Implementation and Optimization of miniGMG — a Compact Geometric Multigrid Benchmark

Samuel Williams[1], Dhiraj D. Kalamkar[2], Amik Singh[3],
Anand M. Deshpande[2], Brian Van Straalen[1], Mikhail Smelyanskiy[2],
Ann Almgren[1], Pradeep Dubey[2], John Shalf[1], Leonid Oliker[1]

[1] *Lawrence Berkeley National Laboratory*
[2] *Intel Corporation*
[3] *University of California Berkeley*
swwilliams@lbl.gov

December 2012

Abstract

Multigrid methods are widely used to accelerate the convergence of iterative solvers for linear systems used in a number of different application areas. In this report, we describe miniGMG, our compact geometric multigrid benchmark designed to proxy the multigrid solves found in AMR applications. We explore optimization techniques for geometric multigrid on existing and emerging multicore systems including the Opteron-based Cray XE6, Intel Sandy Bridge and Nehalem-based Infiniband clusters, as well as manycore-based architectures including NVIDIA's Fermi and Kepler GPUs and Intel's Knights Corner (KNC) co-processor. This report examines a variety of novel techniques including communication-aggregation, threaded wavefront-based DRAM communication-avoiding, dynamic threading decisions, SIMDization, and fusion of operators. We quantify performance through each phase of the V-cycle for both single-node and distributed-memory experiments and provide detailed analysis for each class of optimization. Results show our optimizations yield significant speedups across a variety of subdomain sizes while simultaneously demonstrating the potential of multi- and manycore processors to dramatically accelerate single-node performance. However, our analysis also indicates that improvements in networks and communication will be essential to reap the potential of manycore processors in large-scale multigrid calculations.

NOTE, this report represents an expanded version of our Supercomputing 2012 paper [30].

1 Introduction

In the past decades, continued increases in clock frequencies have delivered exponential improvements in computer system performance. However, this trend came to an abrupt end a few years ago as power consumption became the principal rate limiting factor. As a result, power constraints are driving architectural designs towards ever-increasing numbers of cores, wide data parallelism, potential heterogenous acceleration, and a decreasing trend in *per-core* memory bandwidth. Understanding how to leverage these technologies in the context of demanding numerical algorithms is likely the most urgent challenge in high-end computing.

In this report, itself an expanded version of [30], we present miniGMG, a compact geometric multigrid (GMG) benchmark. We explore the optimization of miniGMG on a variety of leading multi- and manycore architectural designs. Our primary contributions are:

- We examine a broad variety of leading multicore platforms, including the Opteron-based Cray XE6, Intel Sandy Bridge and Nehalem-based Infiniband clusters, as well as NVIDIA's Fermi and Kepler GPUs and Intel's Knights Corner (KNC) manycore co-processor. This is the first study to examine the performance characteristics of the recently announced KNC architecture.
- We optimize and analyze all the required components within an entire multigrid V-cycle using a variable-coefficient, Red-Black, Gauss-Seidel (GSRB) relaxation on these advanced platforms. This is a significantly more complex calculation than exploring just the stencil operation on a large grid.
- We implement a number of effective optimizations geared toward bandwidth-constrained, wide-SIMD, manycore architectures including the application of wavefront to variable-coefficient, Gauss-Seidel, Red-Black (GSRB), SIMDization within the GSRB relaxation, and intelligent communication-avoiding techniques that reduce DRAM traffic. We also explore message aggregation, residual-restriction fusion, nested parallelism, as well as CPU-, KNC-, and GPU-specific tuning strategies.
- Additionally, we proxy the demanding characteristics of real simulations, where relatively small subdomains (32^3 or 64^3) are dictated by the larger application. This results in a broader set of performance challenges on manycore architectures compared to using larger subdomains.

Overall results show a significant performance improvement of up to 3.8× on KNC compared with the parallel baseline implementation (highlighted in Figure 7), while demonstrating scalability on up to 24K cores on the XE6. Additionally, miniGMG highlights the challenges of exploiting GPU-accelerated architectures on highly-efficient algorithms. Finally, our performance analysis with respect to the underlying hardware features provides critical insight into the approaches and challenges of effective numerical code optimization for highly-parallel, next-generation platforms.

2 Multigrid Overview

Multigrid methods provide a powerful technique to accelerate the convergence of iterative solvers for linear systems and are therefore used extensively in a variety of numerical simulations. Conventional iterative solvers operate on data at a single resolution and often require too many iterations to be viewed as computationally efficient. Multigrid simulations create a hierarchy of grid levels and use corrections of the solution from iterations on the coarser levels to improve the convergence rate of the solution at the finest level. Ideally, multigrid is an O(N) algorithm; thus, performance optimization on our studied multigrid implementation can only yield constant speedups.

Figure 1 shows the three phases of the multigrid V-cycle for the solve of $Lu^h = f^h$. First, a series of *smooths* reduce the error while *restrictions* of the residual create progressively coarser grids. The *smooth* is a conventional relaxation such as Jacobi, successive over-relaxation (SOR), or Gauss-Seidel, Red-Black (GSRB) which we used in our study as it has superior convergence properties. The restriction of residual ($f^h - Lu^h$) is used to define the right-hand side at the next coarser grid. At each progressively coarser level, the correction (e.g. u^{2h}) is initialized to zero. Second, once coarsening stops (the grid size reaches one or terminated for performance), the algorithm switches to a bottom solver which can be as simple as applying multiple relaxes or as complicated as an algebraic multigrid or direct sparse linear solver. Finally, the coarsest correction is interpolated back up the V-cycle to progressively finer grids where it is smoothed.

Nominally, one expects an order of magnitude reduction in the residual per V-cycle. As each level performs O(1) operations per grid point and $\frac{1}{8}$ the work of the finer grid, the overall computation is O(N) in the number of variables in u. The linear operator can be arbitrarily complex as dictated by the underlying physics, with a corresponding increase in run time to perform the smooth computation. Section 4.2 details the problem specification used in our study.

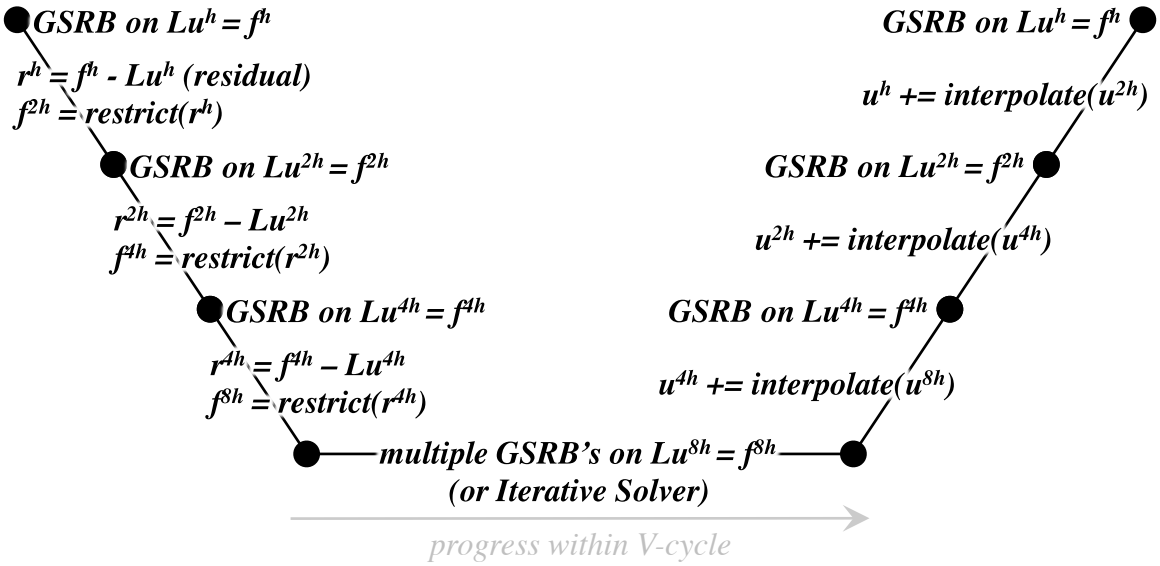

Figure 1: The Multigrid V-cycle for solving $Lu^h = f^h$. Superscripts represent grid spacing. Restriction (coarsening) is terminated at $8h$. Nominally, a high-performance, iterative solver is employed at the bottom.

3 Related Work

Throughout this report, we leverage the 3C's taxonomy when referring to cache misses [14]. In the past, operations on large structured grids could easily be bound by capacity misses, leading to a variety of studies on blocking and tiling optimizations [9, 10, 16, 21, 22, 29, 31]. However, a number of factors have made such approaches progressively obsolete on modern platforms. On-chip caches have grown by orders of magnitude and are increasingly able to capture sufficient locality for the fixed box sizes associated with typical multigrid methods. The rapid increase in on-chip parallelism has also quickly out-striped available DRAM bandwidth resulting in bandwidth-bound performance.

Thus, in recent years, numerous efforts have focused on increasing temporal locality by fusing multiple stencil sweeps through techniques like cache oblivious, time skewing, or wavefront [8, 11, 12, 17, 19, 24, 27, 32–34]. Many of these efforts examined 2D or constant-coefficient problems — features rarely seen in real-world applications.

Chan et al. explored how, using an auto-tuned approach, one could restructure the multigrid V-cycle to improve time-to-solution in the context of a 2D, constant-coefficient Laplacian [5]. This approach is orthogonal to our implemented optimizations and their technique could be incorporated in future work.

Studies have explored the performance of algebraic multigrid on GPUs [1, 2], while Sturmer et al. examined geometric multigrid [25]. Perhaps the most closely related work is that performed in Treibig's, which implements a 2D GSRB on SIMD architectures by separating and reordering the red and black elements [26], additionally a 3D multigrid on an IA-64 (Itanium) is implemented via temporal blocking. This report expands on these efforts by providing a unique set of optimization strategies for multi- and manycore architectures.

4 Experimental Setup

4.1 Evaluated Platforms

We use the following systems in all our experiments. Their key characteristics are summarized in Table 1.

Cray XE6 "Hopper": Hopper is a Cray XE6 MPP at NERSC built from 6384 compute nodes each consisting of two 2.1 GHz 12-core Opteron (MagnyCours) processors [15]. In reality, each Opteron socket is comprised of two 6-core chips each with two DDR3-1333 memory controllers. Effectively, the compute nodes are comprised of four (non-uniform memory access) NUMA nodes, each providing about 12 GB/s of STREAM [18] bandwidth. Each core uses 2-way SSE3 SIMD and includes both a 64KB L1 and a 512KB L2 cache, while each socket includes a 6MB L3 cache with 1MB reserved for the probe filter. The compute nodes are connected through the Gemini network into a 3D torus.

Nehalem-Infiniband Cluster "Carver": The Carver cluster at NERSC is built from 1202 compute nodes mostly consisting of two 2.66 GHz, quad-core Intel Xeon X5550 processors [4]. Thus, each compute node consists of two NUMA nodes. Each quad-core Nehalem (NHM) socket includes an 8 MB L3 cache and three DDR3 memory controllers providing about 18 GB/s of STREAM bandwidth. Each core implements the 2-way SSSE3 SIMD instruction set and includes both a 32KB L1 and a 256KB L2 cache. HyperThreading is disabled on Carver. The compute nodes are connected through the 4X QDR Infiniband network arranged into local fat trees and a global 2D mesh.

Sandy Bridge-Infiniband Cluster "Gordon": The Gordon cluster at the San Diego Supercomputing Center is comprised of 1024 compute nodes each with two 2.6 GHz, 8-core Intel Xeon E5 2670 processors [13]. Each 8-core Sandy Bridge (SNBe) processor includes a 20 MB L3 cache and four DDR3-1333 memory controllers providing about 35 GB/s of STREAM bandwidth. Each core implements the 4-way AVX SIMD instruction set and includes both a 32KB L1 and a 256KB L2 cache. This provides Gordon with four times the peak performance and twice the sustained bandwidth as Carver. HyperThreading is disabled on Gordon. The compute nodes are connected through the 4X QDR Infiniband network with switches arranged into a torus.

Intel Knights Corner: KNC is an Intel MIC architecture platform. Intel MIC is an x86-based, many-core processor architecture based on small, in-order cores that uniquely combines the full programmability of today's general purpose CPU architecture with the compute throughput and memory bandwidth capabilities of modern GPU architectures. As a result, standard parallel programming

approaches like Pthreads or OpenMP apply to KNC — a potential boon to portability. Each core is a general-purpose processor that includes a scalar unit based on the Pentium processor design and a vector unit that may perform eight 64-bit floating-point or integer operations per clock. The KNC[§] pipeline is dual-issue: scalar instructions can pair and issue in the same cycle as vector instructions. To further hide latency, each core is 4-way multithreaded. This provides KNC with a peak double-precision performance of 1.2 TFLop/s and STREAM bandwidth of 150 GB/s (with ECC enabled). KNC has two levels of cache: a low latency 32KB L1 data cache and a larger, globally-coherent L2 cache that is partitioned among the cores, with 512KB per core. All KNC experiments were run in *native mode* in which the memory hierarchy and thread execution model are MIC-centric. Thus, on KNC, heterogeneity is invisible to the programmer.

NVIDIA M2090 Fermi: The M2090 Fermi GPU includes 512 scalar "CUDA cores" running at 1.30 GHz and grouped into sixteen SIMT-based streaming multiprocessors (SM). This provides a peak double-precision floating-point capability of 665 GFlop/s. Each SM includes a 128 KB register file and a 64 KB SRAM that can be partitioned between cache and "shared" memory in a 3:1 ratio. Although the GPU has a raw pin bandwidth of 177 GB/s to its on-board 6 GB of GDDR5 DRAM, the measured bandwidth with ECC enabled is about 120 GB/s. ECC is enabled in all our experiments.

NVIDIA K20x Kepler: The K20x Kepler GPU includes 2688 scalar "CUDA cores" running at 0.733 GHz and grouped into fourteen SIMT-based streaming multiprocessors (SMX). This provides a peak double-precision floating-point capability of 1313 GFlop/s. Each SMX includes a 256 KB register file and a 64 KB SRAM that can be partitioned between cache and "shared" memory. The measured bandwidth with ECC enabled to the 6GB of GDDR memory is about 165 GB/s. ECC is enabled in all our experiments.

Core Architecture	**AMD Opteron**	**Intel NHM**	**Intel SNBe**	**Intel KNC[§]**	**NVIDIA Fermi**	**NVIDIA Kepler**
Clock (GHz)	2.10	2.66	2.60	1.30	1.30	0.733
DP GFlop/s	8.40	10.66	20.80	20.80	83.2[1]	93.8[2]
D$ (KB)	64+512	32+256	32+256	32+512	64[1]	64[2]
Memory-Parallelism	HW-prefetch	HW-prefetch	HW-prefetch	HW & SW prefetch	multi-threading	multi-threading
Node Architecture	**Cray XE6**	**Xeon X5550**	**Xeon E5 2670**	**MIC KNC[§]**	**Tesla M2090**	**Tesla K20x**
Cores/chip	6	4	8	60	16[1]	14[2]
Last Level Cache/chip	5 MB	8 MB	20 MB	—	768 KB	1.5MB
Chips/node	4	2	2	1	1	1
DP GFlop/s	201.6	85.33	332.8	1248	665.6	1313
STREAM[3]	49.4 GB/s	38 GB/s	70 GB/s	150 GB/s	120 GB/s	165 GB/s
Memory	32 GB	24 GB	64 GB	4 GB	6 GB	6 GB
System	**Hopper**	**Carver**	**Gordon**	—	—	**Titan**
Interconnect	Gemini 3D Torus	InfiniBand Fat Tree	InfiniBand 3D Torus	—	—	Gemini 3D Torus
Programming Model	MPI+OMP	MPI+OMP	MPI+OMP	OMP	CUDA	CUDA
Compiler	icc 12.1.2	icc 12.1.2	icc 12.1.2	icc 13.0.036	4.0	5.0

Table 1: **Overview of Evaluated Platforms**. [1]*Each shared multiprocessor (SM) is one "core" and includes 32K thread-private registers and 64KB of shared memory + L1 cache.* [2]*Each shared multiprocessor (SMX) is one "core" and includes 256KB of thread-private registers and 64KB of shared memory + L1 cache.* [3]*STREAM copy on CPUs, SHOC [7] on ECC-enabled GPU.*

[§]Evaluation card only and not necessarily reflective of production card specifications.

4.2 miniGMG Problem Specification

A key goal of this report is to analyze the computational challenges of multigrid in the context of multi- and manycore, optimization, and programming model. We therefore constructed a compact multigrid solver benchmark (miniGMG) that creates a global 3D domain partitioned into subdomains sized to proxy those found in real multigrid applications. We also explore the use of 32^3 and 128^3 subdomains to estimate the ultimate performance of memory bandwidth-constrained multigrid codes. The resultant list of subdomains is then partitioned among multiple MPI processes on platforms with multiple NUMA nodes. All subdomains (whether on the same node or not) must explicitly exchange ghost zones with their neighboring subdomains, ensuring an effective communication proxy of multigrid codes.

We use a double-precision, finite volume discretization of the variable-coefficient operator $L = a\alpha I - b\nabla\beta\nabla$ with periodic boundary conditions as the linear operator within our test problem. Variable-coefficient is an essential (yet particularly challenging) facet as most real-world applications demand it. The right-hand side (f) for miniGMG is $sin(\pi x)sin(\pi y)sin(\pi z)$ on the [0,1]^3 cubical domain. The u, f, and α are cell-centered data, while the β's are face-centered.

To enable direct time-to-solution comparisons of different node architectures, we fix the problem size to a 256^3 discretization on all platforms. This (relatively small) grid size in conjunction with partitioning into subdomains, the variable-coefficient nature of our computation, and buffers required for exchanging data, consumes more than 2.5GB — sufficiently small for both KNC and GPUs. Our baseline for node comparison is the performance of the 4-chip XE6 node, the 2-chip Intel Xeon nodes, the single chip KNC co-processor, and the GPU accelerators solving one 256^3 problem.

To allow for uniform benchmarking across the platforms, we structure a truncated V-cycle where restriction stops at the 4^3 level (the "bottom" level). We fix the number of V-cycles at 10 and perform two relaxations at each level down the V-cycle, 24 relaxes at the bottom, and two relaxations at each level up the V-cycle. As this report is focused on optimization the multigrid V-cycle, a simple relaxation scheme at the bottom is sufficient to attain single-node multigrid convergence.miniGMG includes both CG and BiCGStab bottom solvers whose performance allow scalable multigrid implementations.

Our relaxation scheme uses Gauss-Seidel Red-Black (GSRB) which offers superior convergence compared to other methods. It consists of two grid smooths per relax each updating one color at a time, for a total of eight smooths per subdomain per level per V-cycle. The pseudocode for the resultant inner operation is shown in Figure 2. Here, neither the Laplacian nor the Helmholtz (Identity minus Laplacian) of a grid is ever actually constructed, rather all these operations are fused together into one GSRB relaxation. A similar calculation is used for calculating the residual. Nominally, these operators require a one element deep ghost zone constructed from neighboring subdomain data. However, in order to leverage communication aggregation and communication avoiding techniques, we also explore a 4-deep ghost zone that enables one to avoid DRAM data movement at the expense of redundant computation. Although the CPU code allows for residual correction form, it was not employed in our experiments; observations show its use incurs a negligible impact on performance.

The data structure for a subdomain within a level is a list of equally-sized grids (arrays) representing the correction, right-hand side, residual, and coefficients each stored in a separate array. Our implementations ensure that the core data structures remain relatively unchanged with optimization. Although it has been shown that separation of red and black points into separate arrays can facilitate SIMDization [26], miniGMG forbids such optimization as they lack generality and challenge other phases. As data movement is the fundamental performance limiter, separating red and black will provide little benefit as the same volume of data transfer is still required.

```
helmholtz[i,j,k] = a*alpha[i,j,k]*phi[i,j,k] - b*h2inv*(
  beta_i[i+1,j,k] * ( phi[i+1,j,k] - phi[i,j,k]   ) -
  beta_i[i,j,k]   * ( phi[i,j,k]   - phi[i-1,j,k] ) +
  beta_j[i,j+1,k] * ( phi[i,j+1,k] - phi[i,j,k]   ) -
  beta_j[i,j,k]   * ( phi[i,j,k]   - phi[i,j-1,k] ) +
  beta_k[i,j,k+1] * ( phi[i,j,k+1] - phi[i,j,k]   ) -
  beta_k[i,j,k]   * ( phi[i,j,k]   - phi[i,j,k-1] )
)

phi[i,j,k] = phi[i,j,k] -
  lambda[i,j,k] * ( helmholtz[i,j,k] - rhs[i,j,k] )
```

Figure 2: Inner operation for Gauss-Seidel Red-Black relaxation on a variable-coefficient Helmholtz operator, where `phi[]` is the correction. Note that seven arrays must be read, and one array written assuming the Helmholtz is not stored.

4.3 miniGMG Reference (Baseline) Implementation

The CPUs and KNC share a common reference implementation of miniGMG and is available from http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/xtune/miniGMG. miniGMG builds the multigrid solver by allocating and initializing the requisite data structures, forming and communicating restrictions of the coefficients, and performing multiple solves. Given that each NUMA node is assigned one MPI process, the single-node CPU experiments use four (XE6) or two (NHM, SNBe) MPI processes which collectively solve the one 256^3 problem (see Figure 3(left)). OpenMP parallelization is applied to the list of subdomains owned by a given process.

When using threads for concurrency, each thread operates independently on one subdomain at a time. For example, on Gordon with 64 subdomains per node, each of the 16 OpenMP threads will be responsible for four subdomains. The baseline KNC code uses the same approach via *native mode* with the caveat that only 64 threads scattered across the cores can be active. We acknowledge that this will underutilize and imbalance the 60-core, 240-thread machine. Nevertheless, this style of flat parallelism is designed to proxy today's multigrid frameworks like CHOMBO or BoxLib [3, 6], and serves as a common baseline. No communication avoiding, cache blocking, hierarchical threading, SIMDization, or other explicit optimizations are employed in the baseline OpenMP version making it highly portable.

Conversely, the GPU required substantial modifications to the baseline CPU code to realize a high-performance CUDA implementation. First, the baseline C code's grid creation routines were modified to allocate data directly in device memory on the GPU. Thus, no time is included for PCIe transfer times. Second, the core routines were modified from simple OpenMP into high-performance CUDA kernels. We leverage 3D CUDA grids with the z-dimension used to index the subdomain. The first two dimensions of the CUDA grid directly map to that of the subdomain. We stream through the k-dimension to maximize in-register or in-cache locality. We explored cache-based and shared memory-optimized implementations, explored favoring shared memory vs. favoring L1 (L1 was best), tuned the thread block aspect ratio to minimize the average number of cache lines accessed per element (32×8 was optimal), and compiled with `-dlcm=cg` to reduce the memory transaction from

128B to 32B. This further improved spatial locality. Finally, subdomain ghost zone exchanges were maximally parallelized and performed explicitly in device memory. Overall, this tuning resulted in more than a 10× improvement in performance over the initial CUDA implementation. Thus, the GPU implementation is highly-optimized CUDA.

4.4 Distributed Memory Experiments

To quantify the impact of network architecture and communication aggregation on the performance of large-scale V-cycle simulations, we conduct a series of weak scalability experiments, assigning a full 256^3 domain to each NUMA node. Thus, a $1K^3$ domain requires only 64 NUMA nodes — 16 compute nodes on Hopper and 32 nodes on Carver/Gordon. miniGMG supports GPUs in a distributed memory environment using MPI+CUDA. However, the performance implications of the PCIe bus can severely impede performance for many problem sizes.

5 Performance Challenges and Expectations

In miniGMG, there are five principal functions at each level: smooth, residual, restriction, interpolation, and exchange. Smooth is called eight times, Exchange nine, and the others once. In the baseline implementation of miniGMG, data dependencies mandate each function be called in sequence. Descending through the V-cycle, the amount of parallelism in smooth, residual, restriction, and interpolation decrease by a factor of eight, while working sets decrease by a factor of four. This creates an interesting and challenging interplay between intra- and inter-box parallelism.

Smooth: Nominally, smooth dominates the run time. This function performs the GSRB relax (stencil) on every other point (one color), and writes the updated correction back to DRAM. For a 64^3 subdomain with a 1-deep ghost zone, this corresponds to a working set of about 340KB (10 planes from seven different arrays), the movement of 2.3M doubles ($8 * 66^3$), and execution of 3.3M floating-point operations (25 flops on every other point). As a subsequent call to smooth is not possible without communication, its flop:byte ratio of less than 0.18 results in performance heavily bound by DRAM bandwidth. In fact, with 50 GB/s of STREAM bandwidth, Smooth will consume at least 1.88 seconds at the finest grid. Although GSRB is particularly challenging to SIMDize, its memory-bound nature avoids any performance loss without it.

Residual: Residual is very similar to smooth. However, it is out-of-place, and there is no scaling by the diagonal. Thus, the time required should be quite similar to each invocation of smooth. However, instead of being called eight times per V-Cycle like smooth, it is called only once per V-Cycle.

Restriction: Restriction reads in the residual and averages eight neighboring cells. This produces a grid (f^{2h}) nominally 8× smaller — a 64^3 grid is restricted down to 32^3 (plus a ghost zone). Such a code has a flop:byte ratio of just under 0.09 and is thus likely to be heavily bandwidth-bound.

Interpolation: Interpolation is the mirror image of restriction: each element of u^{2h} is used to increment eight elements of u^h. It typically moves almost twice as much data as restrict and has a flop:byte ratio less than 0.053.

Exchange Boundaries: For the single-node experiments, this function contains three loop nests. First, the (non-ghost zone) surface of each 3D subdomain is extracted and packed into 26 surface buffers representing the 26 possible neighboring subdomains. Second, buffers are exchanged between subdomains into a set of 26 ghost zone buffers. Due to the shape of our stencil, in the baseline implementation, each subdomain only communicates with its six neighboring subdomains. Finally, the data in the ghost zone buffers is copied into the subdomain grids. Although there are no floating-point operations in this routine, there is a great deal of data movement often with poor sequential

locality. In the MPI implementation, each subdomain determines whether its neighbor is on- or off-node, and uses either an `MPI_ISend` or a `memcpy()`.

6 Optimization

6.1 Communication Aggregation

In an MPI version, each subdomain can potentially initiate its own ISend/IRecv combination, thereby flooding the network. To rectify this, the baseline code aggregates and buffers off-node ghost-zone exchanges through 26 process-level Send and Receive buffers, thus keeping the message count constant regardless of the number of subdomains per process.

To improve communication performance, it is well known that transferring a deeper ghost zone can be used to reduce the number of communication phases at the expense of redundant computation required to produce a bit-identical result. That is, rather than four-rounds of smoothing each 64^3 grid, four smooths can be performed in one pass through a 70^3 grid; we therefore explore sending both a 1-element and a 4-element deep ghost zone. The latter necessitates communication with all 26 neighbors (faces, edges, and corners) instead of simply the six faces required by the 1-element ghost zone and our stencil. As we duplicate the work of neighboring subdomains, we are required to duplicate their values of the α and β coefficients when building the solver, and the right-hand side at each level of each V-cycle. Thus, communication aggregation can significantly increase inter-subdomain communication.

6.2 DRAM Communication Avoiding

Given that the core routines within the V-cycle are memory-bound, we explore opportunities to improve temporal locality and avoid DRAM data movement. Although the communication aggregation approach has the potential for improved locality in smooth, it requires an extremely large working set of 20MB (seven arrays $\times 70^3$) per subdomain at the finest resolution — likely too large to be exploited by our processors.

To minimize the working set and enable a streaming model, we implement a wavefront approach [34] to the Gauss-Seidel, Red-Black relax operation. As shown in Figure 3(right), our communication-avoiding approach constructs a wavefront of computation 4-planes deep (matching the number of relaxes) that sweep through the subdomain in the k-dimension (least unit stride). With four planes of seven arrays (plus two leading and one trailing plane), the aggregate working set of this technique is roughly $250 \times Dim_i \times Dim_j$, or approximately 1.23 MB at the finest grid. For the Nehalem and Sandy Bridge-class CPUs, this is small enough to allow each thread to operate on independent subdomains while still attaining locality in the L3 cache (see Table 1). For our other platforms, threading or blocking is necessary within a subdomain to reduce the working set sufficiently to fit into on-chip memory. However, as the computation descends through the V-cycle, the working set is naturally reduced to the point where it fits into the L2 or L1 cache. Ideally, the programmer or underlying architecture provides a mechanism to overlap DRAM-cache data transfers of the next plane (highlighted in blue Figure 3(right)) and the computation on the four trailing planes. Failure to do so will result DRAM overheads being amortized (rather than hidden) by multiple fast cache accesses.

Table 2 quantifies the theoretical benefits from improved locality and costs from increased grid sizes of communication avoiding on local operations at each level. The model is based on the volume of data movement (the 66^3 or 72^3 grids including ghost zones). Finite cache bandwidth, cache capacity, and in-core compute capability will limit the realized benefits attained without affecting the performance costs in residual, restriction, and interpolation. Despite the fact that grids nominally increase by at least 30% with the addition of a 4-deep ghost zone, kernels called multiple times like smooth can be

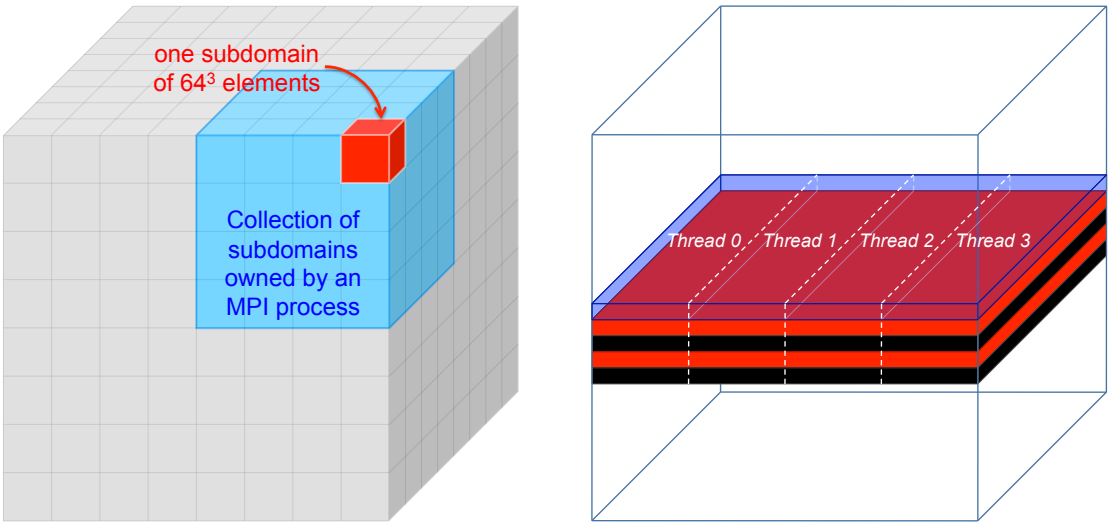

Figure 3: Visualization of the domain/process/subdomain hierarchy (left) and the threaded GSRB wavefront computation and prefetching strategy within each subdomain (right) for miniGMG. Note, each subdomain is comprised of multiple grids (correction, RHS, coefficients, etc...)

restructured to move this data once (instead of four times). Thus, at the finest grid, there is a potential 3× reduction in smooth run time at the cost of at least a 30% increase in residual, restriction, and interpolation time. Subsequent descent through the V-cycle, will reduce the advantage for smooth as an increasing volume of data is transferred per point (thick ghost zones dominate the volume). This crossover point is quantified in Section 7.2.

6.3 Residual-Restriction Fusion

The residual is immediately restricted to form the right-hand side for the next level in the V-cycle and then discarded. We therefore may fuse the residual and restriction into a one operation to avoid superfluous DRAM communication.

6.4 Correction Communication Avoiding

The correction is globally initialized to zero for all levels of the V-cycle except the finest. Thus when using two GSRB relaxes and a ghost-zone depth of 4, we can guarantee that in the one exchange of the correction, each subdomain will entirely receive zeros from its neighboring subdomains. Therefore we can eliminate communication of the correction for all coarsened levels and thus obviate this extraneous communication.

6.5 CPU-Specific Optimizations

The baseline CPU implementation uses MPI to parallelize subdomains across NUMA nodes combined with OpenMP to parallelize computation over the list of subdomains within each process. However, this form of concurrent execution within a process demands an aggregate cache working set proportional to the number of threads. Even when wavefront is applied to realize communication avoiding, this

operation	64^3	32^3	16^3	8^3	4^3
smooth	3.08×	2.46×	1.69×	0.98×	0.50×
residual+restriction	0.77×	0.61×	0.41×	0.24×	N/A
interpolation	0.75×	0.58×	0.38×	0.21×	N/A

Table 2: **Theoretical limits on speedups of DRAM communication-avoiding scheme compared to naive (assuming arbitrarily fast cores/caches) based on data movement of grid sizes with 1- or 4-deep ghost zones for the three principal operations in the V-Cycle. Note, the increased grid sizes resulting from a deeper ghost zone without any increase in temporal locality will make the residual, restriction, and interpolation operations slower.**

working set can reach nearly 8MB on the Opteron — far larger than the 5MB of L3 cache. To rectify this, we developed a collaborative (or threaded) wavefront for Smooth on grids 64^3 and larger, which dramatically reduces the cache working set to the point where locality of four planes may be maintained in the L2 cache. This is realized via the parallelization scheme shown in Figure 3(right) using an OpenMP parallel region in which loop bounds are statically calculated and a spin barrier is implemented to ensure threads proceed in lock-step. Parallelization in the j-Dimension via a `#pragma omp parallel for` or using an OpenMP barrier incurred too much overhead.

However, the threaded wavefront implementation presents a new challenge in the form of sequential locality. Each thread will access a contiguous block of data of size approximately $Dim_i * Dim_j/NThreads$ or roughly 5KB on SNBe. Unfortunately, hardware prefetchers can become ineffective on such short stanza accesses. To rectify this, we incorporated software prefetch intrinsics to prefetch the region highlighted in blue Figure 3(right) and interleave them with the computation on the four trailing planes. Essentially, one prefetch is generated per array for every 32 stencils — the product of the number of planes in the wavefront and the number of elements in a cache line. Although this is a crude approach to decoupling data movement from computation, it did provide a noticeable performance gain. Note that the collaborative threading approach has its limits, and is not used on 32^3 or smaller grids.

To maximize inner loop performance, a code generator was written in Perl to generate SSE or AVX SIMD intrinsics for all stencils within a pencil (all values of i for a given j, k coordinate). Since it is often impractical for multigrid developers to change data structures within full-scale applications, we preserve the interleaving of red and black within memory. Thus to SIMDize the GSRB kernel, we perform the kernel in a SIMD register as if it were Jacobi, but merge the original red or black value into the SIMD register via `blendvpd` before committing to memory. Although such an approach potentially wastes half the compute capability of a core, it incurs no more L1, L2, or DRAM bandwidth — the latter being a fundamental bottleneck for existing and future architectural designs.

Finally, the buffer exchange of the communication phase was optimized via with the cache bypass `movnt` intrinsic. This optimization remains important when descending through the V-cycle as the aggregate buffer size can remain quite large. We use OpenMP to parallelize over the list of subdomains. CPU-specific optimization of the bandwidth-bound interpolation or residual was deemed unnecessary.

6.6 KNC-Specific Optimizations

KNC optimization started with the common baseline code between CPU and KNC (plus a few KNC-specific compiler flags), and explored hierarchical parallelism on inter- and intra-subdomain levels, added SIMD intrinsics, and finally included memory optimizations to maximize cache locality and minimize conflict misses through array padding. Orthogonal to this optimization was the use of

communication aggregation and communication avoiding via wavefront.

The application of OpenMP in the baseline code presumes there is more parallelism across subdomains (64) than there is in hardware. While true on the CPUs, the domain-level parallelism is deficient by nearly a factor of four when targeting KNC which supports 240 hardware threads. To address this we applied nested parallelism at two levels. In the first level, the aggregate L2 capacity was used to heuristically estimate the total concurrency in subdomains (N_S) at the finest level (64^3) that can be attained without generating capacity misses. In the second level, $4(\lfloor 60/N_S \rfloor)$ threads are assigned using compact affinity to smooth a given subdomain. Pencils within a plane are interleaved among threads to maximize sharing among threads within a core. A `#pragma omp parallel` construct was used to control the complex dissemination of subdomain- and pencil-level operations among threads.

This static approach to parallelism becomes inefficient when descending through the V-cycle — exponentially decreasing grid sizes suggests that intra-box parallelism can be traded for inter-box. Therefore, profiling was used to construct an auto-tuner that selects the optimal balance between threads per subdomain and the number of concurrent subdomains at each level of the V-cycle. Once again, this is expressed within a `#pragma omp parallel` region.

Initial application of the communication avoiding wavefront yielded disappointing results. This is because the compiler was unable to accurately insert software prefetches for this complex memory access pattern (unlike the relatively simple memory access pattern of the baseline implementations). Thus, for wavefront approach in GSRB kernel, we inserted prefetches manually. The rest of the code still uses automatic software prefetches inserted by the compiler.

To maximize performance of in-cache computations, SIMD intrinsics were applied to the GSRB kernel. Similar to the approach on CPUs, we compute as if doing Jacobi and use masked stores to selectively update red or black values in memory. Moreover, large (2MB) TLB pages were used, and the starting address of each array was padded to avoid a deluge of conflict misses when multithreaded cores perform variable-coefficient stencils within near power-of-two grids. Similarly, the i-dimension (including the ghost zones) was padded to a multiple of 64 bytes.

To minimize the number of communicating neighbors, the KNC implementation leverages the shift algorithm [20] in which communication proceeds in three phases corresponding to communication in i, j, and k, where in each phase, the subdomains only communicates with their two neighbors.

6.7 GPU-Specific Optimizations

The GPU implementation presented in this report is a highly-optimized CUDA implementation that does not attempt to avoid communication. Thus, it is directly comparable to the baseline implementation of miniGMG. All computation including boundary exchanges takes place on the GPU. As a result, for single-node experiments, there is no need for explicit PCIe communication. Nevertheless, CUDA kernels are dispatched to the PCIe-attached GPUs. In reality, the very act of initiating a CUDA kernel requires PCIe communication. Unfortunately, as multigrid requires operations on ever smaller boxes (arrays), the time for this kernel execution initiation cannot be effectively amortized, and performance (time on coarser levels) departs from the ideal $\frac{1}{8}^{th}$ scaling.

The meta data for miniGMG is relatively complex. Rather than flattening it for offload to the GPU, a deep copy of all meta data associated with boxes and the decomposition is copied to the device. This creates a complex web of CPU pointers to CPU data, CPU pointers to GPU data, GPU pointers to GPU data. Nevertheless it dramatically simplified the interface presented in miniGMG's multigrid solver and facilitated a distributed memory implementation that will be discussed in future work.

Nominally, the GPU implementation executes 3D CUDA kernels decomposed into one dimension of boxes (the box list) and two dimensions of tiles within a box. The 2D tiling (i and j) within each box

maximizes spatial locality and allows a thread block to stream through the k-dimension to maximize reuse. As one descends through the V-Cycle, the number of boxes remain the same. However, as the size of each box decreases, the size of the CUDA kernel also decreases (there are fewer thread blocks per box). Eventually the number of thread blocks equals the number of boxes.

A number of parameters exist in the GPU implementation to facilitate tuning including use of the shared memory to maximize data reuse and to reduce register pressure by caching the read-only and replicated pointers to component data. Additionally, one may tune the size of each thread block to balance occupancy and register pressure. In this report, we use 32×4 thread blocks. Thus, at the finest level, there are 2048 thread blocks each of 128 threads.

6.8 GPU-Specific Communication-Avoiding

In this report, we will only present data from our highly optimized CUDA implementation. However, ongoing work by Singh is examining a communication-avoiding implementation [23]. We discuss some initial insights here.

The limited memory per core (SM) coupled with the lack of cache coherency dramatically complicates the implementation. Like the baseline CUDA implementation, a communication-avoiding version of smooth would stream through the k-dimension exploiting temporal locality in shared memory and the register file. However, unlike the baseline version, when taking s-steps, it must keep up to $7s+3$ planes in registers and shared memory. For a 16×16 thread block and four smooths, this amounts to almost 64KB of memory. Such high memory requirements hurt occupancy and would impede bandwidth. As the working set size is tied directly to the thread block size, using a larger thread block size would not reduce register spilling, while a smaller block exacerbates the redundancy and impedes performance.

7 Single-node Results and Analysis

All experiments in this section use one MPI process per NUMA node (e.g. four MPI processes on one XE6 node). However, as the accelerated-systems have only one accelerator per node, they do not exploit MPI.

7.1 Performance on the Finest Grids

Figure 4 shows a breakdown of the total time miniGMG spent on the finest grids before and after optimization. "Communication" is inter-box ghost zone exchanges and includes both intra- and inter-process communication. Nominally, miniGMG solve time is expected to be dominated by the execution at the finest resolution. Overall results show impressive speedups from our optimization suite ranging from 1.6× on the Nehalem cluster to over 5.4× on KNC; Observe that despite the data movement overhead for Smooth, DRAM communication-avoiding accelerated CPUs and KNC. Conversely, we see that aggregating ghost zone exchanges showed practically no performance benefit on the CPUs, while significantly reduced communication overhead on KNC by 33%.

In order to understand these seemingly contradictory performance effects on Smooth for CPUs, GPUs, and KNC, we construct a simple data movement and bandwidth model for each architecture based on the volume of the grid with a 1-deep (66^3) or 4-deep (72^3) ghost zone. Table 3 shows that the baseline implementation of Smooth on all CPUs attain an extremely high percentage of STREAM bandwidth. Conversely, the baseline implementation using naive threading dramatically underutilizes KNC. With optimized nested parallelism using OpenMP, KNC's Smooth performance more than doubles, attaining a sustained bandwidth of over 120 GB/s (80% of STREAM). Data movement

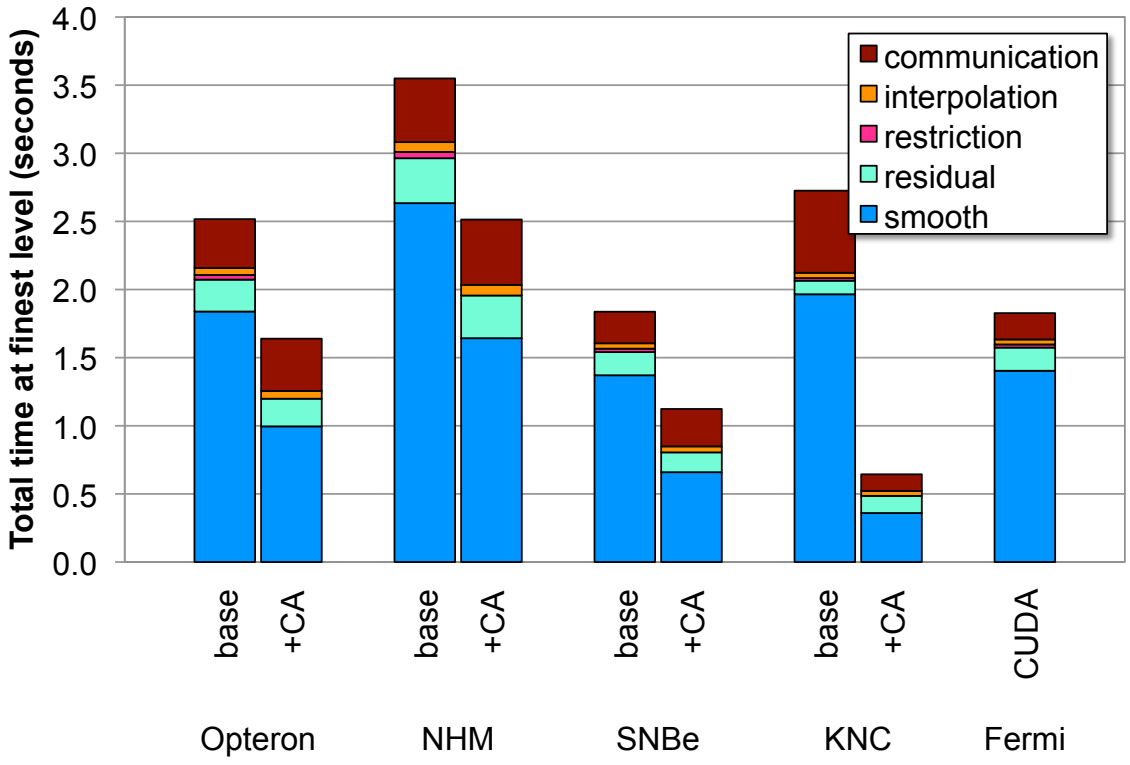

Figure 4: Breakdown of time at the finest level for the baseline and fully-optimized communication-avoiding (+CA) versions.

estimates for the GPU (assuming 32-byte cache lines) show that the baseline GPU implementation moves 28% more data than the basic CPU versions. This is because on-chip memory heavily constrains thread block surface:volume ratios and adjacent thread blocks cannot exploit shared locality.

In practice, DRAM communication-avoiding implementations move less than one-third of the data on CPUs and KNC — the thick ghost zones inhibit attaining the ideal one-quarter. Unfortunately, the complex memory access pattern does not synergize well with a hardware prefetcher. As a result Table 3 shows the sustained bandwidth is less than ideal. On the GPUs, no communication-avoiding (i.e. no redundancy) resulted in the best overall solve time.

	System	**XE6**	**NHM**	**SNBe**	**KNC**	**Fermi**[1]
	STREAM (GB/s)[2]	49.4	38	70	150	120
baseline miniGMG	Time (seconds)	1.84	2.63	1.37	1.97	1.40
	Data Moved (10^9B)	94.2	94.2	94.2	94.2	121
	Bandwidth (GB/s)	51.2	35.8	68.7	47.9	86.5
Communication Avoiding miniGMG	Time (seconds)	1.00	1.64	0.66	0.36	—
	Data Moved (10^9B)	30.6	30.6	30.6	30.6	—
	Bandwidth (GB/s)	30.7	18.6	46.4	85.0	—
	Speedup	1.8×	1.6×	2.1×	5.4×	—

Table 3: **Estimated data movement and effective DRAM bandwidth for *smooth()* on the finest (64^3) grids across all V-cycles before and after communication avoiding.**. [1]*Assumes 32B memory transfers and requisite inter-thread block redundancy.* [2]*See Table 1.*

7.2 Performance of the V-Cycle

Figure 5 presents the total time miniGMG spends at each V-cycle level before and after tuning. On SNBe and KNC, results show expected exponentially decreasing run times by level. Interestingly, we observe a crossover where the GPU actually outperformed the (under parallelized) baseline OpenMP implementation on MIC on the finest grids. On coarser grids, MIC's cache made up for deficiencies in threading in the baseline implementation.

As discussed in Section 7.1, results show that our threading and optimization techniques on CPUs and KNC ensure significant performance improvements at the finest grids. Interestingly, on SNBe, the non-communication avoiding version is faster than either the optimized SNBe or KNC at the coarsest grids. Not surprising given that the communication-avoiding version will perform 27× more flops and move 4× more data between subdomains on the coarsest grid. However, as the overhead at the bottom level is more than two orders of magnitude less than the time at the finest grid, the impact is minor and does not motivate heterogeneous optimization.

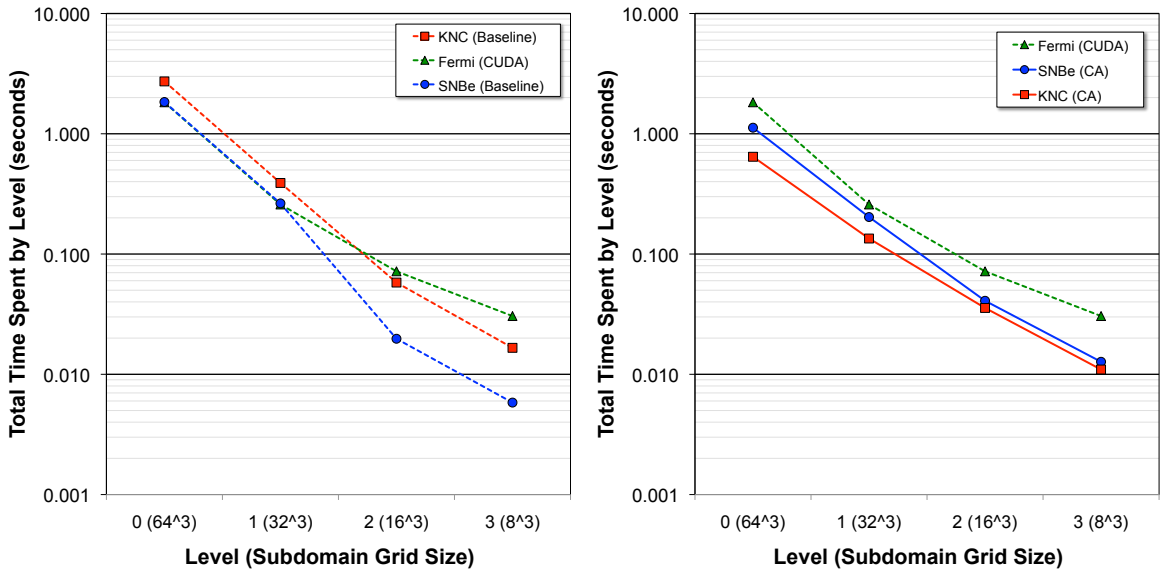

Figure 5: miniGMG solve time by level before and after communication avoiding. Note, dotted lines are baseline, solid are communication-avoiding.

7.3 KNC Performance Analysis

Given that this is the first study describing performance of KNC, we now present a more detailed description of our optimization impact. Figure 6 shows the benefit of progressive optimization levels for both Smooth and the overall solve time normalized to the corresponding OpenMP baseline implementation on KNC. Observe that nested threading's superior utilization of hardware threads improves the overall solve time by a factor of 2.2×. Additionally, our auto-tuned threading further improves performance another 7% by optimizing thread distribution for each level. Note that communication-aggregation/avoiding is only applied starting with wavefront, where without proper prefetching, the complex data movement patterns cannot be efficiently realized by the compiler alone, resulting in lowered performance. However, combining tuned threading, wavefront, hand-tuned prefetching, and SIMD vectorization, improves performance by 74% on Smooth and 43% overall. Finally array padding

and the use of 2MB pages give an additional performance boost resulting in a significant overall speedup of 5.4× and 3.8× for Smooth and solver time respectively.

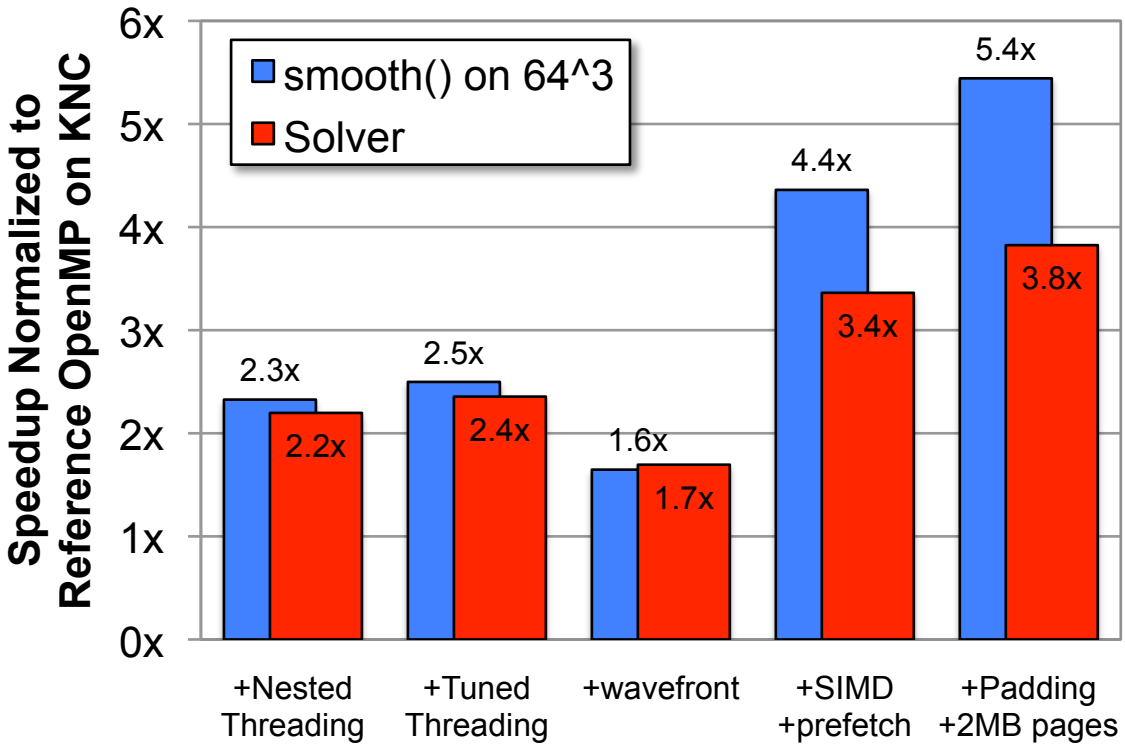

Figure 6: KNC speedup as a function of progressive optimization and normalized to the baseline implementation. Note, in addition to overall solver speedup, we also show speedup of Smooth on the finest grids.

7.4 Overall Speedup for the miniGMG Solver

Figure 7 presents the overall performance (higher is better) of the miniGMG solver before and after optimization, normalized to baseline Opteron performance. We also include the newer K20x (Kepler) GPU as a point of comparison. We integrate the performance trends in Figure 5 and observe that the fastest machine for the baseline implementation is SNBe — no surprise given the mismatch between parallelism in the code and parallelism in KNC. However, proper exploitation of thread-level parallelism on KNC can more than double the performance of the solver.

With manual optimization and no communication-avoiding, the Kepler-based GPUs deliver the best performance. This should come as no surprise given they have the highest bandwidth. However, with full tuning (communication-avoiding, SIMD, prefetch, etc.), results show see an 1.5× and 3.8× increase in solver performance on SNBe and KNC respectively. Although KNC was able to attain a speedup of about 5.4× on Smooth on the finest grids, the time spent in the other phases and in the other levels amortizes this benefit. Overall, the KNC exceeds SNBe performance by 1.75× while exceed the Opteron by almost 2.5×. As discussed in Section 7.1, the high working sets associated with communication-avoiding smooth limit the GPUs ability to form sufficient locality on chip. Nevertheless, as it stands now, the best machine depends heavily on the degree to which one wishes to tailor

the inherent algorithm (not simply optimization of an implementation) to the underlying architecture.

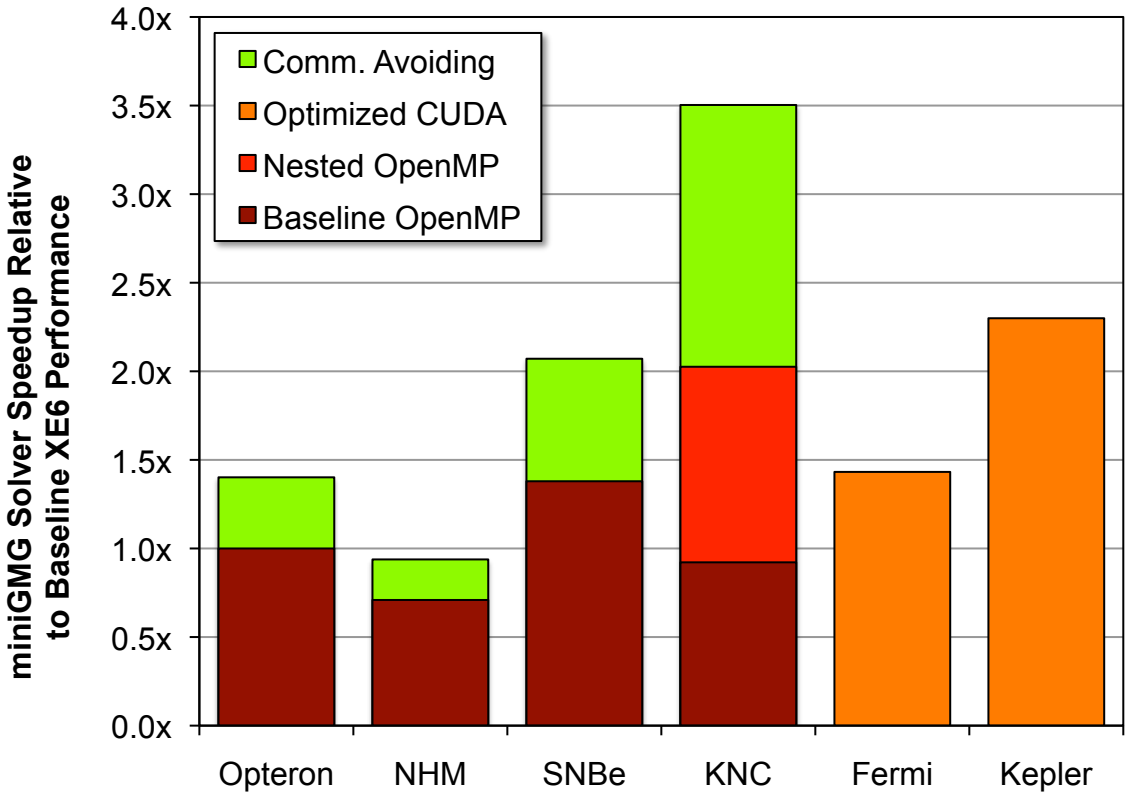

Figure 7: miniGMG performance normalized to the baseline implementation on the Opteron-based Cray XE6 compute node.

8 Scalability of the V-cycle

We now explore the scalability of the V-cycle in the distributed memory environments. Note, the scalability of various bottom solvers is an important yet orthogonal issue that was examined in orthogonal work [28]. Figure 8 shows the weak scaling of the more desirable 256^3 problem *per NUMA node* (vs. 256^3 per compute node) by doubling the number of processes in each dimension, for a total of 24,576 cores on the XE6 (Hopper). As described in Section 6, all off-node messages are always aggregated through process-level buffers to avoid overheads. Additionally, to proxy the limitations of AMR applications in which the underlying grids are subject to refinement, we do not leverage a cartesian mapping of MPI processes. Rather, we are constrained by the performance of the underlying job schedulers and network architectures.

Observe that for all studied concurrencies our optimizations yield significant speedups. The Gordon SNBe cluster attains the best overall performance and comparable scalability. A detailed analysis of the performance breakdown shows that only the `MPI_Waitall()` overhead increases with larger scalability on all systems. Given the lack of cartesian process mapping and an underlying 3D torus interconnect, we may have effectively random placement of processes. Thus, ignoring contention, each octupling

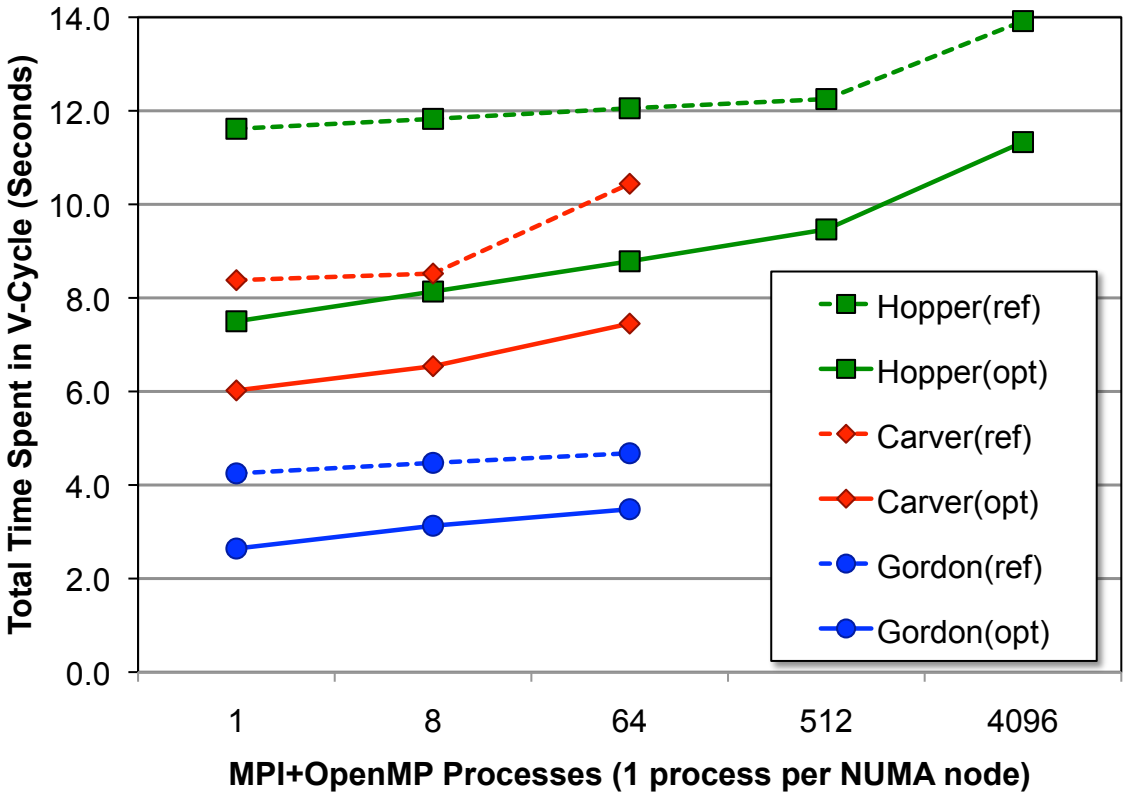

Figure 8: miniGMG weak scaling results. Note, there are 4, 6, and 8 cores per process on Carver, Hopper, and Gordon; thus, Hopper uses up to 24,576 cores. All runs allocate one process per *NUMA node* and 64×64^3 subdomains per process.

in the number processes (for weak scaling) is expected to double the number of required message hops. For the large messages at the finest resolution, we see MPI time saturates at high concurrencies. However, for all coarser grids, the measured MPI time follows the expected exponential trend doubling with concurrency through 512 processes. At 4K processes, results show a 4-7× increase in MPI time, possibly indicating high congestion overheads; future work will investigate reducing these overheads.

Finally, we can consider the impact of these data in the context of a KNC-accelerated multi-node system. An extrapolation based on the interconnect characteristics of the Gordon cluster would result in 46% of the time spent in MPI. Clearly, enhancements to network performance are essential to reap the benefits of co-processors or accelerators on multigrid calculations.

9 Conclusions

Data movement is the primary impediment to high performance on existing and emerging systems. To address this limitation there are fundamentally two paths forward: algorithmic and architectural. The former attempts to restructure computation to minimize the total vertical (DRAM–cache) and horizontal (MPI) data movements, while the latter leverages technological advances to maximize data transfer bandwidth.

This report explores both of these avenues in the context of miniGMG, our 3D geometric multigrid benchmark. To reduce vertical and horizontal data movement, miniGMG allows one to explore the use communication-aggregation and communication-avoiding techniques to create larger (but less frequent) messages between subdomains, while simultaneously increasing the potential temporal reuse within the GSRB relaxation. Ee evaluate performance on Intel's recently announced Knights Corner co-processor, as well as the NVIDIA M2090 and K20x, all of which rely on GDDR5 to maximize bandwidth at the expense of reduced on-node memory.

Results show that our threaded wavefront approach can dramatically improve smooth run time on the finer grids despite the redundant work. Effectively implementing this approach poses two significant challenges: how to productively decouple DRAM loads from the in-cache computation on the wavefront, and how to efficiently express sufficient parallelism without sacrificing sequential locality. On CPUs and KNC, the hardware prefetchers designed to decouple memory access through speculative loads are hindered by the lack of sequential locality — an artifact of extreme thread-level parallelization. On highly-multithreaded architectures like the GPUs, this is not an issue and parallelization in the unit-stride is feasible. However, the CPUs and KNC have sufficient on-chip memory and efficient intra-core coalescing of memory transactions to realize the benefits of communication-avoiding.

On SNBe, our optimizations demonstrated a 50% overall increase in solver performance over the baseline version; this is an impressive speedup particularly given the memory-bound nature of our hybrid MPI+OpenMP calculation. On KNC, the performance gains are even more dramatic achieving a 3.5× improvement. These large gains are an artifact of thread under-utilization in the baseline implementation combined with a 1.6× increase in performance through communication avoiding and various low-level optimization techniques.

This report also shows that on CPUs and KNC, neither communication aggregation nor DRAM communication avoiding provided any substantial value on coarse grids deep in V-cycle. Additionally, distributed memory experiments demonstrated that, despite achieving scalability up to 24,576 cores, the substantial time spent in MPI will be an impediment to any multi- or manycore system. Unfortunately, communication-aggregation's use of deep ghost zones can result in a 50% increase in memory usage for 64^3 boxes and a 100% increase in memory usage for 32^3 boxes. Thus, if DRAM capacity, rather than raw bandwidth, is the ultimate constraint on the road to exascale, the impact of communication-avoiding techniques will be hindered.

The common OpenMP implementation shared by CPUs and KNC is a significant step in manycore portability and programmability. However, substantial restructuring for parallelism was required to deliver high-performance on KNC. Portable constructs that allow multiple, variably-sized, collaborating thread teams will be essential in the future to maximize productivity on all platforms. Unfortunately, in order to maximize performance, CPUs and KNC required the time-consuming use of SIMD intrinsics to make parallelism explicit. The GPUs required a rewrite in CUDA to make parallelism and locality explicit. However, as CUDA is not premised on the concept that programmers should be able to reason about the execution ordering of thread blocks, it is particularly challenging for thread blocks on a GPU to form a shared working set in the L2. This stands in stark contrast to CPUs/KNC where programmers can manually orchestrate thread execution to form constructive locality.

Acknowledgments

Authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number 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 No. DE-AC02-05CH11231. 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. This work was supported in part by National Science Foundation grants: OCI #0910847, Gordon: A Data Intensive Supercomputer; and OCI #1053575, Extreme Science and Engineering Discovery Environment (XSEDE). This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735.

References

[1] N. Bell, S. Dalton, and L. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, June 2011.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[3] BoxLib. https://ccse.lbl.gov/BoxLib.

[4] Carver website. http://www.nersc.gov/users/computational-systems/carver.

[5] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. Autotuning multigrid with petabricks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 5:1–5:12, New York, NY, USA, 2009. ACM.

[6] CHOMBO. http://chombo.lbl.gov.

[7] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*, pages 63–74. ACM, 2010.

[8] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC 2008)*, pages 1–12, 2008.

[10] C. C. Douglas, J. Hu, M. Kowarschik, U. Rde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal*, 10:21–40, 2000.

[11] M. Frigo and V. Strumpen. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS05)*, Boston, MA, 2005.

[12] P. Ghysels, P. Kosiewicz, and W. Vanroose. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra with Applications*, 19(2):253–267, 2012.

[13] Gordon website. http://www.sdsc.edu/us/resources/gordon.

[14] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[15] Hopper website. http://www.nersc.gov/users/computational-systems/hopper.

[16] M. Kowarschik and C. Wei. Dimepack - a cache-optimized multigrid library. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), volume I*, pages 425–430. CSREA, CSREA Press, 2001.

[17] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

[18] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[19] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[20] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of MPP architectures. In *Proc. PDCS International Conference on Parallel and Distributed Computing Systems*, pages 192–197, 2002.

[21] G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.

[22] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[23] A. Singh. Private communication.

[24] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[25] M. Sturmer, H. Kostler, and U. Rude. How to optimize geometric multigrid methods on GPUS. In *Proc. of the 15th Copper Mountain Conference on Multigrid Methods*, Copper Mountain, CO, March, 2011.

[26] J. Treibig. *Efficiency improvements of iterative numerical algorithms on modern architectures*. PhD thesis, 2008.

[27] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *International Computer Software and Applications Conference*, pages 579–586, 2009.

[28] S. Williams, E. Carson, M. Lijewski, N. Knight, A. Almgren, J. Demmel, and B. V. Straalen. s-step krylov subspace methods as bottom solvers for geometric multigrid. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, 2014.

[29] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, Miami, Florida, 2008.

[30] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.

[31] S. Williams, L. Oliker, J. Carter, and J. Shalf. Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 55:1–55:12, New York, NY, USA, 2011. ACM.

[32] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers*, New York, NY, USA, 2006.

[33] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS:Interational Conference on Parallel and Distributed Computing Systems*, Cancun, Mexico, 2000.

[34] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager. Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method. *Progress in Computational Fluid Dynamics*, 8, 2008.