicore to manycore architectures.

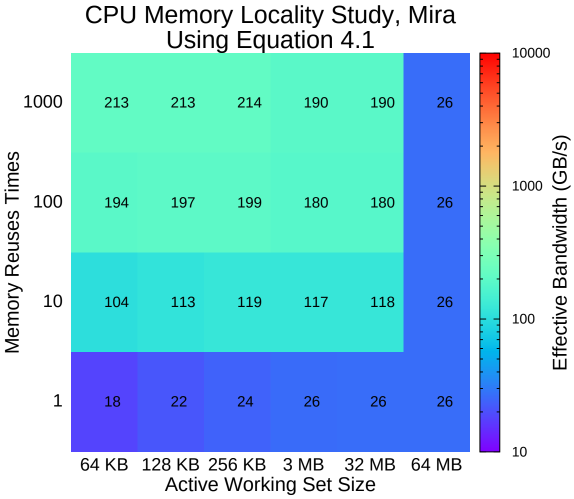


Figure 4.11. Mira's theoretical bandwidth based on Equation 4.1.

4.5 Bandwidth Ceilings

We have expressed deficiencies in locality in Section 4.4. In this section, we perform a sensitivity analysis in which we examine the impact on effective bandwidth as we satisfy less and less memory concurrency. Figure 4.12 presents an expanded empirical Roofline model where all the bandwidth ceilings are shown. The blue line marks the theoretical L1 bandwidth. The red line marks the achievable L1 bandwidth, the pink line marks the performance without exploiting appropriate locality of memory reference, and the cyan line marks the performance when allowing remote memory access (not uniformly utilizing all memory controllers).

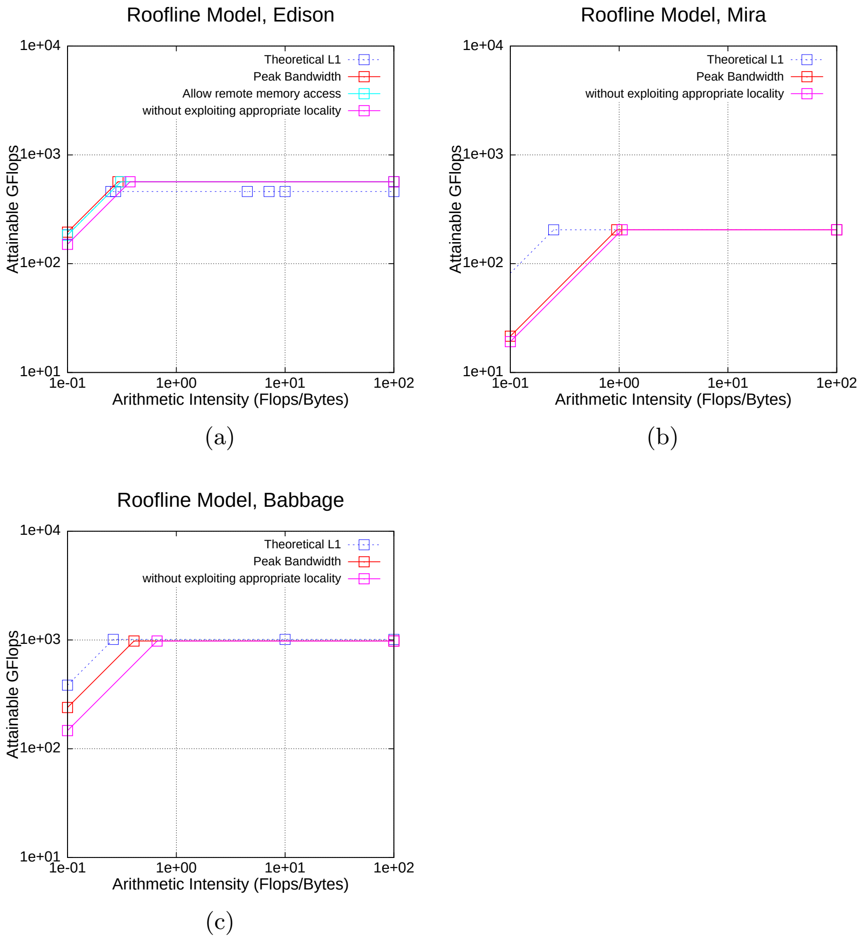
As we remove spatial and temporal locality optimizations, effective bandwidth diminishes among the three architectures. Since Edison is the only dual-sockets architectures, this machine should uniformly distribute memory traffic among the memory controllers both within a socket and across sockets. Failing to satisfy uniformly memory traffic distribution will result in idle cycles on one or more of the memory controllers. We can observe that, on Edison (Fig. 4.12 (a)), removing the `-ss` aprun option to allow remote memory access will expose the limited bandwidth.

4.6 Summary

In this chapter, we discussed three benchmark designs and expanded Empirical Roofline Models by adding bandwidth and computational ceilings. The SIMDize Floating-point benchmark can provide the true performance capability of the architecture and quantify the performance disparity with the compiled code. Results show that Edison can exceed the nominal peak performance due to the TurboBoost effect. Second, the XL/C compiler on Mira was able to effectively SIMDize and unroll the code to attain peak GFLops. On the manycore architecture, only with extremely high arithmetic intensity, the machine can approach its peak performance with sufficient parallelism.

In Section 4.3, we constructed a parallelization overhead benchmark to quantify the time required to create a parallel region or synchronize threads on multicore and manycore architectures. Result shows that the manycore architecture has higher parallelization overheads and is more sensitive to different thread affinity strategies.

In Section 4.4, we constructed a CPU locality benchmark to quantify the impact of spatial and temporal locality with the effect of fine-grained/coarse-grained synchronization, enabling hardware-software co-design to reduce data motion in the system. This benchmark can also be used to evaluate what degree of spatial and temporal locality is necessary to port an application to manycore architectures.



(a)

(b)

(c)

Figure 4.12. Empirical Roofline Models with bandwidth ceilings. (a) Edison. (b) Mira. (c) Babbage (MIC only).

Finally, the benchmark-driven Roofline Model is further enhanced by performing multiple sensitivity analyses in which we examine the performance as we progressively exploit the in-core parallelism and memory concurrency. Such a formulation can be very useful in modeling, predicting, and analyzing application performance.

CHAPTER 5

GPU BENCHMARKING

Accelerated architectures are becoming increasingly popular to offer dramatically better performance for scientific applications. The accelerated architecture differs significantly from multicore and manycore architectures. Hence, understanding the strengths and weaknesses of accelerated architectures, like Nvidia's GPU, is important for the high-performance computing community. In this chapter, we discuss some GPU's benchmark designs and use them to evaluate the performance capability of Nvidia K20 GPUs.

5.1 CUDA's Parallelization Overhead

To quantify the time required to launch a CUDA kernel or synchronize threads, one could create a serial and a parallel program to quantify the overhead like the CPU version we described in Section 4.3. Alternatively, one could simply run an empty CUDA kernel and record the elapsed time by using the CUDA event record "cudaEventElapsedTime".

Figure 5.1 presents the overhead results by using the latter approach. The black line marks the overhead of launching a CUDA kernel. The red line marks the overhead of thread synchronization. We observe that creating threads is relatively cheap on a GPU as using 488 blocks only increases 1 microsecond compared with 244 blocks. Moreover, the overall overhead is not large.

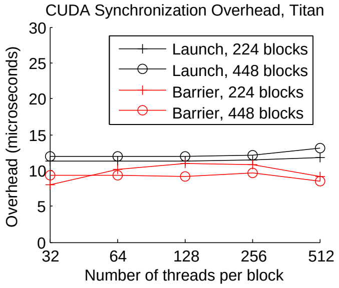


Figure 5.1. CUDA's parallelization overhead on Titan.

5.2 GPU's Memory Locality

We have shown that locality of memory references is important for CPUs. In this section, we want to know whether the same conclusion holds for GPUs.

5.2.1 GPU's Memory Locality Benchmark

Like the CPU's memory locality benchmark described in Section 4.4, we define two working sets and divide them into *AWS_per_thr* and *TWS_per_thr* by total number of threads (total threads per thread block × total thread blocks per kernel). Each thread performs a routine "sharedMem". This routine simply increments a one-dimensional array. We also implemented two versions of synchronization. The first one puts the *REUSE* loop within the CUDA kernel to increase locality within a thread block (Fig. 5.2 (left)). The second approach puts the REUSE loop outside the CUDA kernel call (Fig. 5.2 (right)).

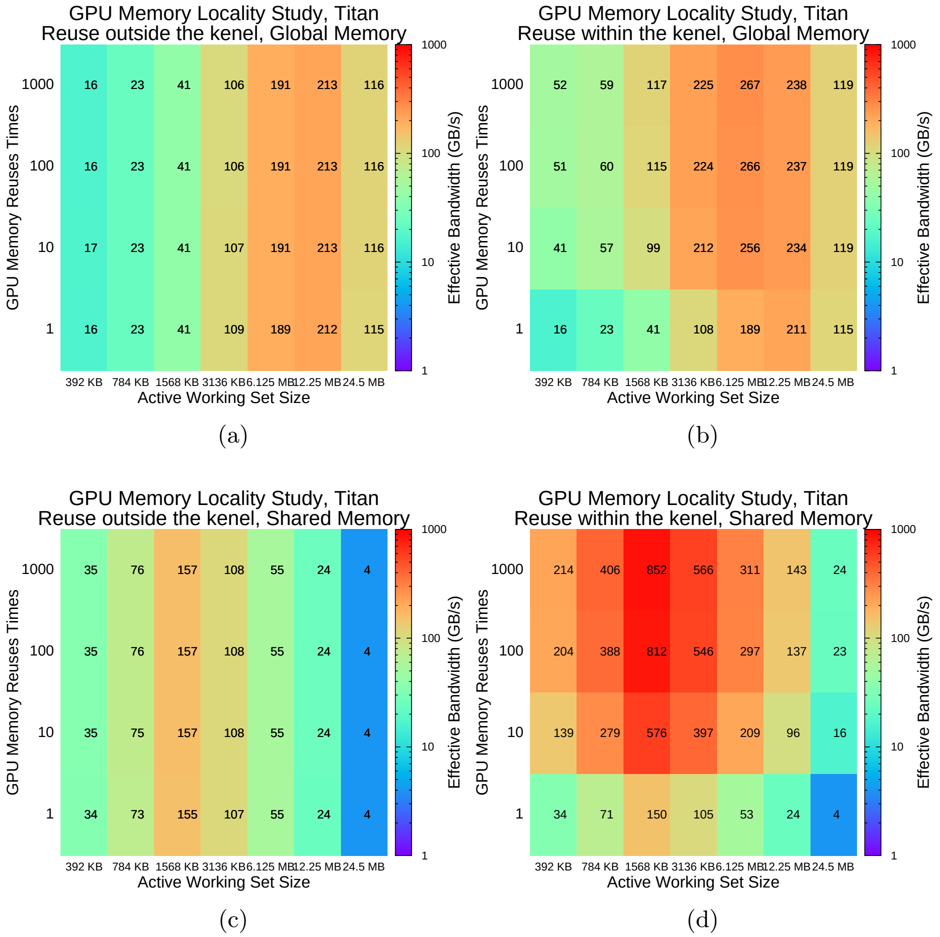
5.2.2 Result

Figure 5.3 presents the effective bandwidth as a function of the active working set size and an amount of temporal reuse. We operated on a one-dimensional array on global

```
#define BSIZE 224
#define TSIZE 32
__global__ void sharedMem(uint64_t size,
                double *A, int REUSE) {
  __shared__ double As[BSIZE];
  for (i = 0; i < size; ++i)
    As[i] = A[i];
  double alpha = 0.5;
  for (r = 0; r < REUSE; ++r) {
    for (i = 0; i < size; ++i) {
      As[i] = As[i] + alpha;
    }
    alpha = alpha * (1 - 1e-8);
  }
  for (i = 0; i < size; ++i)
    A[i] = As[i];
}

int main() {
  dim3 threads(TSIZE);
  dim3 blocks(BSIZE);
  TWS_per_thr = TWS/TSIZE/BSIZE;
  // start timer here...
  for (i = 0; i < TWS_per_thr; i += BSIZE)
    sharedMem <<<blocks, threads>>> (BSIZE,
                         &d_buf[i], REUSE);
  // stop timer here...
}
```

```
#define BSIZE 224
#define TSIZE 32
__global__ void sharedMem(
uint64_t size, double *A, double alpha) {
__shared__ double As[BSIZE];
  for (i = 0; i < size; ++i)
    As[i] = A[i];
  for (i = 0; i < size; ++i)
    As[i] = As[i] + alpha;
  for (i = 0; i < size; ++i)
    A[i] = As[i];
}



int main() {
  dim3 threads(TSIZE);
  dim3 blocks(BSIZE);
  TWS_per_thr = TWS/TSIZE/BSIZE;
  // start timer here...
  for (i = 0; i < TWS_per_thr; i += BSIZE) {
    alpha = 0.5;
    for (r = 0; r < REUSE; ++r)
      sharedMem <<<blocks, threads>>> (BSIZE,
                                &d_buf[i], alpha);
    alpha = alpha * (1e-8);
  }
// stop timer here...
}
```

Figure 5.2. GPU's Memory Locality Benchmark code: (left) Synchronize within the CUDA kernel. (right) Synchronize outside the CUDA kernel.



(a)

(b)

(c)

(d)

Figure 5.3. GPU locality of memory reference impact on Titan. (a) Global memory, REUSE outside the kernel. (b) Global memory, REUSE within the kernel. (c) Shared memory, REUSE outside the kernel. (d) Shared memory, REUSE within the kernel.

memory (Fig. 5.3(a)(b)) and shared memory (Fig. 5.3(c)(d)).

We could observe that, with appropriate spatial and temporal locality of memory reference, the machine can come close to its bandwidth limit like CPUs. However, unlike multicore and manycore architectures, we see increasing temporal locality will not improve the effective bandwidth significantly when using fine-grained synchronization. That is, presumably putting the REUSE loop outside the CUDA kernel will not achieve temporal locality. The surprisingly low bandwidth at the 24.5MB active working set size on Figure. 5.3(c)(d) is due to the overhead of copying data between global and shared memory. With fine-grained synchronization, this overhead increases by the amount of temporal reuse.

5.3 CUDA's Unified Memory

To date, accelerated architectures have been typically used as an accelerator with dedicated memory attached to a conventional system through a PCIe or similar bus. Not only does this dedicated memory have its own unique address space, but programmers were forced to explicitly copy data to and from the device via a library interface. Doing so is not only unproductive, but also exposes the performance disparity between the PCIe bandwidth and device bandwidth.

Recently, CUDA introduced two memory concepts — the Unified Virtual Address (UVA) space, and Unified Memory (i.e. managed memory) [13]. As the name suggests, UVA unifies the CPU and GPU address spaces and ensures (at the program level) that programs may transparently load and store memory without worrying about the locality of data (for correctness). As data remain pinned to host or device, there are strong NUMA effects. Unified (Managed) Memory extends this process by migrating data between the host and the device. As such, device memory can be viewed as a cache on the CPU memory. Ideally, this would address many of the productivity and performance challenges. In this section, we evaluate the performance of these approaches as a function of spatial and temporal locality.

5.3.1 CUDA Managed Memory Benchmark

Our initial approach to this benchmark was to create a benchmark that thrashes data back and forth between host and device. To that end, we reuse the Roofline Bandwidth Benchmark by having the GPU perform $k - 1$ iterations of the summation and the CPU perform 1. As the net reuse k increases, we expect the cost of moving the data between host and device to be amortized.

Please note, this benchmark is not an unreasonable scenario in practice as many applications may package some data for the GPU, copy it to the device, operate on it a few times, then return it to the host. If written using Unified Memory, the data would thrash back and forth between host and device.

We evaluate performance using four different approaches to controlling the locality of data on the device. First, we evaluate the conventional explicit copy (`cudaMemcpy`) approach using either a paged array or a page-locked array allocated on the host. Next, we evaluate the performance of zero copy memory. In this scenario, data are allocated and pinned on the host and it is the responsibility of the CUDA run time to map load and store requests to PCIe transfers. Finally, we evaluate the performance of the Unified (Managed) Memory construct in which the CUDA run time may migrate data.

Figure 5.4 presents these implementations. As one can see, increased locality is affected via multiple CUDA kernel invocations. The macros “_CUDA_ZEROCOPY” and “_CUDA_UM” select the use of page-locked host with zero copy and unified memory management, respectively. Page-locked host memory uses a normal `malloc()` function to allocate memory on the host, and then uses `cudaHostRegister()` to register a device pointer on the host memory address space. For unified memory, one uses `cudaMallocManaged` to allocate both host and device memory.

5.3.2 Results

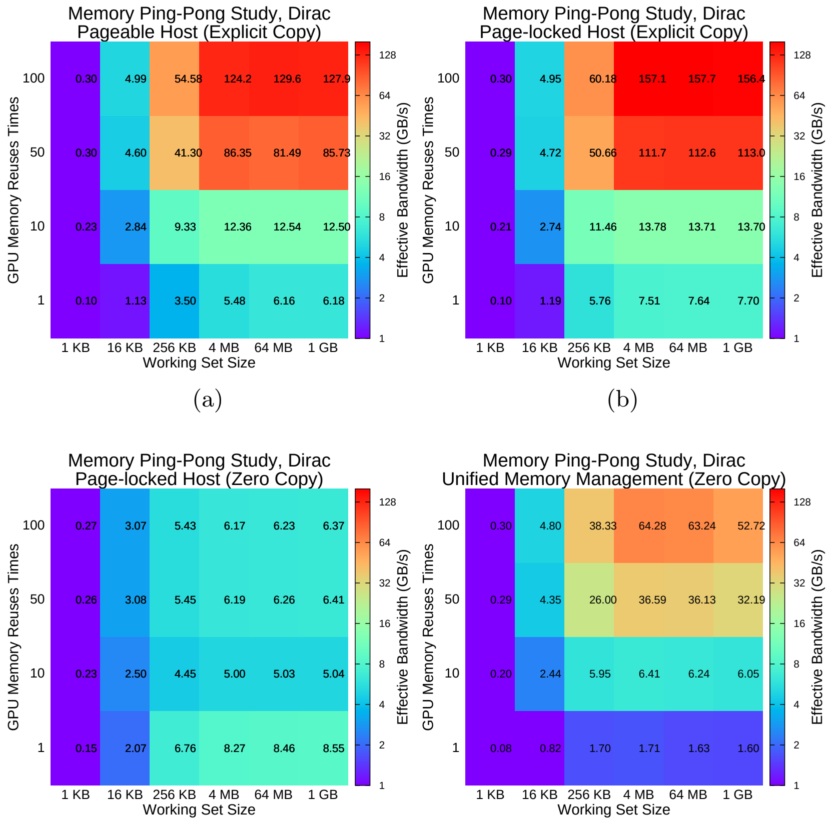
As Titan does not support CUDA 6 yet, all of our experiments were run on a similar K20xm in the Dirac cluster[1].

Figure 5.5 presents the resultant effective bandwidth for the four technologies as a function of working set size and temporal reuse. For small working set sizes, CUDA kernel launch time dominates and effective bandwidth is abysmal. This simply reinforces the conventional wisdom not to use the GPU for small operations. Comparing Figure 5.5(a) and (b), we see that it is possible to approach the device bandwidth limit, but only for

[1] GPU driver version: 331.89; CUDA toolkit version: 6.0beta.

```
int main()
{
  // start timer here...
  for (uint64_t j = 0; j < trials; ++j) {
    #if defined(_CUDA_ZEROCPY) || defined(_CUDA_UM)
      cudaDeviceSynchronize();
    #else
      cudaMemcpy(d_buf, h_buf, SIZE, cudaMemcpyDefault);
    #endif
    for (uint64_t k = 0; k < reuse; ++k) {
      GPUKERNEL <<<blocks, threads>>> (n, d_buf, alpha);
      alpha = alpha * (1e-8);
    }
    #if defined(_CUDA_ZEROCPY) || defined(_CUDA_UM)
      cudaDeviceSynchronize();
    #else
      cudaMemcpy(h_buf, d_buf, SIZE, cudaMemcpyDefault);
    #endif
    CPUKERNEL(n, h_buf, alpha);
  }
  // stop timer here...
  double bytes = 2 * sizeof(double) * (double)n *(double)trials * (double)(reuse + 1);
}
```

Figure 5.4. CUDA Unified Memory Benchmark quantifies the ability of the runtime to manage locality on the device



(a)

(b)

(c)

(d)

Figure 5.5. Effective bandwidth as a function of GPU temporal locality (reuse) and working set size for four different GPU device memory management mechanisms. (a) Pageable host with explicit copy between CPU and GPU. (b) Page-locked host with explicit copy between CPU and GPU. (c) Page-locked host with zero copy. (d) Unified (Managed) Memory.

large working sets that are reused 50-100 times. Thus, offloading iterative solvers to the GPU is a viable option if one expects it to take hundreds of iterations to converge. Conversely, for large working sets with minimal reuse, we see that page-locked memory provides substantially better PCIe bandwidth.

As Zero Copy memory provides no caching benefit, we see no performance benefit in Figure 5.5(c) from increased locality. Conversely, Figure 5.5(d) presents the performance benefit from using Unified Memory to automate the management of data locality on the device. Broadly speaking, performance is qualitatively similar to the performance with explicitly managed locality. Unfortunately, the raw performance is substantially lower. For applications which could guarantee 1000-way reuse on the device, Unified Memory would provide a productive and high-performance solution. One can only hope that advances in hardware and runtime can bridge the performance gap at lower temporal locality.

5.4 Summary

In this chapter, we discussed advanced GPU benchmarks designed to quantify the parallelization overheads and memory locality effect within GPUs. Results show that one wishing to accelerate their applications on GPUs should optimize for locality and use coarse-grained synchronization.

In Section 5.3, we constructed a novel benchmark to evaluate four CUDA's memory management technologies. Generally speaking, the only way to come close to the PCIe performance limit is to operate on a large working set, explicitly copy the data between the host and the device, and reuse the data on device. Although a programmer could ease their burden by using the unified memory, the unified memory construct does not really quantify how much data movement between the host and the device will be needed.

CHAPTER 6

APPLICATION ANALYSIS

This chapter discusses the fundamentals of three HPC applications — the finite-volume High-Performance Geometric Multigrid (HPGMG-FV) benchmark [23], the Gyrokinetic Toroidal Code (GTC) [20], and miniDFT [27]. We use the Empirical Rooflines for Mira to analyze observed performance on three HPC applications. All applications were run on Mira (Blue Gene/Q system) where the performance counters (BGPM API) have been verified.

6.1 HPGMG-FV

High-performance geometric multigrid (HPGMG) is a new and standard benchmark based on geometric multigrid methods as an alternate Top500 benchmark. This benchmark includes two implementations — finite element and finite volume. We focus on the finite volume method in this thesis.

HPGMG-FV is a highly optimized multigrid benchmark that solves a constant- or variable-coefficient Poisson's equation on a structured grid using Full Multigrid (FMG). Geometric Multigrid is a specialization in which the linear operator (e.g. the operator A in Ax = b) is simply a stencil on a structured grid. Thus this benchmark implemented with a portable MPI+OpenMP model uses a distributed "V-Cycle" that allows restriction of a trillion cells distributed across a hundred thousand processes down to one cell, and its data are decomposed level-by-level. Generally speaking, Multigrid has three components that dominate performance. First, data movement within DRAM and flops to perform each stencil is constrained by DRAM and flop rates. Second, MPI data movement for halo/ghost zone exchanges is limited by MPI point-to-point bandwidth. Finally, MPI/OpenMP/CUDA parallelism may result in latency/overhead for each operation.

- **Application Analysis:**

We use the Mira's Empirical Roofline Model to analyze memory bandwidth-intensive HPGMG-FV. Figure 6.1 shows that it has low compute intensity, but it delivers performance very close to its DRAM bandwidth limit either using flat MPI or OpenMP.

6.2 GTC

The gyrokinetic toroidal code (GTC) is a turbulent transport fusion simulation, a massively parallel code implemented with a hybrid MPI+OpenMP model, that uses the particle-in-cell (PIC) method. Particle-grid interpolation is a known performance bottleneck in several PIC applications. In GTC, its two dominant kernels are particle-to-grid interpolation (`chargei`) and grid-to-particle interpolation (`pushi`). Theoretically, these kernels are moderately compute intensive (`pushi` slightly more) but involve random access to a structured grid.

- **Application Analysis:**

Figure 6.2 shows the resultant GTC's performance on Mira. We could clearly observe that the performance of both `chargei` and `pushi` routines is well below the Roofline bandwidth bound, and the overall GTC's performance sits between two routines. As we expected, `pushi` is relatively close to its computational performance bound.

6.3 MiniDFT

The MiniDFT code, implemented with OpenMP and MPI, uses plane-wave density functional theory (DFT) to compute the Kohn-Sham equations that performs only LDA

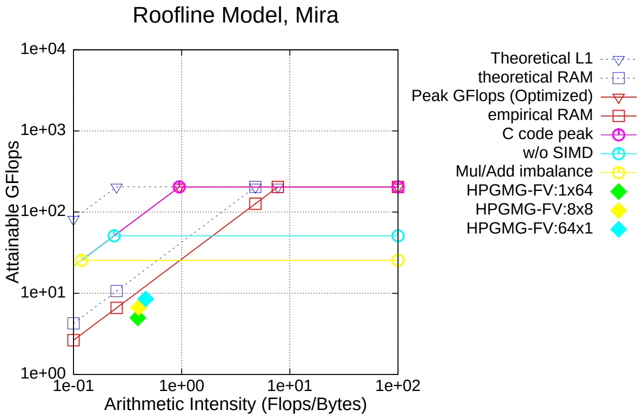


Figure 6.1. HPGMG-FV's resultant performance on Mira. Legends denote "benchmark: number of MPI tasks x number of OpenMP threads."

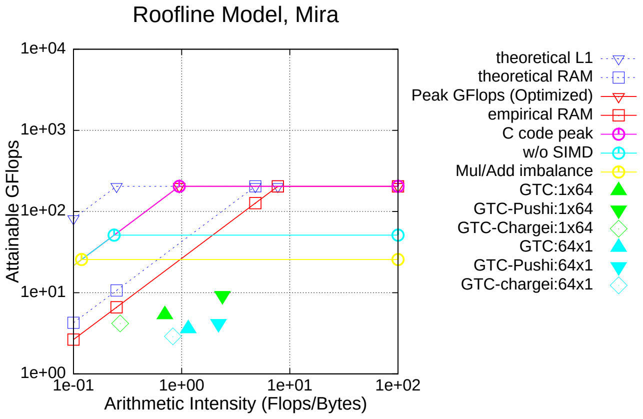


Figure 6.2. GTC's resultant performance on Mira. Legends denote "benchmark: number of MPI tasks x number of OpenMP threads."

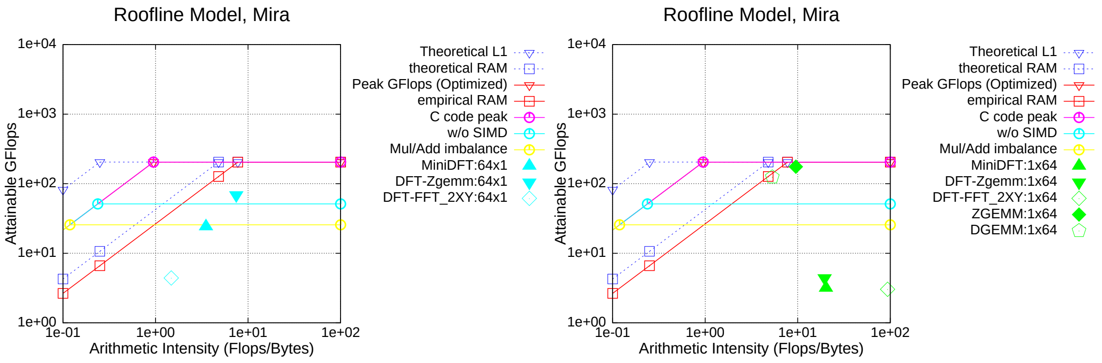
exchange-correlation functions, part of the general-purpose Quantum Espresso (QE) code for modeling materials. This can be used to explore new parallelization schemes and programming models, and evaluate their suitability for plane-wave DFT calculations. MiniDFT is a compute-intensive code, dominated by dense linear algebra and 3D FFTs.

- **Application Analysis:**

Figure 6.3 presents the resultant performance on Mira compared with DGEMM and ZGEMM routines ($C := alpha \times op(A) \times op(B) + beta \times C$). Although miniDFT uses matrix-matrix multiplications, the application performance is far less than peak DGEMM or ZGEMM performance. This is likely an artifact of the inherent performance differences between square multiplications and the block vector multiplications used in miniDFT. We can observe that flat MPI performance generally tracks the Roofline well. Conversely, the performance of the threaded code was orders of magnitude less than ideal although the arithmetic intensity of threaded code reaches the computational bound. Perhaps it is due to limited parallelism in any one dimension.

6.4 Summary

In this chapter, we discussed our observations and insights derived from analyzing three HPC applications, HPGMG-FV, GTC, and MiniDFT on Mira's Empirical Roofline Model. We believe that developers wishing to understand their applications' performance will find the Empirical Roofline Model is extremely useful to effectively find optimization benefits from the target architecture.



(a)

(b)

Figure 6.3. MiniDFT's resultant performance on Mira. Legends denote "benchmark: number of MPI tasks x number of OpenMP threads." (a) MPI Only. (b) MPI+OpenMP.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter summarizes this thesis and primary contributions. Also, it discusses the future work to generalize the Roofline Model concept.

7.1 Summary and Conclusions

This thesis proposed the benchmark-driven Roofline Model Toolkit that not only provides performance expectations for each target architecture, but also offers insights into the capabilities and efficiency associated with each architecture. Beyond conventional Roofline analysis, these Roofline micro benchmarks could help one to get extensive knowledge of memory subsystems.

This thesis's contributions include:

- In Chapter 3, we have described the Roofline Benchmarks that use a hybrid MPI and OpenMP model. These benchmarks focus on capturing the attainable bandwidth and the floating-point performance of each level of the memory hierarchy, along with thread-level, instruction-level, and explicit SIMD parallelism. In Chapter 4, we proposed the other advanced benchmarks to evaluate parallelism and locality issues, enabling hardware-software co-design to reduce data motion within applications. Finally, in Chapter 5, we benchmarked GPUs to evaluate parallelism and locality issues like we observed in CPUs. Moreover, we also created a novel benchmark to evaluate software managed cache technologies in CUDA to provide in-depth knowledge in memory subsystems.

- The architecture characterization engine can not only provide the achievable performance and its resultant gap between reality and theory on current architectures, but also be used to predict performance on potential new architectures, enabling informed algorithm design and implementation. In Chapter 4, we also extended the Roofline Model by adding computation performance and bandwidth ceilings to diagnose performance problems on existing architectures.

- We use this toolkit to benchmark four leading HPC systems: Edison, Mira, Babbage, and Titan. Edison's Empirical Roofline Model, with sufficient parallelism, is very close to its theoretical model. On Mira, although the XLC compiler is able to effectively SIMDize and unroll the code sufficiently to hide floating-point latency, the low L1 bandwidth issue results in the large gap between theory and reality. On Babbage and Titan, the extreme multithreading paradigm allows the MIC or GPU to deliver a high fraction of its theoretical bandwidth when running on the device. Finally, we leveraged the Empirical Roofline Model to analyze three HPC applications — HPGMG-FV, GTC, and MiniDFT, to offer observations and insights into performance benefits to optimize application performance or redesign architectures.

7.2 Future Work

There is a beta-release of the Empirical Roofline Tool, ERT, in Lawrence Berkeley National Laboratory [19]. Future work will continue to generalize this ERT tool, design other benchmarks to generalize the Roofline Model, as well as continue instrumentation, benchmarking, and analysis of HPC applications to explore performance and parallelism issues on emerging HPC platforms. We summarize potential benchmarking efforts here:

- A benchmark for characterizing short-stanza memory performance to express more memory-level parallelism.
- A benchmark for characterizing MPI performance, including point-to-point messaging and collectives (one-to-all, all-to-one, all-to-all).
- A benchmark for characterizing more parallelization overhead issues, including OpenMP critical section, etc.
- A benchmark for characterizing Network-on-Chip performance.

We believe that the use of Empirical Roofline Tool will help understand current and future design points of architectures as well as algorithms/applications.

REFERENCES

[1] Application Performance Characterization Benchmarking (APEX). `http://crd.lbl.gov/departments/computer-science/performance-and-algorithms-research/research/previous-projects/apex/`.

[2] Babbage Testbed. `www.nersc.gov/users/computational-systems/testbeds/babbage`.

[3] Bailey, D. H., Lucas, R. F., and Williams, S. W. *Performance Tuning of Scientific Applications*. CRC Press, New York, 2011.

[4] Bronevetsky, G., Gyllenhaal, J., and de Supinski, B. R. Clomp: Accurately characterizing openmp application overheads. *International Journal of Parallel Programming, Volume 37, Issue 3* (2009).

[5] Bull, J. M. Measuring synchronisation and scheduling overheads in openmp. *Proceedings of the First European Workshop on OpenMP, Lund, Sweden* (1999).

[6] Bull, J. M., and D, I. A microbenchmark suite for openmp 2.0. *SIGARCH Computer Architecture News* (2001).

[7] Bull, J. M., and D, I. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. *Symposium on Application Accelerators in High-Performance Computing (SAAHPC'11)* (2011).

[8] Choi, J., Bedard, D., Fowler, R., and Vuduc, R. A roofline model of energy. *IEEE IPDPS* (May 2013).

[9] Cori Cray XC30. `www.nersc.gov/users/computational-systems/nersc-8-system-cori`.

[10] Corporation, I. Intel xeon phi corprocessor system softeare developers guide. *Intel* (June 2012).

[11] Corporation, I. Ibm system blue gene solution: Blue gene/q application development. *IBM* (June 2013).

[12] Corporation, N. Kepler gk 110: The fatest, most efficient hpc architecture ever built. *Nvidia v1.0* (2012).

[13] Corporation, N. Cuda c programming guide. *Nvidia PG-02819 v6.0* (Feb. 2014).

[14] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and v. Eicken, T. Logp: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices 28* (1993).

[15] Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., and Yelick, K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review* (2009).

[16] Dirac Testbed. www.nersc.gov/users/computational-systems/testbeds/dirac.

[17] Edison Cray XC30. www.nersc.gov/systems/edison-cray-xc30.

[18] EPCC OpenMP micro-benchmarkSuite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite.

[19] Empirical Roofline Tool website. http://crd.lbl.gov/departments/computer-science/performance-and-algorithms-research/research/roofline/.

[20] Gyrokinetic Toroidal Code Website. http://phoenix.ps.uci.edu/GTC.

[21] Hager, G., Treibig, J., Habich, J., and Wellein, G. Exploring performance and power properties of modern multicore chips via simple machine models. *CoRR abs/1208.2908* (2012).

[22] HPCToolkit website. http://hpctoolkit.org/.

[23] HPGMG website. http://hpgmg.org.

[24] Immerman, N. Expressibility and parallel complexity. *Siam Journal on Computing, vol. 18, no. 3, pp. 625-638* (1989).

[25] Kamil, S., Husbands, P., Oliker, L., Shalf, J., and Yelick, K. Impact of modern memory subsystems on cache optimizations for stencil computations. *ACM MSP* (2005).

[26] LLCBench - Low Level Architectural Characterization Benchmark Suite. http://icl.cs.utk.edu/projects/llcbench/index.htm.

[27] QEforge website: MiniDFT. qe-forge.org/gf/project/minidft.

[28] Narayanan, S. H. K., Norris, B., and Hovland, P. D. Generating performance bounds from source code. *In Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)* (September 2010).

[29] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[30] Stockmeyer, L., and Vishkin, U. Expressibility and parallel complexity. *Siam Journal on Computing, vol. 13, pp. 409-422* (1984).

[31] STREAM benchmark. www.cs.virginia.edu/stream/ref.html.

[32] Strohmaier, E., and Shan, H. Architecture independent performance characterization and benchmarking for scientific applications. *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (October 2004).

[33] Strohmaier, E., and Shan, H. Apex-map: A global data access benchmark to analyze hpc systems and parallel programming paradigms. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference* (November 2005).

[34] Tuning and Analysis Utilties. `http://www.cs.uoregon.edu/research/tau/home.php`.

[35] Performance Analysis Tools: Vampir, VampirTrace, and Score-P. `http://www.paratools.com/Vampir`.

[36] Williams, S. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, December 2008.

[37] Williams, S., Watterman, A., and Patterson, D. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM* (April 2009).