Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis

Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker

University of Utah,
Lawerence Berkeley National Laboratory
{yujunglo,mhall}@cs.utah.edu
{swwilliams,bvstraalen,tjligocki,mjcordery,njwright,loliker}@lbl.gov

Abstract. We present preliminary results of the Roofline Toolkit for multicore, manycore, and accelerated architectures. This paper focuses on the processor architecture characterization engine, a collection of portable instrumented micro benchmarks implemented with Message Passing Interface (MPI), and OpenMP used to express thread-level parallelism. These benchmarks are specialized to quantify the behavior of different architectural features. Compared to previous work on performance characterization, these microbenchmarks focus on capturing the performance of each level of the memory hierarchy, along with thread-level parallelism, instruction-level parallelism and explicit SIMD parallelism, measured in the context of the compilers and run-time environments. We also measure sustained PCIe throughput with four GPU memory managed mechanisms. By combining results from the architecture characterization with the Roofline model based solely on architectural specifications, 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.

Keywords: Roofline, Memory Bandwidth, CUDA Unified Memory

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. 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 Roofline Model [20, 19, 2]. The Roofline model combines arithmetic intensity, memory performance, and floating-point performance together into a two-dimensional graph using bound and bottleneck analysis. In the conventional use, the x-axis is arithmetic intensity (flops per byte) and y-axis is performance in GFlop/s. The model thus defines an envelope in which one may attain performance. To date, this "textbook" Roofline model requires a human to manually analyze an architecture and any application kernels in order to populate the roofline. We wish to automate that process.

This paper will present our initial approach that constructs the Roofline model using an automated characterization engine. Moreover, we extend the Roofline formalism to address the emerging challenges associated with accelerated architectures. To that end, we constructed three benchmarks designed to drive empirical Roofline-based analysis. The first two represent the conventional memory hierarchy bandwidth and floating-point computation aspects of the Roofline. The third benchmark is a novel and visually intuitive approach to analyzing the importance of locality on accelerated architectures like GPUs. It quantifies the performance relationship between explicitly and implicitly managed spatial and temporal locality on a GPU. We evaluate these benchmarks on four platforms — Edison (Intel Xeon 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 Rooflines to analyze three HPC benchmarks — HPGMG-FV, GTC, and miniDFT.

2 Related Work

Today, data movement often dominates computation. Typically, this data movement is between DRAM and the cache hierarchy and is often structured streaming (array) accesses. As such, the STREAM benchmark has become the de-facto solution for benchmarking the ultimate DRAM bandwidth of a multicore processor [18]. STREAM is OpenMP threaded and will perform a series of benchmarks designed to quantify the memory subsystem's performance as a function of common array operations. 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.

Modern microprocessors use hardware stream prefetchers to hide memory latency by speculatively loading cache lines. Unfortunately, the performance of these prefetchers is highly dependent on architecture and it has been observed that bandwidth is highly correlated with the number of elements accessed contiguously [15]. Short "stanzas" of memory access see substantially degraded performance. Stanza Triad was created to quantify this effect [9]. Unfortunately, it is not threaded and as such cannot identify when one has transitioned from a concurrency-limited regime to a throughput-limited regime when running on multicore processors.

When DRAM bandwidth is not the bottleneck to on-node application performance, then cache bandwidth often is. CacheBench (part of LLCbench) can be used to understand the capacities and bandwidths of the cache hierarchy. [16]. 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). 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 [13].

Perhaps the most similar work to ours is encapsulated in the benchmarks used to drive the Energy Roofline Model [3]. In that work a series of experiments were constructed that varies arithmetic intensity in order 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.

3 Experimental Setup

The diversity of existing and emerging hardware and programming models makes construction of generalized benchmarks particularly difficult. To demonstrate the utility of our automation strategy, we evaluate performance on 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 many-core processor (Babbage), and a high-performance NVIDIA Kepler K20x GPU accelerated system (Titan). These systems represent a basis of system architectures within the HPC community today. The next three sections provide some background on their processor architectures, programming model and compilation options, and execution on our selected platforms.

3.1 Architectural Platforms

Table 1 summarizes the key architectural characteristics of these platforms. Please note that the peak GFlop/s and bandwidths shown are theoretical.

Edison: is a MPP at NERSC [11]. 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 4-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. 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.

Mira: is an IBM Blue Gene/Q system installed at the Argonne National Lab [5]. Each node includes one 16-core BGQ SOC. Each of the 16 A2 cores is a 4-way SMT dual-issue in-order core capable of performing one ALU/Load/Store instruction and one four-way FMA per cycle. However, in order 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: is a Knights Corner (KNC) Manycore Integrated Core (MIC) testbed at NERSC [1, 6]. The KNC processor includes 60 dual-issue in-order 4-way HyperThreaded cores. Each core includes a 32KB L1 data cache, a 512KB L2 cache, and a 8-way vector unit. Although the L2 cache's 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. In order to proxy the future Knights Landing (KNL) MIC processor that will form the heart of the NERSC8 Supercomputer Cori [4], we conduct all experiments in "native" mode. As such, the host processor, the host memory, and the PCIe connection are not exercised.

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 [7]. 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 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 [10] in order to evaluate the CUDA unified virtual address and Unified (managed) Memory. For our purposes, the K20x and K20xm GPUs are identical.

3.2 Programming Model and Compilation

In this section, we provide the compiler flags that were on different platforms (Table 2). 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".

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 1. Architectural characteristics of four evaluation platforms. [1]One GPU per node. [2]CUDA cores. [3] without TurboBoost.

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

Table 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

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 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 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

Table 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]

of MPI tasks. Unlike Edison where aprun controls affinity, one must use the "KMP_AFFINITY" environment variable on Babbage. We set it to "scatter" to distribute threads across the chip. On Titan, we once again use the aprun options. However, as we don't use the CPU cores, there was no need to control CPU thread affinity or NUMA bindings.

4 Memory and Cache Bandwidth

Today, bandwidth and data movement are perhaps the paramount aspect of performance 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. Thus, programmers wishing to proxy a flat MPI code and run the Roofline benchmark in a flat MPI model. Those wishing to understand the performance on NUMA architectures can run in the hybrid mode.

4.1 Bandwidth Code

Like CacheBench, our Roofline bandwidth benchmark 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 form thread parallelism and finite NoC bandwidth. In that regime, it is similar to STREAM code [18] which uses the OpenMP work-share constructs to split loop iterations across multiple threads (Fig. 1). Rather than using the work-share construct, our Roofline bandwidth code creates a single parallel region and statically assigns threads to ranges of array indices. All initialization, synchronization, and computation takes place within this parallel region. The computation is expressed as the sum of a finite geometric series as it was hoped that no compiler could automatically eliminate this nested loop. Essentially each term in the geometric series is a trial in the STREAM benchmark.

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

```
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 * 0.5;
}}

int main(){
  ...
  #pragma omp parallel private(id)
  {
  uint64_t n, t;
  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);
      }
}}}}
```

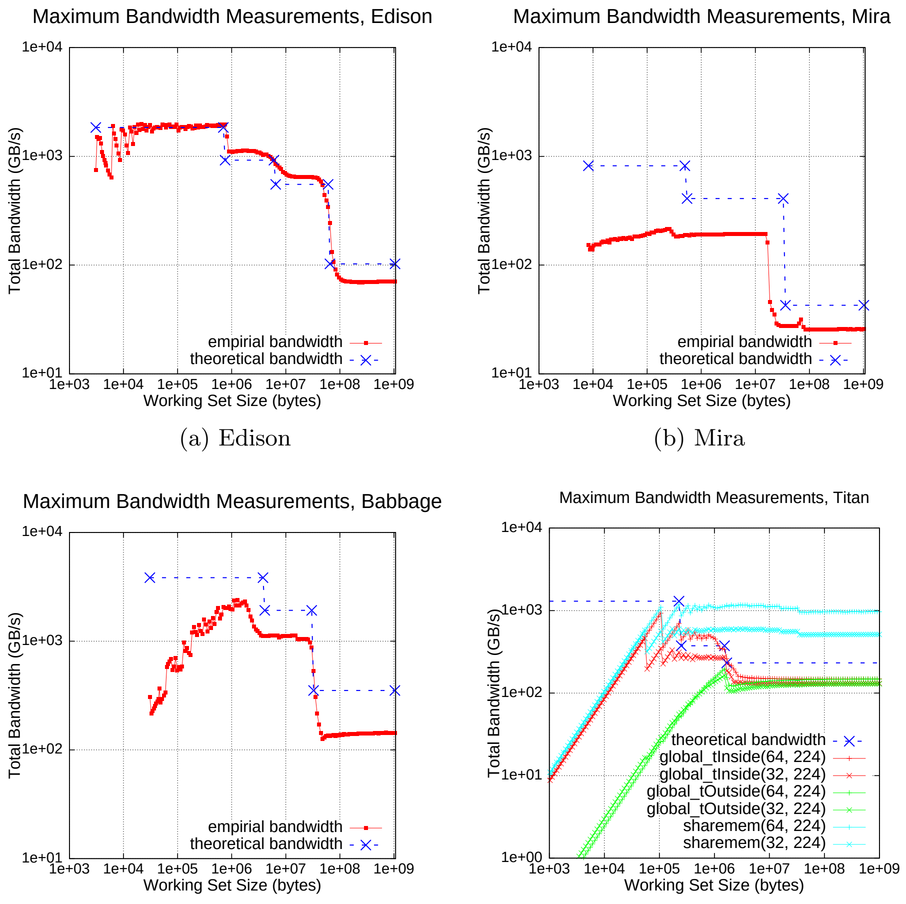
Fig. 1. (left) STREAM facsimile. (right) Roofline Bandwidth Benchmark.

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.

4.2 Bandwidth Result

Figure 2 presents the results of our Roofline bandwidth benchmark running on our four platforms. On Edison, we run two processes per node, while all other machines run with a single process. Note, the x-axis represents the total working set summed across all threads. 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 Roofline bandwidth.

We observe that on Edison, the hardware comes very close to the theoretical performance and transitions at the expected cache capacities. The smooth transitions in bandwidth at the cache capacities suggest the cache replacement policy may not be a true LRU or FIFO but a pseudo-variant. The notable exception is that Edison fails to come close to the DRAM pin bandwidth. This is not necessarily surprising as few machines have such high bandwidth and few machines ever attain the pin bandwidth. Moreover, the simple read-modify-write memory access pattern may be suboptimal for this architecture. Future work will explore alternate kernels that change the balance between reads and writes.



(a) Edison (b) Mira

(c) Babbage (MIC only) (d) Titan (GPU only)

Fig. 2. 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).

On Mira, performance was consistently below the theoretical bandwidth limits and the transitions seemed to indicate reduced effective cache capacities. The low L1 bandwidth was particularly surprising and may indicate the presence of a write-through or store-through L1 architecture. Further investigation is required.

On the highly-multithreaded MIC (Babbage), we found it was necessary to operate on working sets exceeding 1MB (well 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, using the GPU, we found it illustrative to run three slightly different kernels designed to quantify the effects of explicit and implicit reuse within the GPU's memory hierarchy. Both Kernel A ("global_tInside" legend on the Fig. 2 (d)) and Kernel B ("global_tOnside") use global memory, but with the trials loop inside and outside, respectively. Kernel C ("sharemem') copies global memory data to shared memory, does trials loop inside the kernel, and copies back to global memory.

"Kernel B" 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. "Kernel A" 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, "Kernel C" 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.

5 Floating-Point Compute Capability

Although many applications are limited by memory bandwidth, there are some that are still limited by on-chip computation and ultimately the in-core performance. When performance is on the cusp, 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 manycore and accelerated architectures. To address this deficiency, we constructed a Roofline floating-point benchmark.

5.1 Reference Roofline Floating-point Benchmark

We modified the Roofline bandwidth benchmark to implement a polynomial for each element. By varying the degree of the polynomial (a preprocessor macro), one can vary the number of flops per element. Doing so allows one to change the balance between loads/stores and floating-point operations from L1-limited to flop-limited. Fig. 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 rather than peak performance. A compiler could unroll this loop (at least by the floating-point latency) and express instruction-level parallelism 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. Although, an out-of-order parallelism could reorder the instruction stream, it can never automatically SIMDize the instruction stream. As such, without compiler support for SIMD, it can never attain peak performance.

```
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 < nsize; ++i) {
      double beta = 0.8;
      #if FLOPPERITER == 2
      beta = beta * A[i] + alpha;
      #elif FLOPPERITER == 4
      ...
      #endif
      A[i] = beta;
    }
  alpha = alpha * 0.5;
}
```

Fig. 3. Roofline Floating-Point Benchmark

5.2 Performance as a Function of Implicit and Explicit Parallelism

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. In order 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 Roofline floating-point benchmark — AVX, QPX, and AVX-512 versions. Fig. 4 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 expected. Although the machine is sensitive to instruction-level parallelism (unrolling), it generally does not require HyperThreading to attain good performance.

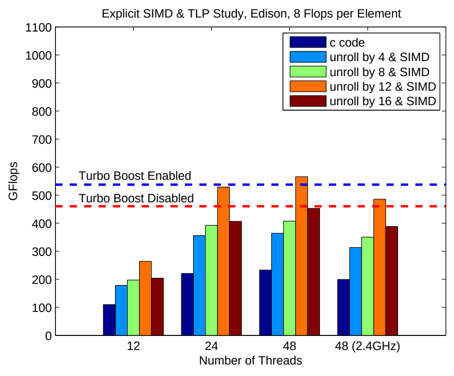
Running a similar set of experiments on Mira (BGQ), we see a very different outcome. First, compiled code delivers very good performance. This indicates that the XL/C compiler was able to effectively SIMDize and unroll the code sufficiently to hide the floating-point latency. Using explicitly unrolled 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.

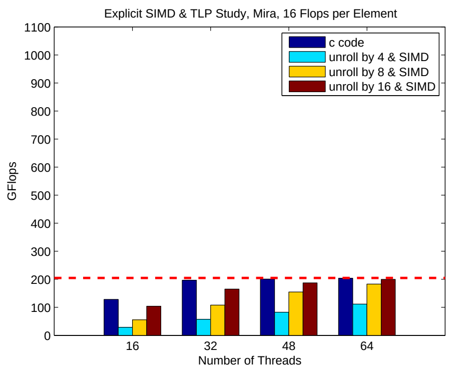
5.3 Performance as a function of L1 Arithmetic Intensity

Even when one can maintain a working set in the L1, 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 5 presents the resultant performance on each architecture. For reference, we include (in blue) a 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.

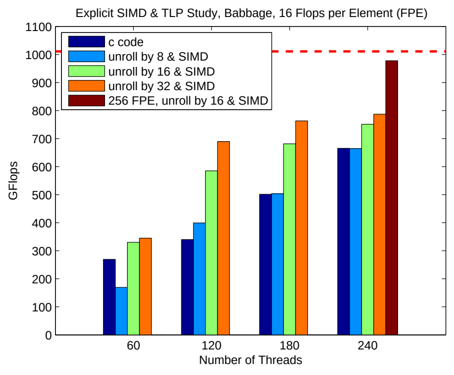
Figure 5 demonstrates that Edison can quickly reach its peak performance and that performance tracks well with the theoretical model. Generally, speaking, at low FPE, performance is diminished due to the fact that the core can perform



(a) Edison



(b) Mira



(c) Babbage

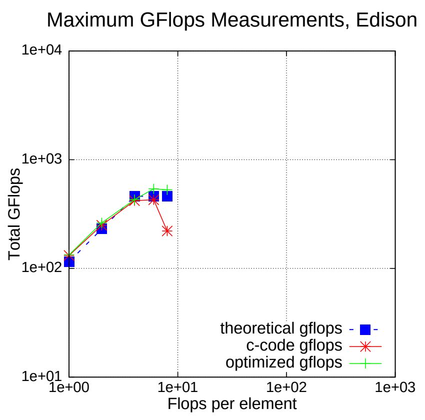
Fig. 4. Performance disparity between compiled code and optimized code in which thread-, instruction-, and data-level parallelism have been made explicit.

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.

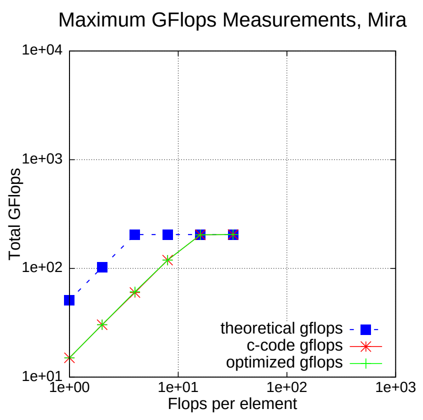
Mira's performance on both compiled and optimized code is shifted to the right. Generally, this suggests that additional instructions are consuming the same issue slots as loads or stores. On the dual issue A2 architecture, this could very well be integer or branch instructions. This effect was not present 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.

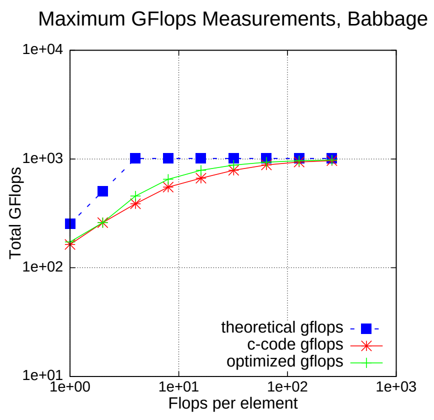
Finally, we constructed a similar CUDA C benchmark to run on the GPU. The theoretical bound is based on the assumption that each load/store unit can sustain loading 4 bytes per cycle (128 per SMX) from memory. We observe that the GPU's performance seems to embody characteristics of both BGQ and MIC. 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.



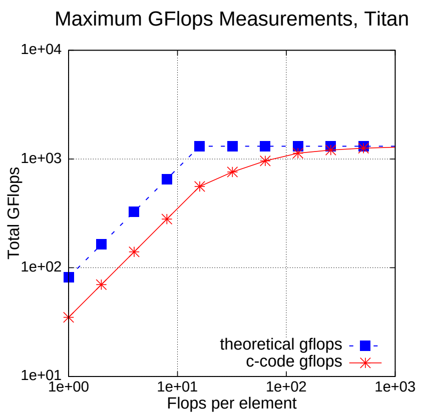
(a) Edison



(b) Mira



(c) Babbage (MIC only)



(d) Titan (GPU only)

Fig. 5. Basic GFlops code and optimized SIMDized unrolling GFlops code compared to theoretical GFlops on four platforms.

6 Beyond the Roofline — CUDA's Unified Memory

To date, accelerated architectures have been typically used as an accelerator with dedicated memory attached to a conventional system with 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 device via a library interface. Doing so is not only unproductive, but also exposes the performance disparity between PCIe bandwidth and device bandwidth.

Recently, CUDA introduced two memory concepts — the Unified Virtual Address (UVA) space, and Unified Memory (i.e. managed memory) [8]. 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 remains 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.

6.1 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.

In this paper, 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 is 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.

Fig. 6 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 host, and then uses `cudaHostRegister()` to register a device pointer on host memory address space. For unified memory, one uses `cudaMallocManaged` to allocate both host and device memory.

```
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);
}
```

Fig. 6. CUDA Unified Memory Benchmark quantifies the ability of the run time to mange locality on the device

6.2 Results

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

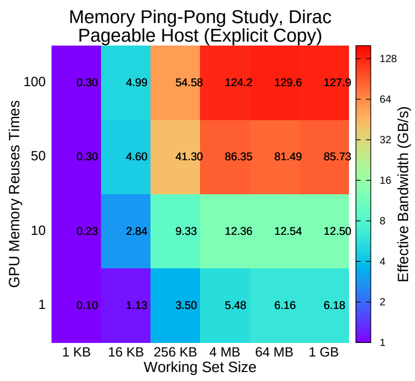
Fig. 7 presents the resultant "effective bandwidth" for the four technologies as a function of working set size and temporal reuse. For small working set sizes, CUDA kernel launch time dominates and effective bandwidth is abysmal. This simply reinforces the conventional wisdom not to use the GPU for small operations. Comparing Fig. 7(a) and (b), we see that it is possible to approach the device bandwidth limit, but only for 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 Fig. 7(c) from increased locality. Conversely, Fig. 7(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

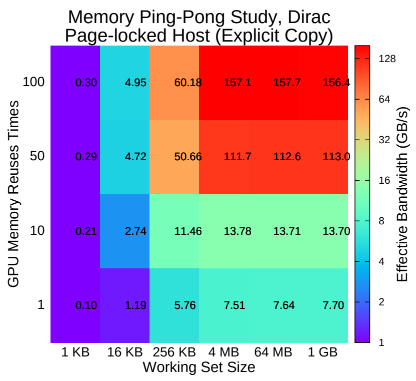
[1] GPU driver version: 331.89; CUDA toolkit version: 6.0beta.

high performance solution. One can only hope that advances in hardware and runtime can bridge the performance gap at lower temporal locality.

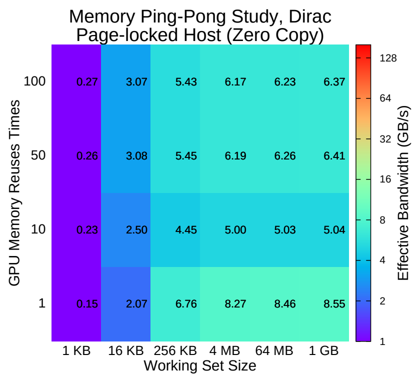
Future work will extend this technology to track the development of any software cache coherency protocol NVIDIA implements. That is, there is no reason why both the CPU and GPU must both read-modify-write the array. Either could perform a read-only operation.



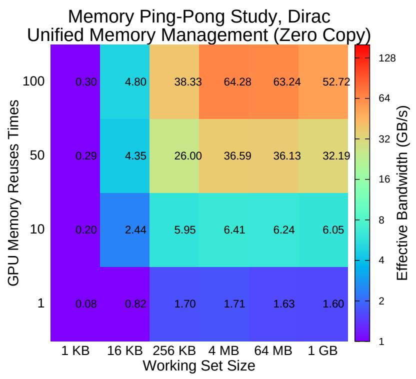
(a) Pageable host with explicit copy between CPU and GPU



(b) Page-locked host with explicit copy between CPU and GPU



(c) Page-locked host with zero copy



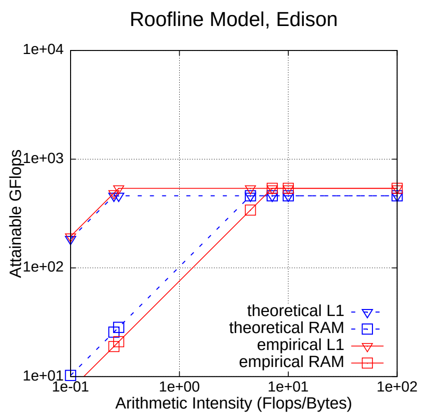
(d) Unified (managed) memory

Fig. 7. Effective bandwidth as a function of GPU temporal locality (reuse) and working set size for four different GPU device memory management mechanisms.

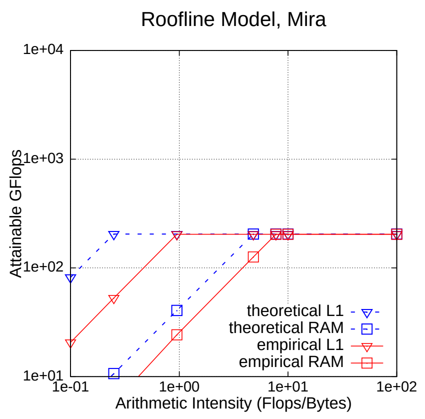
7 Empirical Roofline Models and Their Use

Now that we have benchmarked the bandwidth and compute characteristics on each of our four platforms, we may construct empirical Roofline Models for each.

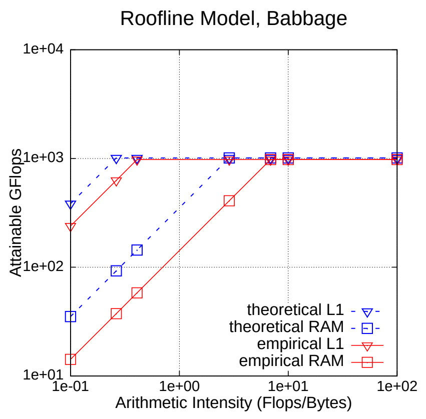
Figure 8 shows the resultant models using both DRAM and L1 bandwidths as well as the theoretical or “textbook” Roofline for each platform. An ideal architecture is one that can fully exploit the technology on which it is built. We see that in general, Edison’s empirical performance is very close to its 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.



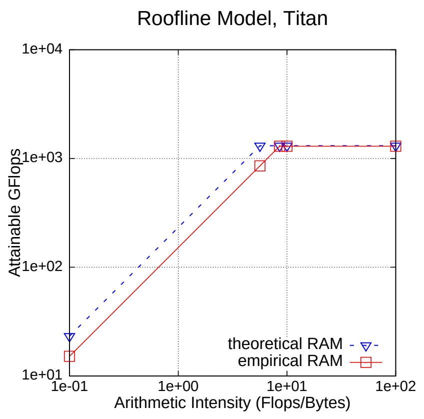
(a) Edison



(b) Mira



(c) Babbage



(d) Titan (GPU-only)

Fig. 8. Roofline model for four platforms.

7.1 Program Analysis

We use the resultant empirical Rooflines to analyze observed performance on three HPC benchmarks — the finite-volume High-Performance Geometric Multigrid (HPGMG-FV) benchmark [14], the Gyrokinetic Toroidal Code (GTC) [12], and miniDFT [17]. All benchmarks were run on Mira where the performance counters have been verified.

HPGMG-FV is a highly optimized multigrid benchmark that solves a variable coefficient Poisson's equation on a structured grid. Fig. 9(a) shows that it has low compute intensity, but it delivers performance, whether flat MPI or OpenMP is DRAM, very close to its bandwidth limit.

GTC is a turbulent transport fusion simulation that uses the particle-in-cell (PIC) method. 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. Clearly, the performance of both routines is well below the roofline suggesting optimization could significantly improve it.

MiniDFT code uses plane-wave density functional theory (DFT) to compute the Kohn-Sham equations, part of the general-purpose Quantum Espresso (QE) code. This is a compute-intensive code, dominated by dense linear algebra and 3D FFT's (Fig. 9 (b)). 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. While flat MPI performance generally tracked the Roofline, the performance of the threaded code was orders of magnitude less than ideal perhaps due to limited parallelism in any one dimension. Further investigation is warranted.

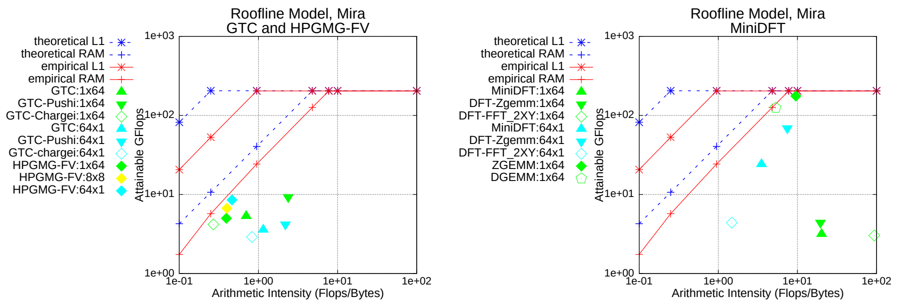
8 Summary

In this paper, we have described a prototype architecture characterization engine for the Roofline Toolkit that quantifies the bandwidth and compute characteristics of multicore, manycore, and accelerated systems. We use the Toolkit to benchmark four leading HPC systems: Edison, Mira, Babbage, and Titan. The measurements demonstrate the ability of each architecture to attain peak bandwidth or performance and quantify its sensitivity to changes in parallelism or arithmetic intensity.

In order to quantify the benefits of the emerging software managed cache technologies in CUDA, we developed a benchmark that measures the performance of CUDA's Unified memory as a function of spatial and temporal locality. Although performance never reaches parity with explicitly managed locality, performance was far superior to the productive Zero Copy alternative.

Finally, we evaluated three complex HPC compputing benchmarks: HPGMG-FV, GTC, and miniDFT running on Mira (BGQ). Using the HPM performance counters, we plotted benchmark performance on an empirical Roofline model in order to quantitatively note which applications deliver and which underperform.

Future work will continue to generalize the Roofline toolkit as well as continued instrumentation, benchmarking, and analysis of HPC applications in order to explore performance and parallelism issues on emerging HPC platforms.



(a) GTC and HPGMG-FV (b) MiniDFT

Fig. 9. GTC, HPGMG-FV, and MiniDFT results on Mira collected from BGQ performance counters. Legeneds denote “benchmark: number of MPI tasks x number of OpenMP threads.”

Acknowledgments

Authors from Lawrence Berkeley National Laboratory were supported by the U.S. Department of Energy’s Advanced Scientific Computing Research Program under contract DE-AC02-05CH11231. 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

References

1. Babbage Testbed. `https://www.nersc.gov/users/computational-systems/testbeds/babbage/`.
2. David H. Bailey, Robert F. Lucas, and Samuel W. Williams. *Performance Tuning of Scientific Applications*. CRC Press, New York, 2011.
3. Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. *IEEE IPDPS*, May 2013.

4. Cori Cray XC30. `https://www.nersc.gov/users/computational-systems/nersc-8-system-cori//`.
5. IBM Corporation. Ibm system blue gene solution: Blue gene/q application development. *IBM*, June 2013.
6. Intel Corporation. Intel xeon phi corprocessor system softeare developers guide. *Intel*, June 2012.
7. Nvidia Corporation. Kepler gk 110: The fatest, most efficient hpc architecture ever built. *Nvidia v1.0*, 2012.
8. Nvidia Corporation. Cuda c programming guide. *Nvidia PG-02819 v6.0*, Feb. 2014.
9. Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 2009.
10. Dirac Testbed. `http://www.nersc.gov/users/computational-systems/testbeds/dirac/`.
11. Edison Cray XC30. `http://www.nersc.gov/systems/edison-cray-xc30/`.
12. Gyrokinetic Toroidal Code Website. http://phoenix.ps.uci.edu/GTC/.
13. Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multicore chips via simple machine models. *CoRR abs/1208.2908*, 2012.
14. HPGMG website. `http://hpgmg.org`.
15. Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. *ACM MSP*, 2005.
16. LLCBench - Low Level Architectural Characterization Benchmark Suite. `http://icl.cs.utk.edu/projects/llcbench/index.htm`.
17. QEforge website: MiniDFT. http://qe-forge.org/gf/project/minidft/.
18. STREAM benchmark. `http://www.cs.virginia.edu/stream/ref.html`.
19. S. Williams. *Auto-tuning Performance on Multicore Computers.* PhD thesis, EECS Department, University of California, Berkeley, December 2008.
20. S. Williams, A. Watterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, April 2009.