Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems

Sundaresan Venkatasubramanian
Georgia Institute of Technology
College of Computing
School of Computer Science
266 Ferst Drive, Altanta, Georgia, USA
sundarvenkat2@gmail.com

Richard W. Vuduc
Georgia Institute of Technology
College of Computing
Computational Science and Engineering Division
266 Ferst Drive, Altanta, Georgia, USA
richie@cc.gatech.edu

ABSTRACT

We describe heterogeneous multi-CPU and multi-GPU implementations of Jacobi's iterative method for the 2-D Poisson equation on a structured grid, in both single- and double-precision. Properly tuned, our best implementation achieves 98% of the empirical streaming GPU bandwidth (66% of peak) on a NVIDIA C1060, and 78% on a C870. Motivated to find a still faster implementation, we further consider "wildly asynchronous" implementations that can reduce or even eliminate the synchronization bottleneck between iterations. In these versions, which are based on *chaotic relaxation* (Chazan and Miranker, 1969), we simply remove or delay synchronization between iterations. By doing so, we trade-off more flops, via more iterations to converge, for a higher degree of asynchronous parallelism. Our wild implementations on a GPU can be 1.2–2.5× faster than our best synchronized GPU implementation while achieving the same accuracy. Looking forward, this result suggests research on similarly "fast-and-loose" algorithms in the coming era of increasingly massive concurrency and relatively high synchronization or communication costs.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Parallel and vector implementations; D.1.3 [**Programming Techniques**]: Concurrent programming—*parallel programming*; C.4 [**Performance of Systems**]: Performance attributes, Modeling techniques

General Terms

Algorithms, experimentation, performance

1. INTRODUCTION AND SCOPE

This study began with what we thought would be a trivial exercise: given a memory bandwidth-rich GPU platform, take a bandwidth-bound computation with regular memory access and produce a code that runs at the bandwidth limit.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...$5.00.

In particular, we studied the problem of Figure 1, a textbook instance of Jacobi's method for a centered finite-difference approximation of the 2-D Poisson equation on a square domain, regularly discretized [9, Chap. 6]. At first glance, this kernel appeared simple to implement on a GPU.

Contrary to our expectation, we find that achieving a very high-level of performance in practice—*i.e.*, running near the bandwidth limit using the high-level CUDA programming model [1]—requires careful implementation. Our first contribution is to describe some of the tuning lessons we learned along the way. A second related contribution is our extension of our initial GPU-only implementation to the hetergeneous multi-GPU and hybrid multi-CPU/multi-GPU cases. These extensions are accompanied by predictive performance models that help decide when these implementations pay off.

Though these implementations perform well, they are still limited by a fundamental bottleneck: the cost of synchronization. Looking forward, we expect this cost will only get worse as core counts and core heterogeneity increase. These observations motivate the third contribution of this paper, which is to "throw out" the deterministic synchronization and replace it with non-deterministic asynchronous parallelism. This technique exploits the structure of our specific computation, but yields 1.2–2.5× speedups while achieving the same level of accuracy.

This specific idea is not new; Chazan and Miranker suggested it in their seminal 1969 paper on *chaotic relaxation* [7]. Most recently, researchers have revisited chaotic relaxation in a more general form, termed *asynchronous iteration*, particularly for the context of heterogeneous clusters and grid environments [10]. If a GPU reflects possible future multi- and many-core architectural designs, our positive results on GPUs suggest that the pursuit of similarly "fast-and-loose" algorithms will be a fruitful direction.

The scope of this paper is limited in at least three ways. First, we consider a relatively simple kernel. Whether the tuning techniques and more radical unsynchronized implementations apply more broadly is not well-understood. Secondly, our experimental results for the hybrid CPU/GPU implementations do not show significant speedups. Nevertheless, we believe these results are due to our specific evaluation hardware; our performance models suggest payoffs should be possible on other configurations. Finally, the unsychronized methods have inherent non-determinism. They depart from the baseline parallel implementation in ways that require careful mathematical justification and analysis, which related work addresses (Section 2). Our focus

Problem: Solve Poisson's equation in 2-D on a square grid:

$$\begin{aligned} -\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)u(x,y) &= f(x,y), \\ 0 < \quad x,y &< 1, \\ u(0,y)=u(x,0) &= 0 \end{aligned}$$

Centered finite-difference approximation on a $(N+2)\times(N+2)$ regular grid with step size h:

$$\begin{aligned} 4\cdot u_{i,j}-u_{i-1,j}-u_{i+1,j}-u_{i,j-1}-u_{i,j+1} \\ = h^2\cdot f_{i,j}+O(h^2) \end{aligned}$$

Jacobi's method for this problem and approximation:

```
1: $U^0 \leftarrow 0$ // Initializes an $(N+2)\times(N+2)$ grid
2: // For each iteration, $t$ ($T$ iterations in all)
3: for $t \leftarrow 1,2,\dots T$ do
4:     for $i \leftarrow 1\dots N$ do
5:         for $j \leftarrow 1\dots N$ do
6:             $U^{t+1}_{i,j} \leftarrow \frac{1}{4}\cdot(U^t_{i+1,j}+U^t_{i-1,j}+U^t_{i,j+1}+U^t_{i,j-1}+h^2\cdot F_{i,j})$
7:         end for
8:     end for
9: end for
```

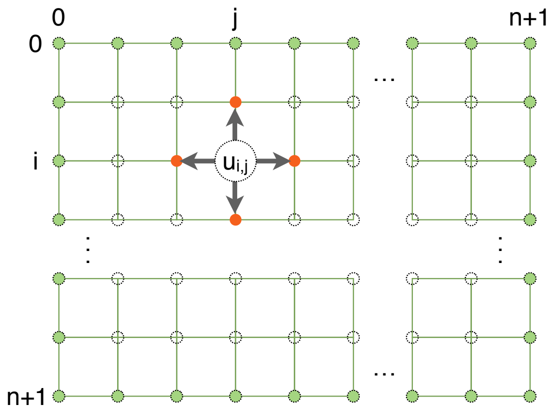


Figure 1: Jacobi's method for the 2-D Poisson equation. (*Left*) Continuous problem, discrete approximation, and solver pseudocode. (*Right*) Illustration of the discrete grid and nearest-neighbor dependences.

here is on *potential performance gains*, to suggest the feasibility of loose synchronization in other problem and architectural contexts.

2. RELATED WORK

A number of other researchers are investigating stencil kernels on GPUs. In the most extensive recent study of which we are aware, Datta, *et al.*, tune 3-D stencil kernels for a broad variety of multicore platforms, including GPUs [8]. In an unpublished manuscript, Giles reports a high fraction (75%) of sustained bandwidth for a 3-D kernel as well on a GPU platform [11]. Amorim, *et al.*, consider the 2-D case as we do here, though they focus on the 9-point stencil and examine a subset of our tuning space [2]. Beyond stencils, some researchers have considered more irregular general sparse matrix kernels and solvers for GPUs [4]. Our paper differs from these in that (a) we consider algorithmic variations via loosely synchronized designs; and (b) we develop hybrid CPU/GPU implementations as well as GPU-only versions.

The concepts of chaotic relaxation and asynchronous iteration, have a long history but is largely considered "outside" mainstream numerical methods because of the use of non-determinism [7, 3, 10, 16]. Researchers have considered these methods for use in heterogeneous parallel systems, including grid systems, as noted in the survey paper by Frommer and Szyld (2000) [10]. We show these basic techniques are relevant to GPUs, further suggesting that the techniques will become only more relevant to future many-core systems.

Though we use non-deterministic—but ultimately still converging—methods, other researchers have strongly argued that asynchronous algorithms are necessary for current and future multicore platforms even in the deterministic case [6, 5]. Our philosophy and results are aligned with these views.

Looking to alternative paradigms, a related emerging body of work attempts to formulate numerical solvers for time-dependent partial differential equations based on *parallel discrete-event simulation*, including methods known as *asynchronous variational integrators* [15, 14, 13, 12]. These formulations result in similarly asynchronous parallel behavior, and so we regard these methods as another promising class of approaches for future multi- and many-core systems.

3. CODE DESIGN AND TUNING

In this section, we describe our implementations and tuning process. We consider multi-CPU variants, single- and multi-GPU variants, hybrid CPU/GPU designs, and finally "wildly" asynchronous implementations, which can trade-off higher numbers of flops for fewer or cheaper synchronizations. We evaluate these implementations in Section 4.

3.1 CPU baselines

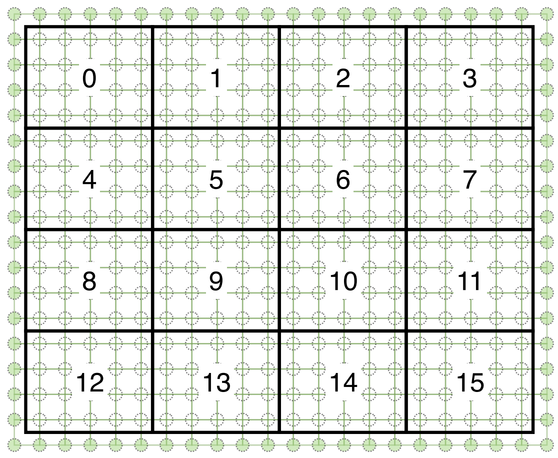
We consider both sequential and parallel Pthreads-based implementations as our CPU-based baselines. We use a 2-D block partitioning of the domain in the parallel case. The single-program multiple data (SPMD) style pseudocode for each thread of the parallel implementation, which executes T Jacobi iterations, is as follows:

```
1: for $t \leftarrow 1,2,\dots,T$ do
2:     Update my block, $b$: $U^{\text{new}}_b \leftarrow$ update($U^{\text{cur}}_b$)
3:     barrier()
4:     Logically swap $U^{\text{new}}_b$ and $U^{\text{cur}}_b$ (i.e., swap pointers)
5: end for
```

The explicit barrier in line 3 ensures the parallel code computes the same result (to within round-off) as the sequential code in Figure 1.

For the CPU implementation, we perform a "minimal" tuning as follows.

First, we write the code so that the compiler SIMD vectorizes it. We verify this vectorization by inspecting the assembly code.



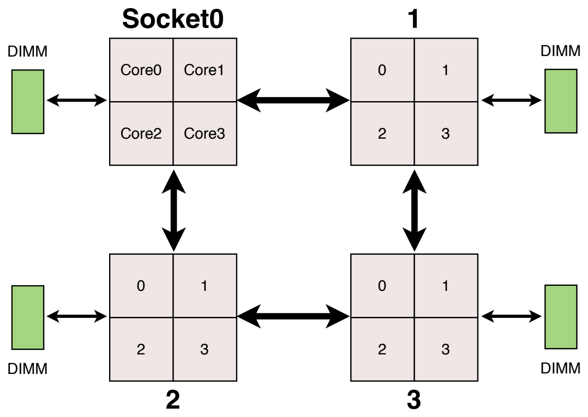


Figure 2: (*Left*) A 16×20 grid of unknowns (plus boundaries) partitioned into a 2-D blocked grid of 4×5 unknowns per block. (*Right*) A quad-socket NUMA system with quad-core processors.

Secondly, we bind threads to cores, *e.g.*, using the Linux affinity scheduling routines. For our experimental evaluation, we consider both explicitly bound and unbound cases.

Thirdly, for the non-uniform memory access (NUMA) multicore CPU architecture used in our evaluations, we allocate the per-thread data blocks, U^{new} and U^{cur}, in the memory of the closest processor socket [18]. An example of such an architecture appears in Figure 2 (*right*). Our NUMA implementation requires (1) ghost-cells to exchange thread-boundary data; (2) one more barrier to preserve the serial code semantics; and (3) that we also bind threads to sockets in a way that physically matches the layouts of the cores and sockets. For instance, we would map blocks {0, 1, 4, 5} in Figure 2 (*left*) to socket 0, blocks {2,3,6,7} to socket 1.

Much more extensive tuning for the CPU is possible, as discussed by others [8, 11, 2]. Our intent here is *not* to compare the CPU and GPU. Instead, we aim (a) to provide the reader with a useful baseline CPU implementation for rough comparison to a GPU, and (b) to provide our subsequent hybrid CPU/GPU implementations with a "reasonably tuned" sequential and parallel CPU components.

3.2 GPU implementations

We consider a variety of implementation techniques for NVIDIA CUDA-based GPU systems, based on the extensive experience of others [17, 1].

We treat current-generation NVIDIA GPUs using the following hardware abstraction, as proposed by others [17]. First, a GPU is a multiprocessor system with a memory hierarchy consisting of a *device memory*, which is the main memory on the GPU; a *shared memory*, which is a local-store shared among blocks of threads; and a *texture memory* for explicitly declared read-only data, which is automatically cached and has comparable latency to shared memory but with fewer capacity and alignment constraints. Secondly, each multiprocessor may be viewed as a massively multi-threaded vector processor.

Informed by this view, we tuned our GPU implementation by considering following techniques in turn (Section 4).

A baseline ("naïve") GPU implementation. The 2-D block partitioning used for the Pthreads implementation (Section 3.1) is straightforward to implement on a GPU. We divide the domain as shown in Figure 2 (left), where each subblock is assigned to a CUDA thread block, with a logical thread assigned to each unknown. This implementation mirrors the CPU-only case: we maintain two grids in the device memory for the "current" and "new" grid point values, respectively, and logically swap these grids at each iteration. The block size is a tuning parameter, which can be chosen through either manual analysis, use of CUDA diagnostic tools (*e.g.*, occupancy calculator), or experiment.

The CPU pseudocode is roughly as follows:

```
1: Copy the grid from main host memory to the GPU.
2: for $t \leftarrow 1 \ldots T$ do
3:    Invoke GPU kernel for all "thread-blocks."
4:    (Implicit) Synchronize host and GPU.
5:    Logically swap active grid.
6: end for
7: Copy grid results from GPU to host memory.
```

Line 3 invokes execution of 1 iteration on the GPU, which corresponds to a 2-D block parallel implementation. The synchronization in line 4 occurs implicitly when the invoked kernel returns.

This baseline suffers from at least two performance problems. First, it ignores the GPU memory hierarchy by not explicitly making use of shared memory. Secondly, it will usually suffer from non-coalesced memory accesses, which are essentially unaligned vector memory operations.

Padding. We can avoid non-coalesced memory accesses by padding the grid(s) in the GPU memory to guarantee 256-byte aligned memory accesses. We illustrate padding in Figure 3 where, assuming a row-major layout, we simply store extra elements at the beginning of each row.

Shared memory, without padding. For each thread-block, we explicitly allocate a block of local-store ("shared") memory that all threads in the associated thread-block share. This block is stored in row-major order, and includes ghost cells to hold elements from neighboring blocks. The kernel computation now consists of three phases: (A) copying the block and fringe/boundary elements from device memory to shared memory; (B) computation; and (C) writing

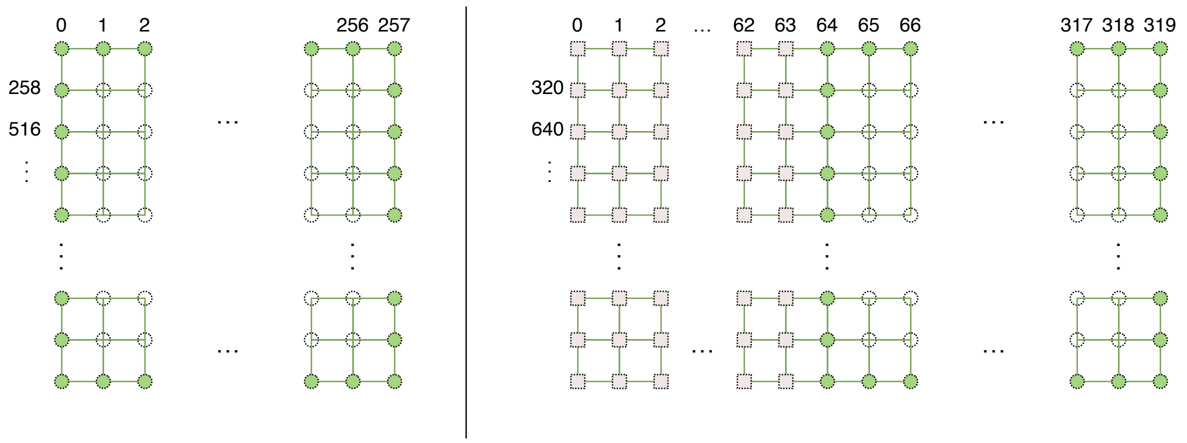


Figure 3: (*Left*) A conventional row-major layout, for $n = 256$ (258×258 grid), which could lead to costly non-coalesced memory accesses on a GPU. (*Right*) A padded row-major layout, for $n = 256$, avoids the problems of the conventional layout.

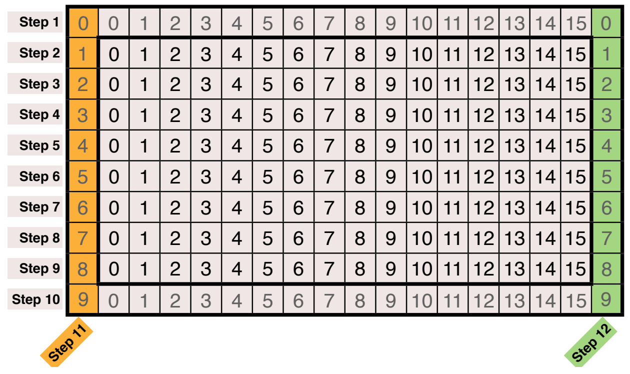


Figure 4: Thread block assignments, for a 16 thread thread-block operating on a 8×16 block of unknowns (plus fringe elements).

the updated unknowns back to the grid in global memory. Compared to the baseline, this code reduces the total number of device memory fetches by $3\times$ *and* we can eliminate storage of the second grid.

We must carefully assign threads to words of data, to avoid shared memory *bank conflicts*. For a thread block consisting of a particular number of threads, Figure 4 shows how we assign threads to unknown elements within the block during the reading phase (A). Each interior square represents an unknown, numbered by the thread assigned to read that element during phase (A). However, we cannot avoid bank conflicts (or even non-coalesced reads) for the left and right fringes. Thus, we expect to try to make a block as large as possible while minimizing the number of rows in each block. One may argue that we could have 2 more threads and fetch left and right fringes while doing a row access (steps 1 to 10 in the figure) and leave the two threads idle during computation. Such an access pattern causes either reduced *occupancy* (thread/register utilization) or causes non-coalesced accesses, depending on the number of threads.

Texture memory. A read-only texture cache is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, and otherwise costs just one read from the texture cache [1]. We consider binding the global memory to 1-D textures.

Shared memory with padding. We again pad the grid in device memory to reduce non-coalesced reads during phase (A) above. Optimum allocation of shared memory per thread block is also necessary to achieve good performance.

Unrolling. We also explicitly request unrolling of the innermost loops of the computation using the appropriate CUDA directive, and tune the unrolling depth.

A double-precision trick. Using double-precision (8-byte words) leads to 2-way bank conflicts during shared memory accesses, because the banks are arranged in a way that favors vector loads on 4-byte words. We avoid this problem by separately storing the lower and upper 4-byte words, and recombining them prior to computation using a pre-defined CUDA macro (`__hiloint2double()`) [1].

3.3 Hybrid implementations

We also consider hybrid CPU/GPU implementations for systems in which one wishes to avoid idle CPUs and/or GPUs. We consider a very basic strategy that assigns a block of rows to the CPU(s) with the remaining rows assigned in blocks to available GPUs. For T Jacobi iterations on an $n \times n$ grid, we illustrate our approach in Figure 5, which implements the following.

```
1: // Assign rows $1 \dots s$ to the CPU(s),
2: // and rows $s+1 \dots n$ to the GPU(s).
3: for $t \leftarrow 1 \dots T$ do
4:     Step 1 (GPU part): Compute one iteration of Jacobi
       for the last $n - s$ rows of the grid.
5:     Step 2 (CPU part): Simultaneously compute one
       iteration of Jacobi on rows $1 \dots s$.
6:     Step 3 (Exchange data): Transfer the boundary
       rows between CPU and GPU.
7: end for
```

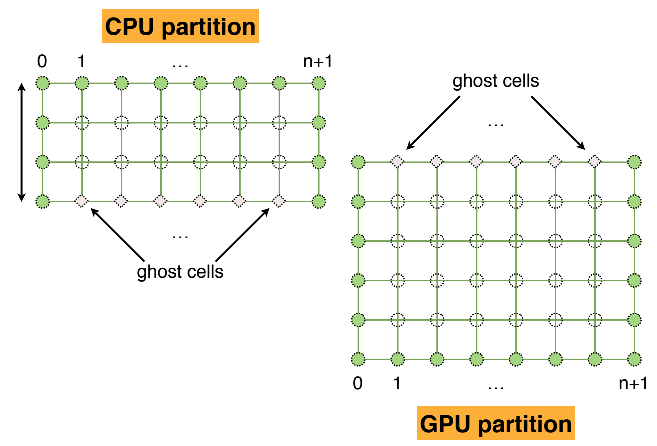


Figure 5: Block row partitioning used for the hybrid CPU/GPU implementation.

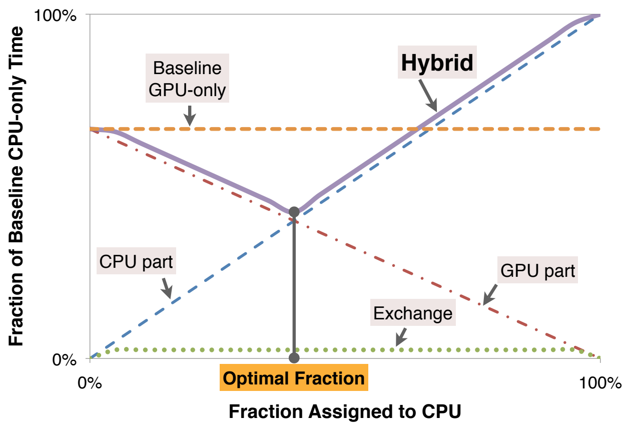


Figure 6: Illustration of expected performance of a hybrid CPU/GPU implementation.

This variant wins if Step 2 can do useful work before Step 1 finishes, given the copying overhead in Step 3. We illustrate this concept in Figure 6, which shows the hypothetical execution times for a CPU-GPU combination. As more rows are assigned to the CPU (increasing x-axis values), the time for the CPU part (Step 2) increases while the GPU part (Step 1) decreases. Assuming perfect overlap of Steps 1 and 2, the hybrid execution time is the maximum of these two times *plus* the data exchange overhead (Step 3). To beat the presumably faster GPU-only code, we ideally want (a) sufficiently small exchange overhead, and (b) the absolute value of the slope of the CPU and GPU lines to be roughly equal. Therefore, we expect a hybrid implementation will not lead to speedups overall if there is a large gap between CPU and GPU speeds or a high transfer overhead.

For our multi-GPU and multi-CPU/multi-GPU variants, we apply the same basic principles. To use multiple GPUs in parallel, the host must dedicate one thread per GPU and manage GPU resources separately within each thread.

Figure 6 is essentially a performance model that can be used to guide multi-CPU/multi-GPU work distribution in practical settings. The component and overhead curves that vary as a function of work distribution serve as the input to the model. These curves could be non-linear or empirically statistically modeled using off-line experiments, and quickly evaluated at run-time to decide on a splitting.

3.4 Wild asynchronicity

In Section 3.2, we described a tuned GPU-only implementation that uses padding to reduce non-coalesced device memory accesses, a proper access pattern to reduce bank conflicts, and a judicious allocation of shared memory within a thread-block to increase occupancy. We shall refer to this tuned GPU-only variant with "conventional" synchronizations as the **TunedSync** variant.

Intuitively, Jacobi's method is an iterative algorithm that converges to a fixed point, so we might guess that there are many variations that could reach the same fixed point, including variations in which we do not strictly synchronize the entire grid (and therefore all processes) between iterations. Indeed, this notion can be mathematically formalized and conditions for convergence proved in a variety of fixed-point algorithms and relaxations of the strict global synchronization (beyond the scope of this paper, but see Section 2 for references). In this section, we consider such "wildly asynchronous" variations that attempt to improve performance further by *changing* the computation and synchronization.

3.4.1 Review: TunedSync

First, recall the basic tuned GPU implementation of Section 3.2 for an $n \times n$ grid executing T Jacobi iterations, assuming $R \times C$ blocks of unknowns assigned to each CUDA thread-block, with R threads allocated per thread-block.

```
1: // TunedSync:
2: Transfer $(n+2) \times (n+2)$ grid to device memory.
3: for $t \leftarrow 1 \ldots T$ do
4:     Execute the tuned GPU kernel (see below).
5:     (Implicit) Sync CPU/GPU; logically swap grids.
6: end for
```

The tuned kernel uses padding and thread assignment:

```
1: // Tuned GPU CUDA Kernel:
2: Declare $(R+2) \times (C+2)$ shared memory block.
3: Fetch elements from device memory to shared block,
   incl. fringes.
4: Synchronize threads (sync_threads) within the thread
   block.
5: Compute 1 iteration for $R \times C$ unknowns in parallel and
   write to device memory.
```

3.4.2 Asynchronous variations

TunedSync requires, at every iteration: (a) 1 "global" synchronization, which occurs implicitly when the GPU kernel returns; (b) one "local" synchronization (`sync_threads`); and (c) device memory reads and writes for every grid element. We seek variations that are exactly or even only approximately equivalent to TunedSync variant and that can reduce these costs.

Async 0 and the async factor, α: All asynchronous variations we consider are parameterized by α, an *asynchronicity factor*, or just "async factor." To explain α, consider **Async 0** in Figure 7, our first asynchronous variant,

```
 1: // Async 0:
 2: // Threads / block is $R$
 3: // Assign $R \times C$ unknowns to each thread-block
 4: Transfer $(n+2) \times (n+2)$ grid to GPU.
 5: for $t \leftarrow 1 \ldots T_{\text{eff}}/\alpha$ do
 6:     Execute Async 0 GPU-Kernel.
 7:     (Implicit) Sync CPU-GPU.
 8:     Logically swap grids.
 9: end for
10: Transfer $(n+2) \times (n+2)$ grid to GPU.
```

```
 1: // Async 0 GPU-Kernel:
 2: // Executes on all thread-blocks
 3: Declare two $(R+2) \times (C+2)$ shared memory grid
    blocks, $B_1$ and $B_2$.
 4: Fetch $(R+2) \times (C+2)$ elements from device memory
    into $B_1$.
 5: Copy just the fringes to $B_2$.
 6: sync_threads: Sync thread-block.
 7: // Inner $\alpha$ loop is "unrolled" by 2
 8: for $v \leftarrow 1 \ldots \alpha/2$ do
 9:     Compute 1 iteration in $B_1$, writing to $B_2$.
10:     sync_threads
11:     Write penultimate fringe from $B_2$ to device memory.
12:     sync_threads
13:     Fetch fringe elements from device memory to $B_2$.
14:     sync_threads
15:     Compute 1 iteration in $B_2$, writing to $B_1$.
16:     sync_threads
17:     Write penultimate fringe from $B_1$ to device memory.
18:     sync_threads
19:     Fetch fringe elements from device memory to $B_1$.
20:     sync_threads
21: end for
22: Compute one Jacobi step with elements in $B_1$.
23: Write results back the results to the device memory.
```

Figure 7: Algorithm: Async 0. This algorithm is the basic skeleton in which we consider removing synchronizations and/or device memory accesses to create other "fast-and-loose" variants. Here, the "penultimate fringe" is the boundary of unknowns (outermost ring of unknowns bordering the ghost cells) that neighboring thread-blocks will need.

which tries to replace the T global synchronizations with $\approx T/\alpha$ such synchronizations. Intuitively, we will use α as an approximate measure of the degree to which we are willing to allow threads to get "out of sync." Such a parameterization is necessary because the mathematical convergence theory requires it [10] but the CUDA model cannot guarantee that thread block execution will be interleaved.

Async 0 differs from TunedSync in three major ways.

1. Async 0 reduces the number of device memory accesses, operating on the $R \times C$ unknowns entirely in shared memory (lines 8–20).

2. Async 0 executes lesser number of global synchronizations, replacing them with some number of thread-block-level `sync_threads` calls that depends on α.

3. The *effective number of iterations* executed by Async 1 is $T_{\text{eff}} = \frac{T}{\alpha} \cdot (\alpha + 1)$, meaning that Async 0 performs slightly more flops than TunedSync (line 22).

At this stage, it is not obvious whether Async 0 should be faster or slower than TunedSync.

Async 1. This variant further reduces the number of synchronizations. The outer algorithm is identical to Figure 7, but we replace the thread-block GPU kernel with the one shown in Figure 8. Async 1 eliminates 4 of the 6 local synchronizations in Async 0 (Figure 7, lines 10, 12, 16, and 18; the remaining 2 local syncs are needed to obtain convergence). However, it also introduces non-determinism: individual threads could access or modify an unknown mix of old and new values through the asynchronous fringe element accesses. Since we have eliminated a number of synchronizations, we hope that we can trade additional flops

```
 1: // Async 1 GPU-Kernel:
 2: // Executes on all thread-blocks
 3: Declare two $(R+2) \times (C+2)$ shared memory grid blocks,
    $B_1$ and $B_2$.
 4: Fetch $(R+2) \times (C+2)$ elements from device memory
    into $B_1$.
 5: Copy just the fringes to $B_2$.
 6: sync_threads: Sync thread-block.
 7: for $v \leftarrow 1 \ldots \alpha/2$ do
 8:     Compute 1 iteration in $B_1$, writing to $B_2$.
 9:     Write penultimate fringe from $B_2$ to device memory.
10:     Fetch fringe elements from device memory to $B_2$.
11:     sync_threads
12:     Compute 1 iteration in $B_2$, writing to $B_1$.
13:     Write penultimate fringe from $B_1$ to device memory.
14:     Fetch fringe elements from device memory to $B_1$.
15:     sync_threads
16: end for
17: Compute one Jacobi step with elements in $B_1$.
18: Write results back the results to the device memory.
```

Figure 8: Algorithm: Async 1. (GPU-Kernel only) This variant eliminates 4 of the 6 local syncs in Figure 7.

```
1: // Async 2 GPU-Kernel:
2: // Executes on all thread-blocks
3: Declare two $(R+2)\times(C+2)$ shared memory grid blocks, $B_1$ and $B_2$.
4: Fetch $(R+2)\times(C+2)$ elements from device memory into $B_1$.
5: Copy just the fringes to $B_2$.
6: sync_threads: Sync thread-block.
7: for $v \leftarrow 1\ldots\alpha/2$ do
8:    Compute 1 iteration in $B_1$, writing to $B_2$.
9:    sync_threads
10:   Compute 1 iteration in $B_2$, writing to $B_1$.
11:   sync_threads
12: end for
13: Compute one Jacobi step with elements in $B_1$.
14: Write results back the results to the device memory.
```

Figure 9: Algorithm: Async 2. (GPU-Kernel only) This variant eliminates the fringe writes and reads in lines 9, 10, 13, and 14 of Figure 8.

```
1: // Async 3 GPU-Kernel:
2: // Executes on all thread-blocks
3: Declare one $(R+2)\times(C+2)$ shared memory grid block, $B$.
4: Fetch $(R+2)\times(C+2)$ elements from device memory into $B$.
5: sync_threads: Sync thread-block.
6: for $v \leftarrow 1\ldots\alpha$ do
7:    Compute 1 iteration in $B$, writing to $B$.
8:    Write penultimate fringe from $B$ to device memory.
9:    Fetch fringe elements from device memory to $B$.
10: end for
11: Compute one Jacobi step with elements in $B$.
12: Write results back the results to the device memory.
```

Figure 10: Algorithm: Async 3. This "most wild" variant replaces the two shared memory grid blocks with 1, and eliminates all local synchronization.

via $T_{\text{eff}} > T$ for less time while still achieving the same level of accuracy as TunedSync with T iterations.

Async 2. This variant, shown in Figure 9, has the same number of synchronizations as Async 1 but eliminates fringe reads and writes to reduce the number of device memory accesses. We expect to need a larger T_{eff} than Async 1 but hope it is offset by the speed improvements due to fewer device memory accesses.

Async 3. Our final variant maintains just a single shared memory grid block, rather than two, as shown in Figure 10. This variant is the "most wild" in that we eliminate all local synchronization. This method is similar in spirit to Gauss-Seidel iteration—grid points can use the latest updated values—except that there is no deterministic sequential ordering of updates.

4. RESULTS AND DISCUSSION

We use the hardware evaluation platforms shown in Table 1. There are two systems: (1) a dual-socket 2.33 GHz dual-core Intel E6550 "Conroe" processor and two NVIDIA GPUs, a Tesla C1060 and a Quadro FX 570; and (2) a quad-socket quad-core AMD Opteron 8350 "Barcelona" processors

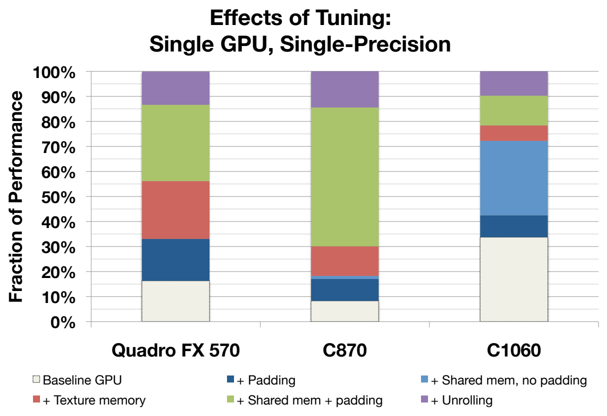


Figure 12: Cumulative impact of various tuning techniques on our implementation's final (best) performance.

and an NVIDIA Tesla C870. We use gcc 4.3.2 with the `-O4 -mtune=native` flags to compile our CPU implementations, and the CUDA 2.0 SDK for our GPU implementations.

Our implementation can take $f_{i,j}$ in Figure 1 to be either a separate grid containing arbitrary values or an inline function evaluated for any (i, j) at run-time. For our experiments, we use an inline function corresponding to a spike in the middle of the domain.

We use 5 flops per unknown when computing Gflop/s.

For reference, the baseline CPU implementations described in Section 3.1 achieve up to 4.8 Gflop/s on 16 cores of the Barcelona platform.

4.1 GPU implementations

We manually tuned GPU implementations on all three GPU cards, using the techniques of Section 3.2. Figure 11 shows we achieve up to 62.8% of empirical streaming bandwidth on the Quadro FX 570, up to 78.5% on the C870, and up to 98.6% on the C1060, when the grid fits within the GPU's device memory. These performance numbers translate into up to 1.96 Gflop/s, 22.8 Gflop/s, and 37.1 Gflop/s, respectively. However, performance is greatly diminished as the problem size falls below $n = 1024$.

On the C1060, we achieve 17.0 Gflop/s in double-precision and effectively sustain a bandwidth of 90.3% of the empirical streaming bandwidth. Compared to 98.6% bandwidth for single-precision, the difference reflects the penalty of having to pack and unpack double values.

We use a 8×8 thread block with 64×8 unknowns per thread-block to reach the best performance. These particular values minimize non-coalesced memory accesses and yield high occupancy values. It is important to get the block size right; for example, using a 16×8 block size increases the execution time by a factor of 1.7× over the optimal 64×8 case on the C1060 (in single-precision).

Which of our tuning techniques had the most impact on performance? Figure 12 breaks down the final tuned GPU performance into its parts for $n = 4096$. (Performance was largely independent of the number of iterations, T, ignoring initial and final host-device copies.) Looking across generations (oldest = FX 570, newest = C1060), there are no

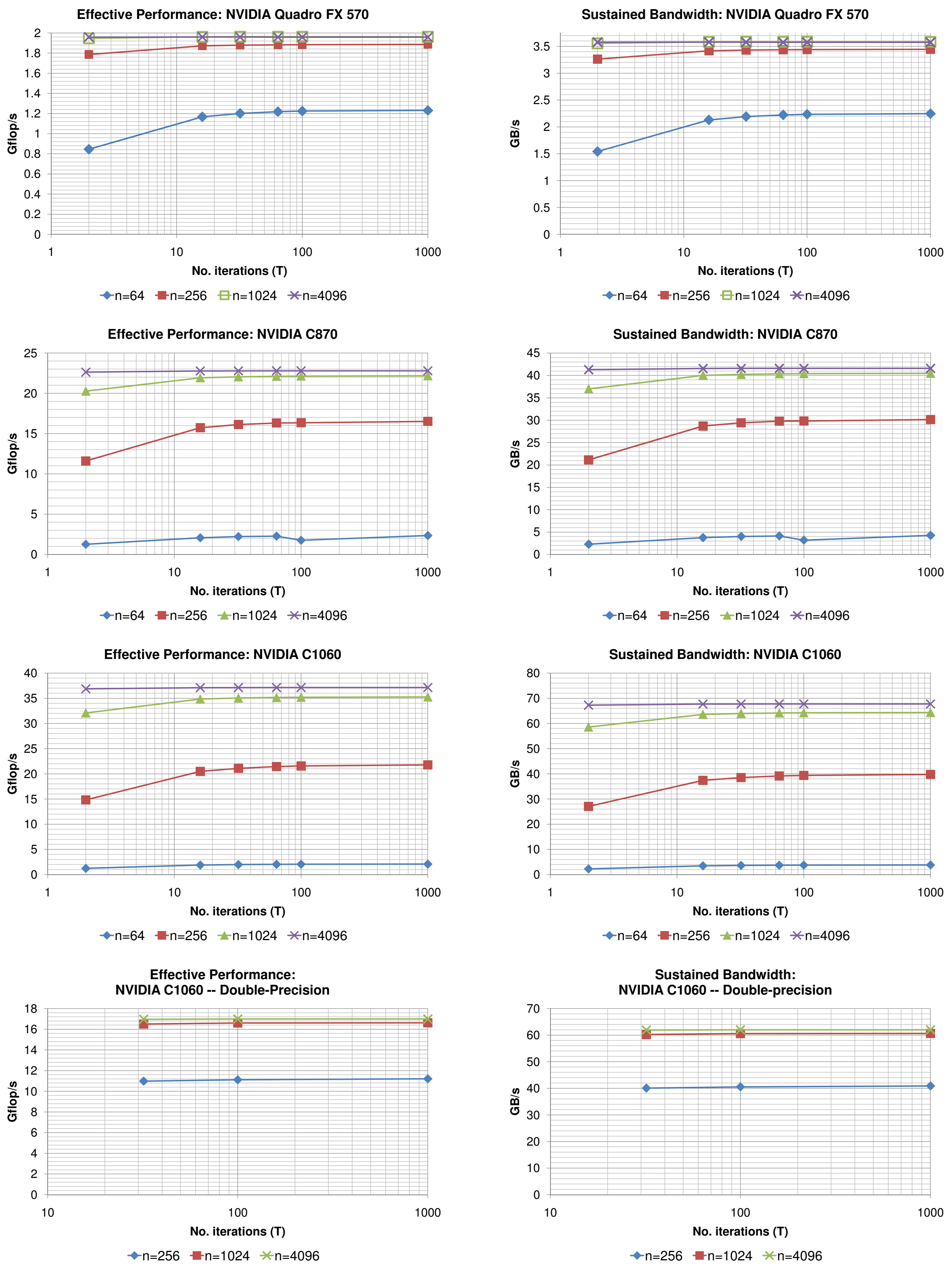


Figure 11: Sustained performance (Gflop/s) and bandwidth (GB/s) on various GPU hardware, as a function of grid size (n) and number of iterations (T). Top 3 rows show single-precision data for the NVIDIA Quadro FX 570, C870, and C1060, respectively. The bottom row shows double-precision data for the NVIDIA C1060.

Feature	NVIDIA Tesla C1060	NVIDIA Tesla C870	NVIDIA Quadro FX 570	Intel Core2Duo E6550 “Conroe”	AMD Opteron 8350 “Barcelona”
Number of multiprocessors	30	16	2	2	4
Total no. of cores	240	128	16	4	16
Peak bandwidth GB/s	102	76.8	12.8	10	21.6
Empirical streaming bandwidth (GB/s)	68.7	53.0	5.5	4.7	9.9
Double-precision?	Yes	No	No	Yes	Yes
Peak GFlop/s (Single-precision)	933	512	44	74.6[1]	256 [2]
Peak GFlop/s (Double-precision)	78	N/A	N/A	37.8	128

Table 1: Hardware platforms used in our experimental evaluation. “Empirical streaming bandwidth” measured using NVIDIA’s `bandwidthTest` utility and McCalpin’s STREAM Triad, as appropriate. Note that NVIDIA’s bandwidth test benchmark reports “GB/s” assuming 1 GB = 1024^3 bytes, whereas we instead use the more conventional method of computing the rate via “bytes times 10^{-9} divided by time.”

clearly discernable trends. For example, shared memory with padding—which reduces non-coalesced accesses—is absolutely necessary on the two older cards, but provides a relatively small improvement on the C1060. It is also worth mentioning that our experiments with prefetching data in shared memory did not yield better performance.

The preceeding data ignore the *initial* and *final* data transfers between host and device, which are critical in practice. We analyze these transfers in Figure 13. When $n = 4096$ and $T = 32$ on our Barcelona+C870 platform, these transfers account for just under 44% of the total execution time, a substantial cost. (The effective host-to-device transfer rate is 1.2 GB/s.) Figure 13 (right) examines the total execution time as a function of T, to see when a GPU implementation—including the transfers—can beat a multicore CPU implementation. In this case, we need at least 5 iterations to match a tuned 16-thread CPU implementation, and 60–100 iterations to hide most of the transfer cost.

4.2 Hybrid CPU/GPU execution

We consider two hybrid CPU/GPU implementations in Figure 14: Conroe + Quadro FX 5703 and Barcelona + C870. We see a small 8% improvement from a hybrid implementation on the relatively slow Conroe + Quadro FX system (11% of rows assigned to the CPU), which qualitatively resembles our expectations in Figure 6. This small improvement is due to the CPU being much slower than the GPU, as reflected in the relative slopes of the CPU-part and GPU-part lines. On the Barcelona + C870 system, we see no improvement, in large part due both to the gap in CPU/GPU processing power and the data transfer overheads.

We tested our multi-GPU (no CPU) on two systems. On a FX 570 + C1060 system we observed no speedup, as is evident in Figure 15. This result is due largely to the $\approx 18\times$ gap between the fast and slow GPU. On a system with 2 C1060 cards, we were able to see a speedup of $\approx 1.8\times$

Though disappointing, these hybrid-case findings were not surprising in light of our performance model (Figure 6). At the very least, it should be simple to instantiate this model automatically given a new hardware platform, to determine whether a hybrid implementation could outperform a homogeneous multi-CPU or single-GPU variant.

4.3 Wildly asynchronous implementations

We compare Async 1, 2, and 3 implementations of Section 3.4 on a $n = 4096$ grid for a single-GPU C1060. The results appear in Figure 16. In all cases, we run the Async algorithms for as large a value of T_{eff} as is necessary to achieve the same accuracy as TunedSync with $T = 1000$ iterations. We measure accuracy as the maximum absolute value of the relative error computed against the solution after many (4096) iterations. We evaluate both (a) the bottom-line speedup relative to TunedSync and (b) the relative increase in the effective iteration count, T_{eff}. Because the algorithms are non-deterministic, we include error bars, though in all cases they are too small to see clearly.

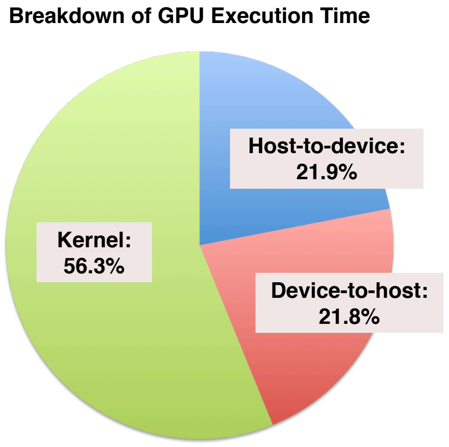
Async 1 is never faster than TunedSync. Though there are fewer global syncs, there are several local syncs and device memory accesses for fringe elements. As shown in Figure 16 (right), Async 1 also always has $T_{\text{eff}} > T$.

Async 2, by contrast, exceeds the performance of TunedSync by up to $1.3\times$ for async factors $\alpha \leq 15$. This best speedup, which occurs for $\alpha = 6$ (Figure 16 (left)), exists despite the fact that Async 2 is performing $1.7\times$ as many flops (Figure 16 (right)).

The most aggressive algorithm, Async 3, achieves the best speedups, being up to $2.5\times$ faster than TunedSync. This improvement is due to reduced synchronization, reduced device memory accesses, and accelerated convergence.

5. CONCLUSIONS AND FUTURE WORK

Though several studies exist on stencil kernels for GPU systems, our examination of hybrid and heterogeneous designs, simple performance models for these designs, and an exploration of fast-and-loose asynchronous algorithms extend our collective understanding in the area. In particular, two aspects of our work seem especially relevant to future systems, which will feature many heterogeneous cores.



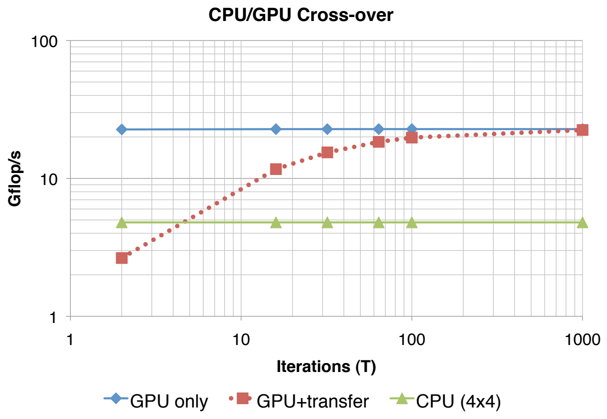
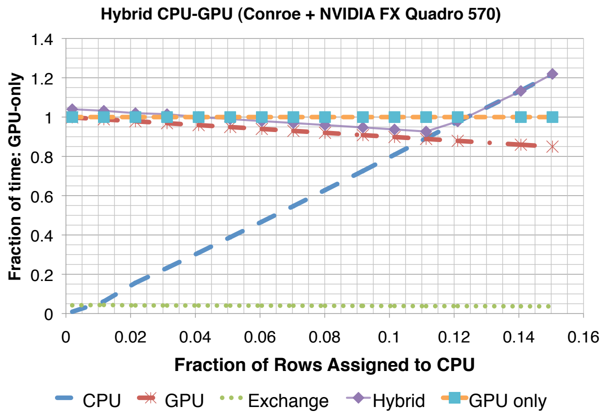


Figure 13: (*Left*) Breakdown of the total execution time using a single GPU (NVIDIA C870). Grid size is $n = 4096$, and number of iterations $T = 32$. A substantial amount of time is spent just transfering data between the host memory and GPU memory. (*Right*) Number of iterations needed for the single-GPU (NVIDIA C870) performance to exceed the baseline parallel CPU performance (Barcelona 4×4) when the initial and final grid transfers between host and device are taken into account. Grid size is $n = 4096$.



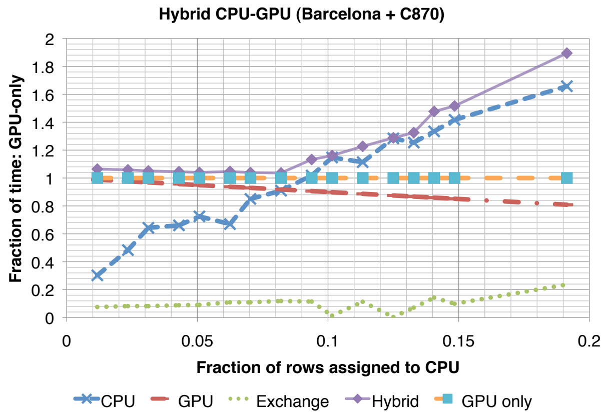
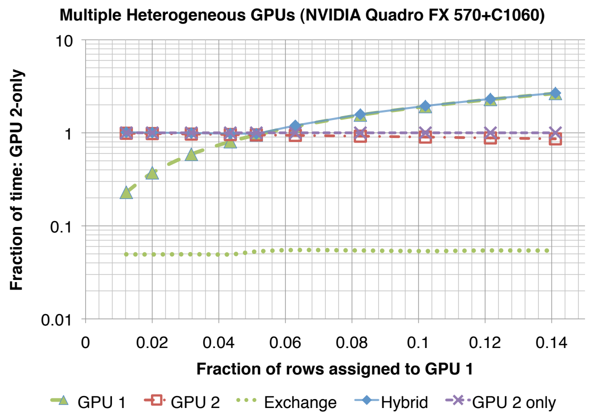


Figure 14: Measured time for hybrid multi-CPU and single-GPU implementations, normalized to GPU-only time. Compare to our hybrid performance model, illustrated in Figure 6. (*Left*) Intel single-socket dual-core Conroe + NVIDIA Quadro FX 570. At approximately 11% of rows assigned to the CPU, there is a small $\approx 8\%$ speedup over the GPU-only code. (*Right*) AMD quad-socket quad-core Barcelona (16 threads) + NVIDIA C870. The hybrid code never beats the GPU-only code due to the data exchange/synchronization.



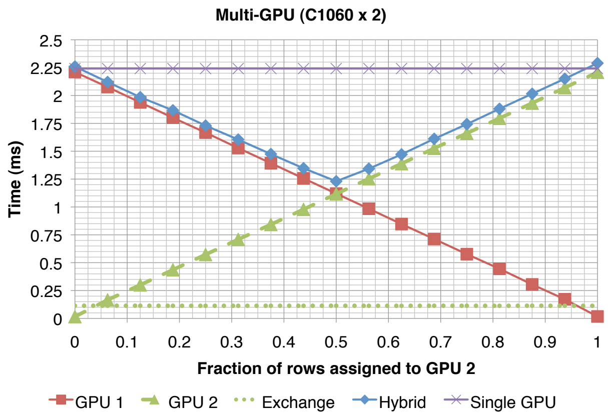
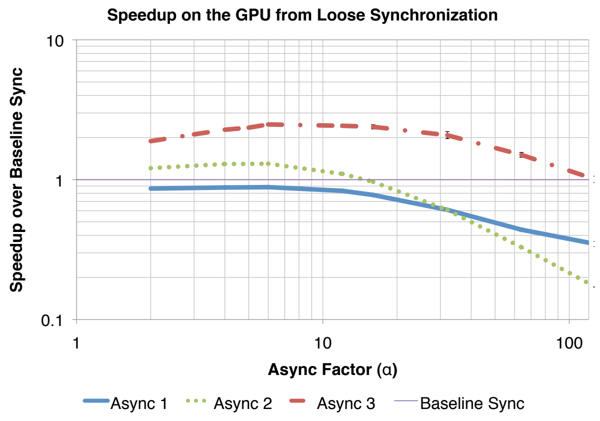


Figure 15: Measured multiple GPU performance, (*Left*) One NVIDIA Quadro FX 570 ("GPU 1") and One NVIDIA C1060 ("GPU 2"). Because of the large gap between the performance of the two GPUs ($\approx 18\times$ difference, not shown), the optimal fraction does not beat the GPU 2-only code. (Compare to Figure 6.) (*Right*) Two NVIDIA Tesla C1060 cards. At approximately 50% of rows assigned to GPU 1 there is a speedup of $\approx 1.8\times$ over the GPU-only code.



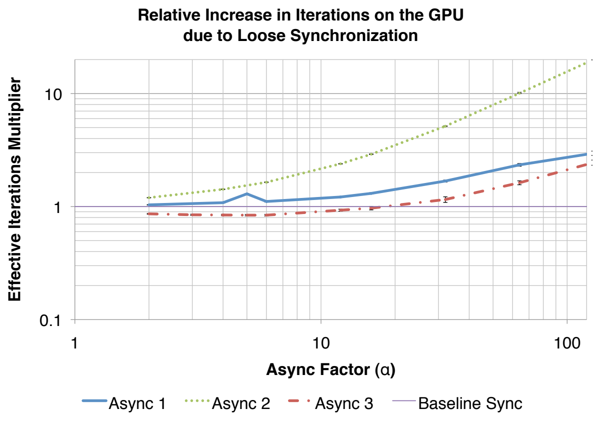


Figure 16: Comparison of asynchronous implementations on the NVIDIA C1060, for $n = 4096$ and $T = 1000$. (*Left*) Speedup relative to the synchronized baseline. (*Right*) Relative increase in effective iterations required to reach the same level of accuracy as the tuned synchronized GPU baseline (synchronized baseline = 1).

First, we consider real costs, such as host-to-device transfer time, that have sometimes been omitted in prior work. The corresponding performance model for hybrid designs, which we presented mostly to explain why our hybrid implementations did not yield significant speedups, is very simple but informative, and should be relevant to the design of future implementations and systems.

Secondly, our evaluation of the classical idea of chaotic relaxation, though highly speculative, would seem to be a fruitful future direction in the regime of high sychronization and communication costs. The mathematical survey of Frommer and Szyld [10] summarizes known convergence conditions, and references demonstrated applications in linear and non-linear solvers and optimization, inverse problems in geophysics, and power network analysis, among others. Extending these ideas to other domains will of course require careful analysis, but we believe our results suggest that now is an appropriate time to take a fresh look into this area of research.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF) under award number 0833136, NSF TeraGrid allocation CCR-090024, and a grant from the Defense Advanced Research Projects Agency (DARPA). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or DARPA.

6. REFERENCES

[1] NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`, June 2008.

[2] R. Amorim, G. Haase, M. Liebmann, and R. W. dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration. Technical Report SFB-Report No. 2008-025, Universität Graz, Graz, Austria, December 2008.

[3] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, April 1978.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proc. ACM SIGGRAPH*, San Diego, CA, USA, 2003. `http://www.multires.caltech.edu/pubs/GPUSim.pdf`.

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600 (LAPACK Working Note 191), Innovative Computing Laboratory, University of Tennessee Knoxville, September 2007. `http://www.netlib.org/lapack/lawnspdf/lawn191.pdf`.

[6] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, pages 116–125, San Diego, CA, USA, June 2007.

[7] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2(2):199–222, April 1969.

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

[9] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[10] A. Frommer and D. B. Szyld. On asynchronous iterations. *J. Comp. Appl. Math.*, 123(1–2):201–216, November 2000.

[11] M. Giles. Jacobi iteration for a Laplace discretization on a 3D structured grid. `http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/laplace3d.pdf`, April 2008.

[12] K. G. Kale and A. J. Lew. Parallel asynchronous variational integrators. *Int. J. Numer. Meth. Engng.*, 70:291–321, 2007.

[13] H. Karimabadi, J. Driscoll, Y. A. Omelchenko, and N. Omidi. A new asychronous methodology for modeling of continuous systems: Breaking the curse of the Courant condition. *J. Comp. Phys.*, 205:755–775, 2005.

[14] A. Lew, J. Marsden, M. Ortiz, and M. West. Asychronous variational integrators. *Arch. Rational Mech. Anal.*, 2003.

[15] J. J. Nutaro. *Parallel discrete event simulation with application to continuous systems*. PhD thesis, University of Arizona, 2003.

[16] J. C. Strikwerda. A probabilistic analysis of asynchronous iteration. *Linear Algebra and Its Applications*, 349(1–3):125–154, January 2002.

[17] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, Austin, TX, USA, November 2008.

[18] S. Williams, L. Oliker, R. Vuduc, J. Shalf, and K. A. Yelick. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, Reno, NV, USA, November 2007.