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

Presenter:

Zhang. Jiayue

10M54041

Reference

- Sundaresan Venkatasubramanian, Richard W. Vuduc Georgia Institute of Technology
- International Conference on Supercomputing ’09

Outline

- Background
- Code Design and Tuning
  - CPU
  - GPU
  - Hybrid
- Results and Discussion
- Conclusions and Future Work
- My Comments

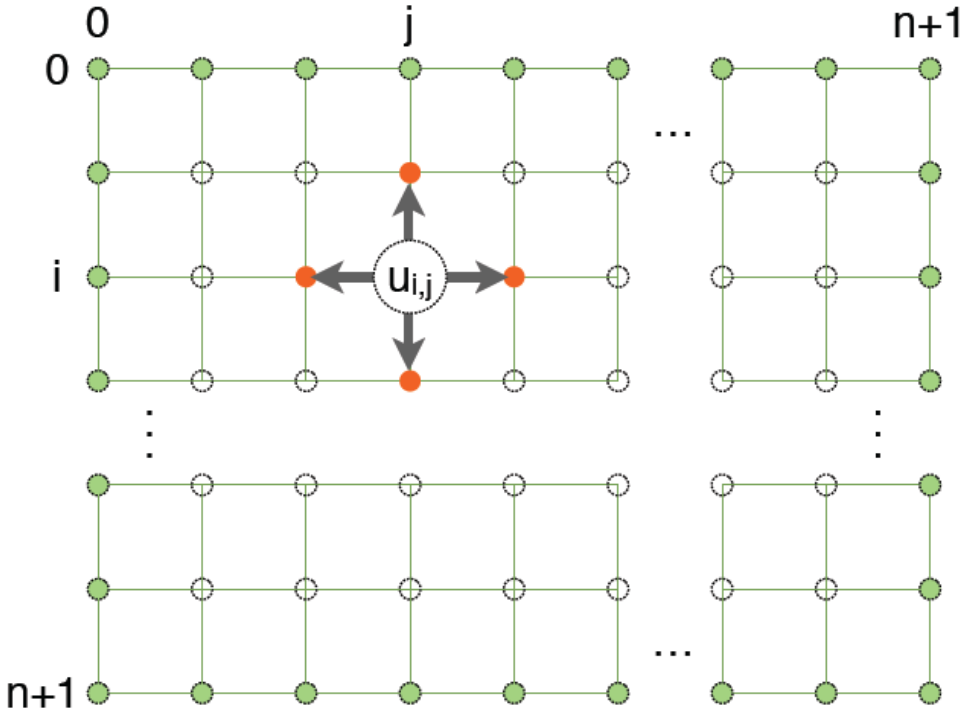
Background

- 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.

Background

- Problem:

  Jacobi's method for the 2-D Poisson equation



Background

- Poisson's equation:

  A **partial differential equation** with broad utility in electrostatics, mechanical engineering and theoretical physics.

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

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

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

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

Background

- Jacobi method:

  Jacobi's method for the 2-D Poisson equation

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** $t \leftarrow 1, 2, \ldots T$ **do**
4: **for** $i \leftarrow 1 \ldots N$ **do**
5: **for** $j \leftarrow 1 \ldots N$ **do**
6: $U_{i,j}^{t+1} \leftarrow \frac{1}{4} \cdot (U_{i+1,j}^t + U_{i-1,j}^t + U_{i,j+1}^t + U_{i,j-1}^t + h^2 \cdot F_{i,j})$
7: **end for**
8: **end for**
9: **end for**

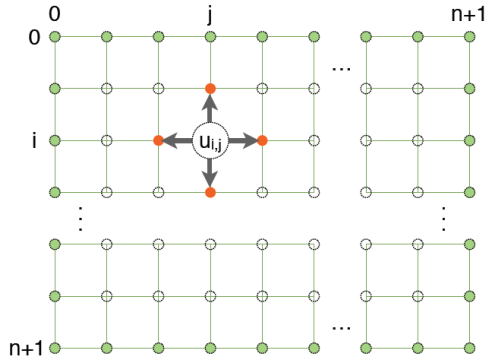
Background

- Recent studies
  - GPU-only
  - Synchronized designs

- Our main contributions:
  - CUDA platform tuning lessons
  - Hybrid multi-CPU/multi-GPU implementation
  - Asynchronous parallelism

Code Design and Tuning

- We consider both sequential and parallel Pthreads-based implementations as our CPU-based baselines.
- CPU-sequential version:



1: $U^0 \leftarrow 0$ // **Initializes an** $(N+2) \times (N+2)$ **grid**
2: // **For each iteration,** t (T **iterations in all**)
3: **for** $t \leftarrow 1, 2, \ldots T$ **do**
4: **for** $i \leftarrow 1 \ldots N$ **do**
5: **for** $j \leftarrow 1 \ldots N$ **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**

Code Design and Tuning

- CPU-parallel version:
- Single-program multiple data (SPMD) style

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

Code Design and Tuning

- CPU implementation tuning:
  - Compiler SIMD optimization (verified by the assembly code)
  - Bind threads to cores

    (Using pthread_attr_setaffinity_np())
  - Bind data blocks to sockets in a way that physically matches the layouts of the cores and sockets.

Code Design and Tuning

- CPU implementation tuning:

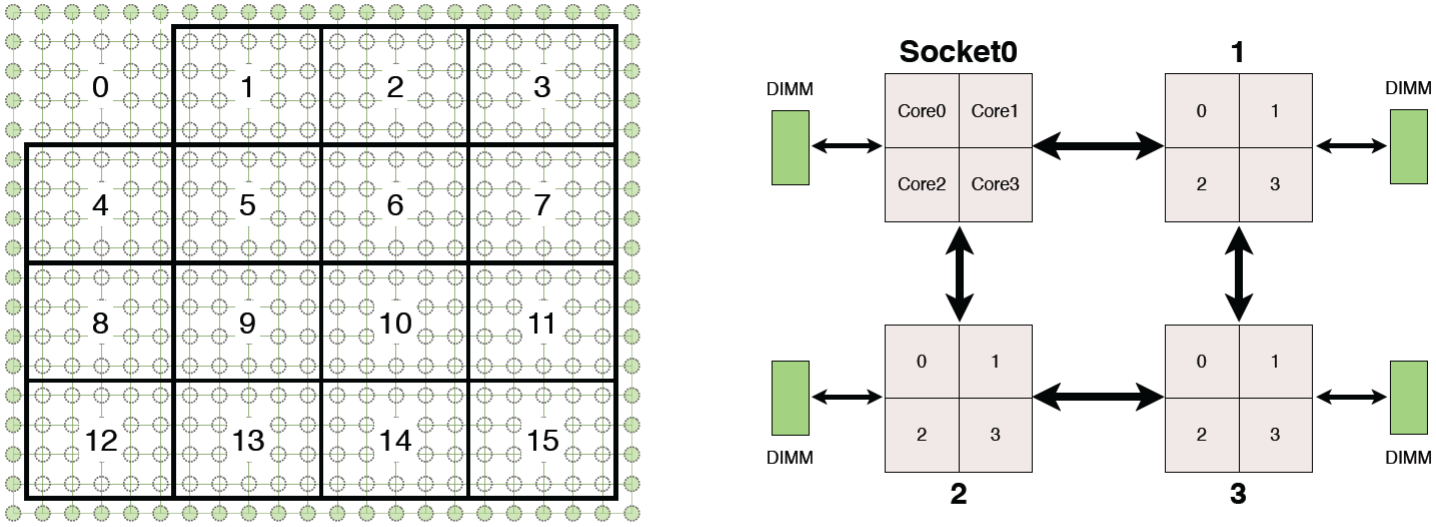


Figure 2: (*Left*) A 16×20 grid of unknowns (plus boundaries) partitioned into a 2-D blocked grid of 4×5 unknowns per block. (*Right*) A quad-socket NUMA system with quad-core processors.

  - NUMA: non-uniform memory access

Code Design and Tuning

- GPU implementation:

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.

Code Design and Tuning

- GPU implementation tuning:
  - Padding for coalescing memory access

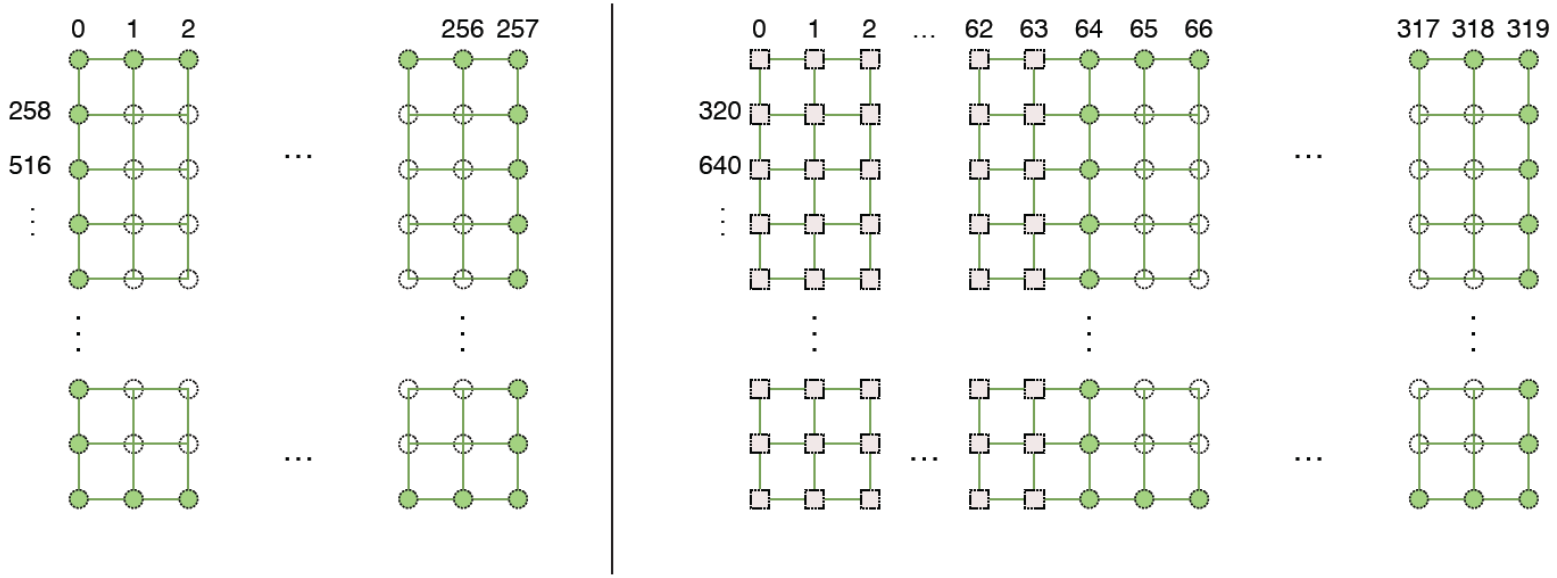


Figure 3: (*Left*) A conventional row-major layout, for $n = 256$ (258×258 grid), which could lead to costly non-coalesced memory accesses on a GPU. (*Right*) A padded row-major layout, for $n = 256$, avoids the problems of the conventional layout.

Code Design and Tuning

- GPU implementation tuning:
  - Shared memory, without padding.
    - Hold elements from neighboring blocks.
    - reduces the total number of device memory fetches by 3x and we can eliminate storage of the second grid.
  - Shared memory, with padding.
    - Avoid shared memory bank conflicts.
  - Texture memory.
    - Binding the global memory to 1-D textures as a cache.
  - Unrolling
    - Avoid shared memory bank conflicts.

Code Design and Tuning

- GPU implementation tuning:
  - A double-precision trick.
    - Using double-precision (8-byte words) leads to 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].

Code Design and Tuning

- Hybrid implementations:

1: // **Assign rows** $1 \ldots s$ **to the CPU(s),**
2: // **and rows** $s+1 \ldots n$ **to the GPU(s).**
3: **for** $t \leftarrow 1 \ldots T$ **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 $1 \ldots s$.
6: Step 3 (**Exchange data**): Transfer the boundary rows between CPU and GPU.
7: **end for**

Code Design and Tuning

- Hybrid implementations:

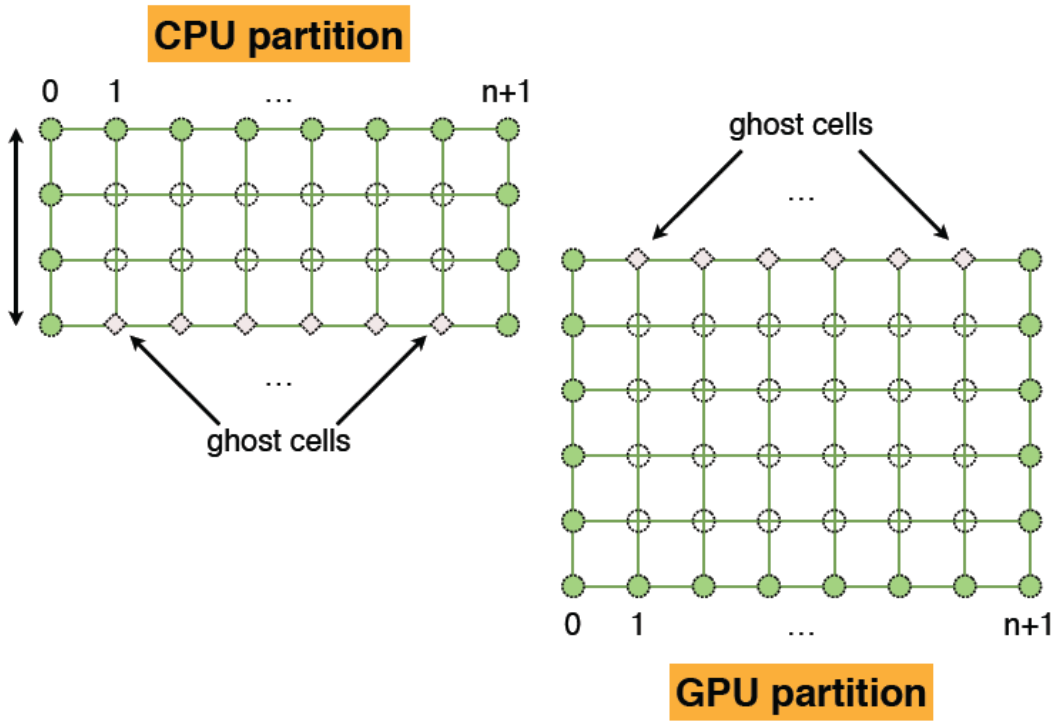


Figure 5: Block row partitioning used for the hybrid CPU/GPU implementation.

Code Design and Tuning

- Hybrid implementations tuning:

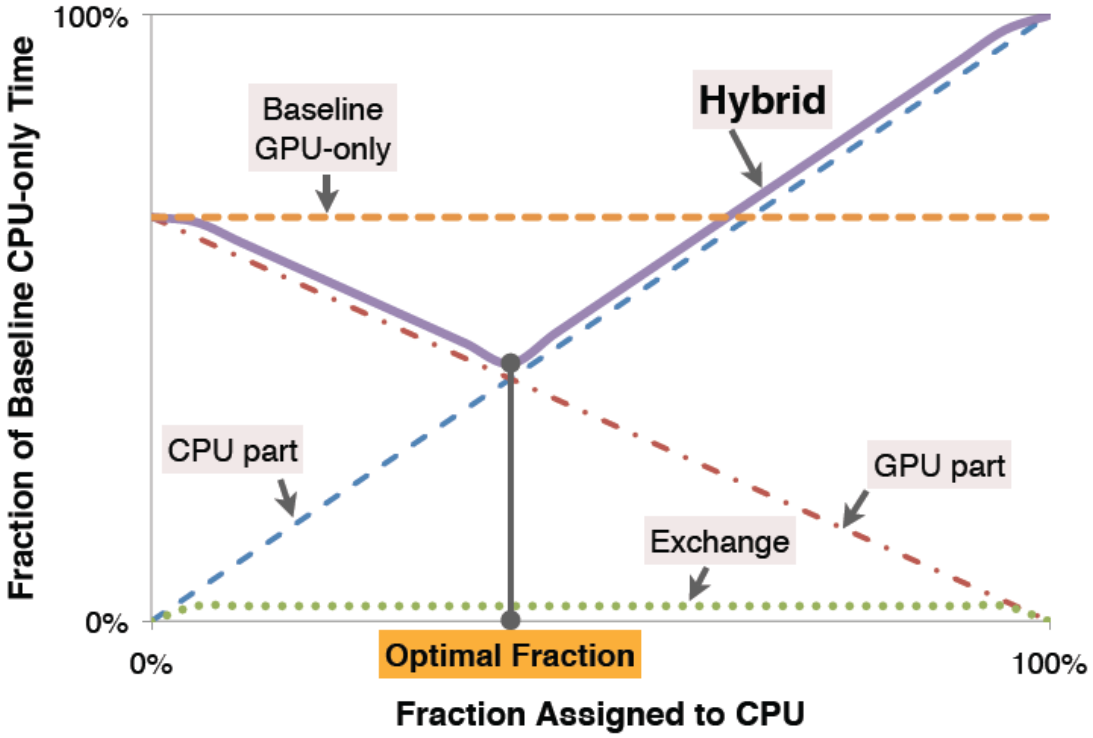


Figure 6: Illustration of expected performance of a hybrid CPU/GPU implementation.

Code Design and Tuning

- Hybrid implementations tuning:
  - Parameterized by α, an asynchronicity factor.
  - An approximate measure of the degree to which we are willing to allow threads to get “out of sync.”

Code Design and Tuning

- Hybrid implementations tuning:
  - Asynchronous variations 0

```
1: // Async 0:
2: // Threads / block is R
3: // Assign R×C unknowns to each thread-block
4: Transfer (n + 2) × (n + 2) grid to GPU.
5: for t ← 1 . . . T_eff/α do
6:     Execute Async 0 GPU-Kernel.
7:     (Implicit) Sync CPU-GPU.
8:     Logically swap grids.
9: end for
10: Transfer (n + 2) × (n + 2) grid to GPU.
```

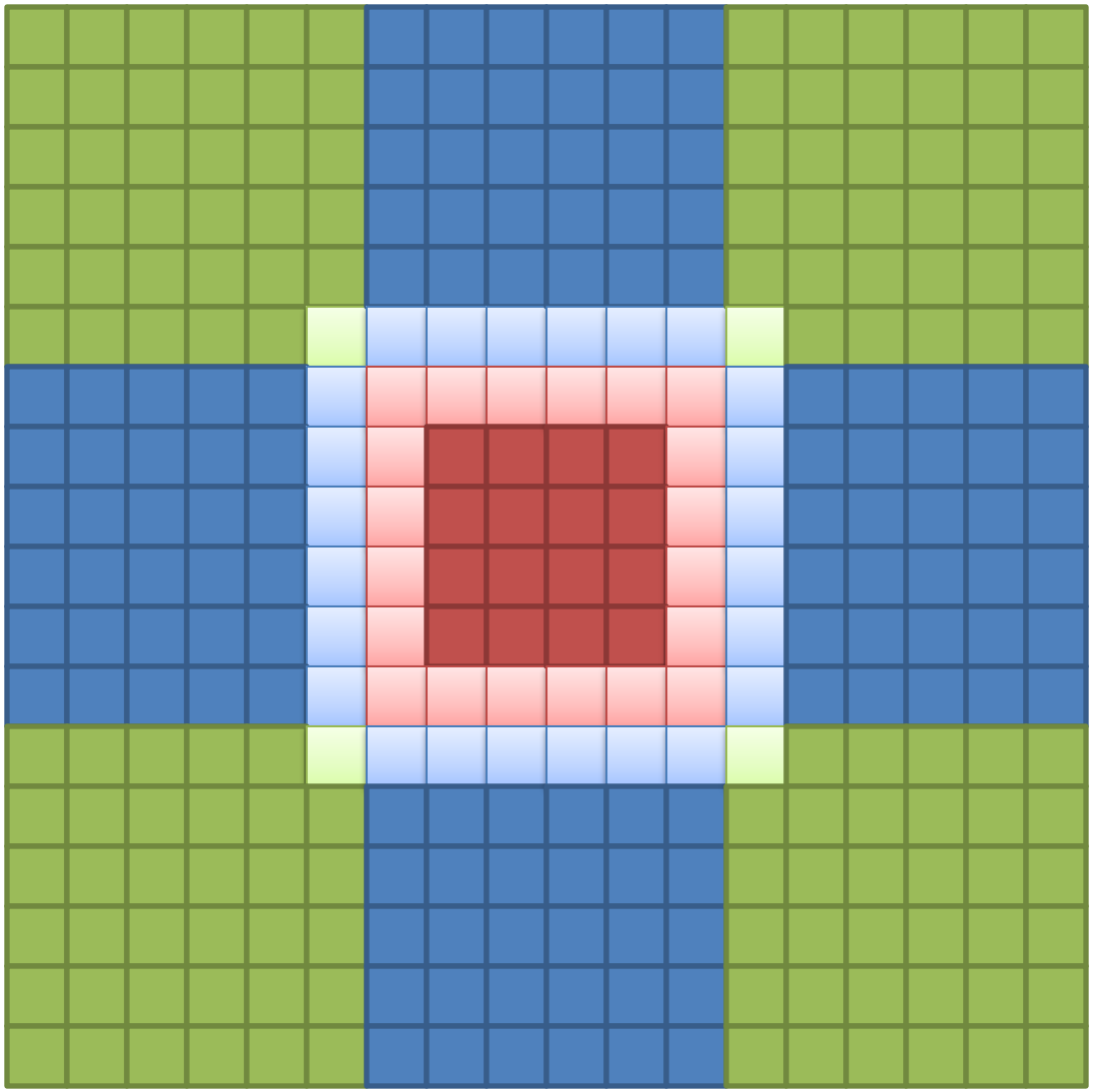
```
1: // Async 0 GPU-Kernel:
2: // Executes on all thread-blocks
3: Declare two (R + 2) × (C + 2) shared memory grid
   blocks, B_1 and B_2.
4: Fetch (R + 2) × (C + 2) elements from device memory
   into B_1.
5: Copy just the fringes to B_2.
6: sync_threads: Sync thread-block.
7: // Inner α loop is "unrolled" by 2
8: for v ← 1 . . . α/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.

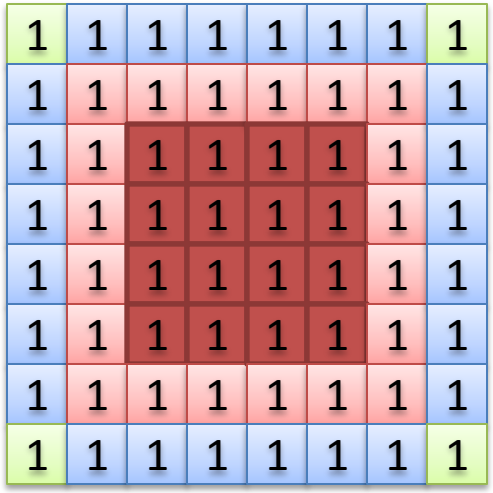
```
// Async 0:
// Threads / block is R
// Assign R×C unknowns to each thread-block
Transfer (n + 2) × (n + 2) grid to GPU.
for t ← 1 ... T_eff/α do
    Execute Async 0 GPU-Kernel.
    (Implicit) Sync CPU-GPU.
    Logically swap grids.
end for
Transfer (n + 2) × (n + 2) grid to GPU.
```

```
// Async 0 GPU-Kernel:
// Executes on all thread-blocks
Declare two (R + 2) × (C + 2) shared memory grid blocks, B_1 and B_2.
Fetch (R + 2) × (C + 2) elements from device memory into B_1.
Copy just the fringes to B_2.
sync_threads: Sync thread-block.
// Inner α loop is "unrolled" by 2
for v ← 1 ... α/2 do
    Compute 1 iteration in B_1, writing to B_2.
    sync_threads
    Write penultimate fringe from B_2 to device memory.
    sync_threads
    Fetch fringe elements from device memory to B_2.
    sync_threads
    Compute 1 iteration in B_2, writing to B_1.
    sync_threads
    Write penultimate fringe from B_1 to device memory.
    sync_threads
    Fetch fringe elements from device memory to B_1.
    sync_threads
end for
Compute one Jacobi step with elements in B_1.
Write results back the results to the device memory.
```

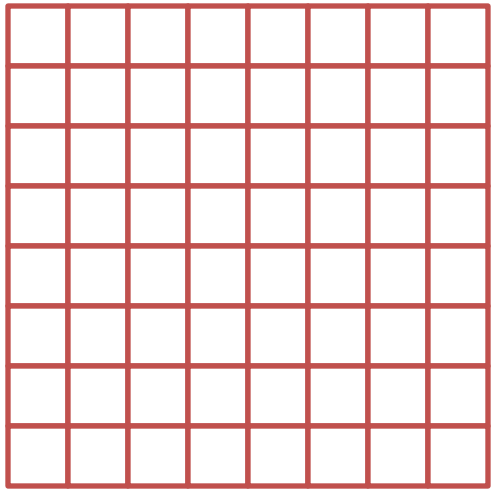
Figure 7: Algorithm: Async 0. This algorithm is the basic skeleton in which we consider removing synchronizations and/or device memory accesses to create other "fast-and-loose" variants. Here, the "penultimate fringe" is the boundary of unknowns (outermost ring of unknowns bordering the ghost cells)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



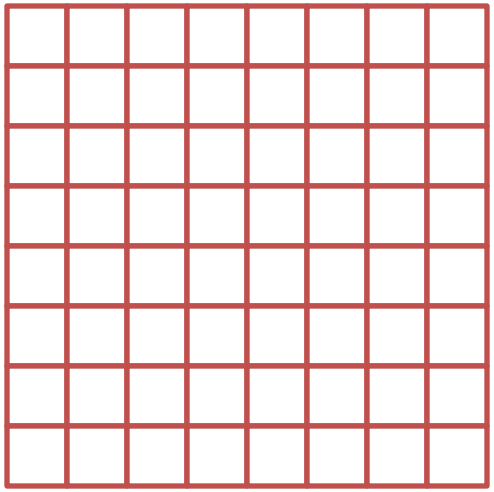
Blocks



Global Memory

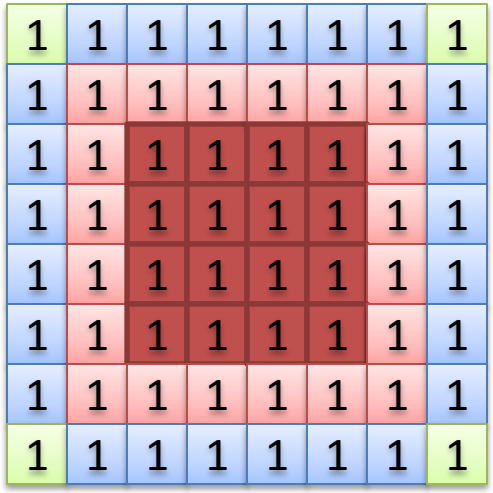


B1

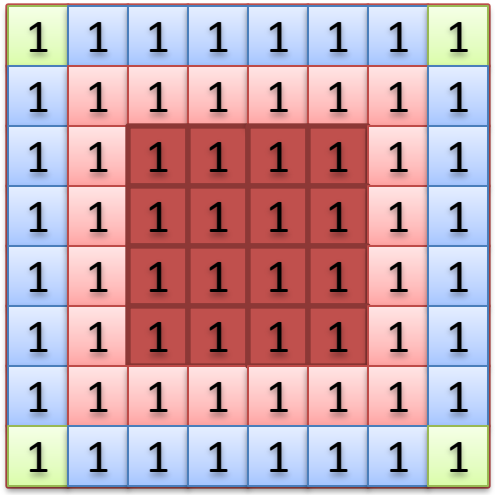


B2

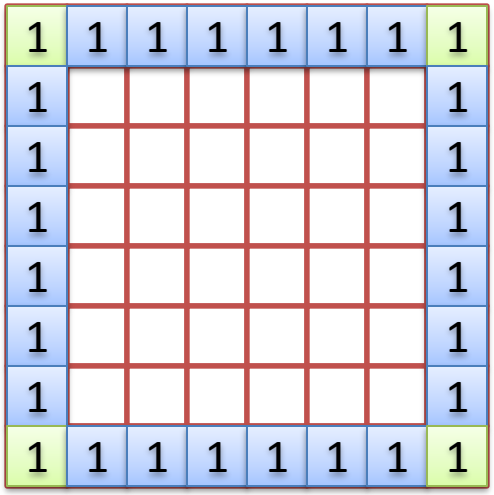
4: Fetch (R + 2) × (C + 2) elements from device memory into B1.
5: Copy just the fringes to B2.
6: sync_threads:



Global Memory

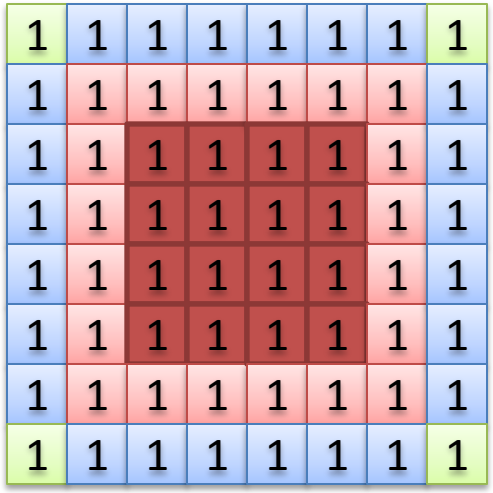


B1

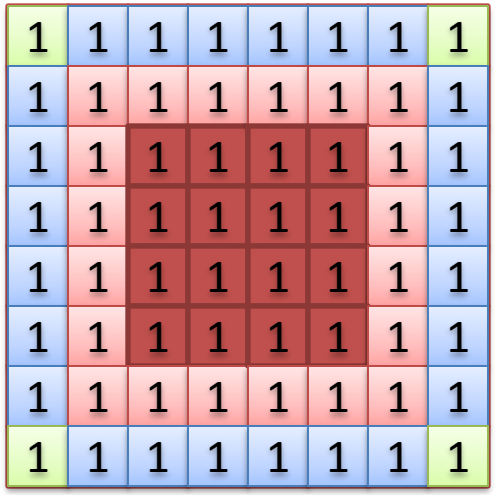


B2

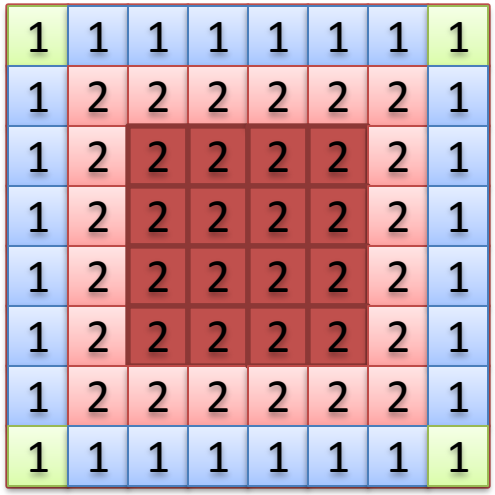
9: Compute 1 iteration in B1, writing to B2.
10: sync_threads



Global Memory

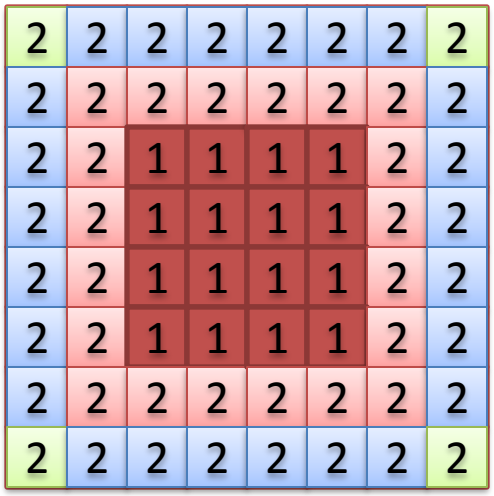


B1

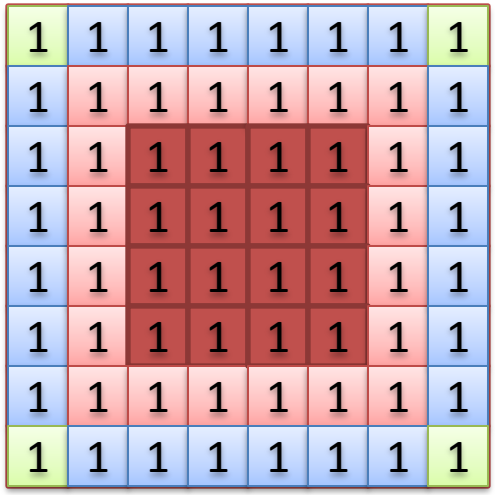


B2

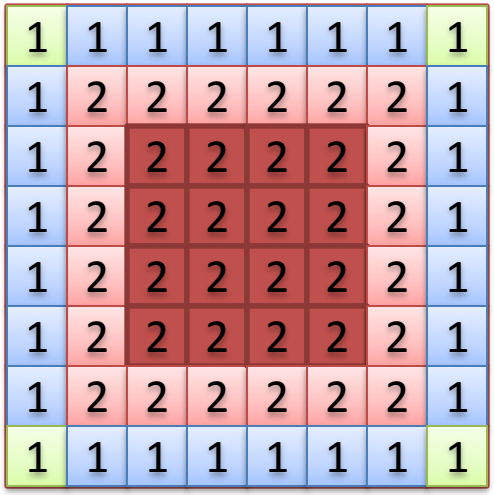
11: Write penultimate fringe from B2 to device memory.
12: sync_threads



Global Memory

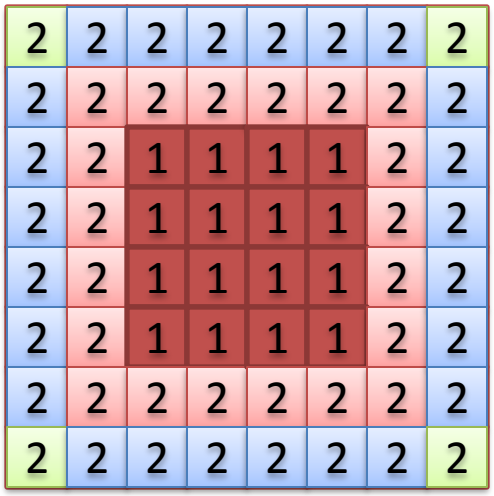


B1

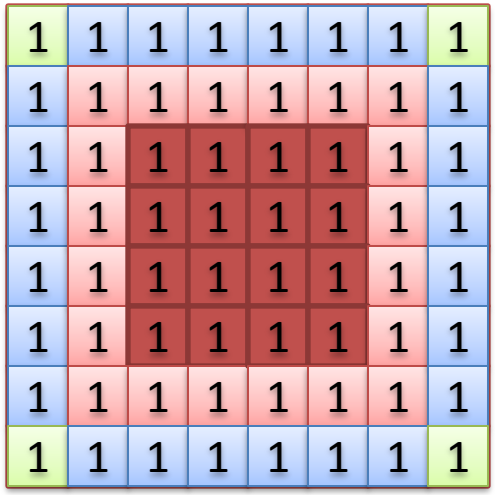


B2

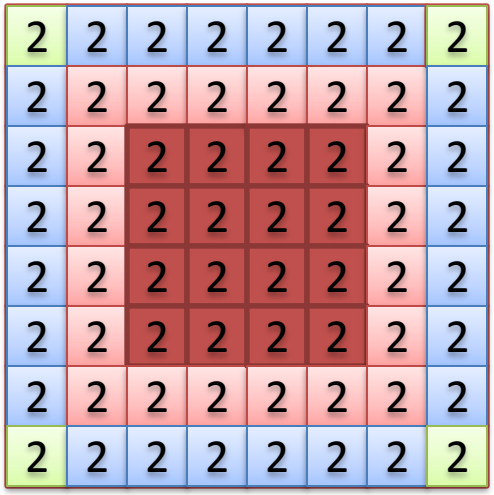
13: Fetch fringe elements from device memory to B2.
14: sync_threads



Global Memory

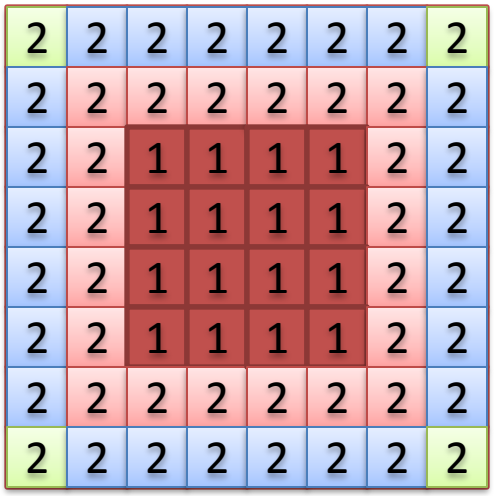


B1

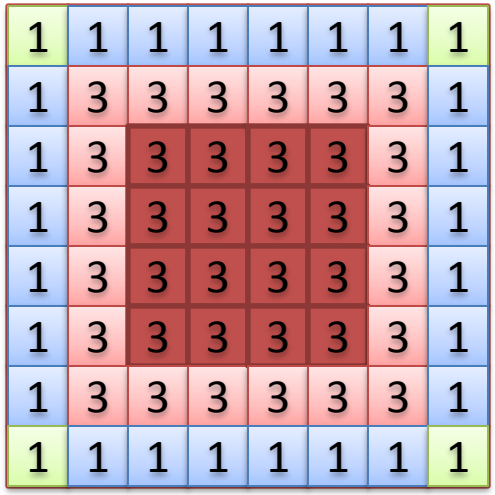


B2

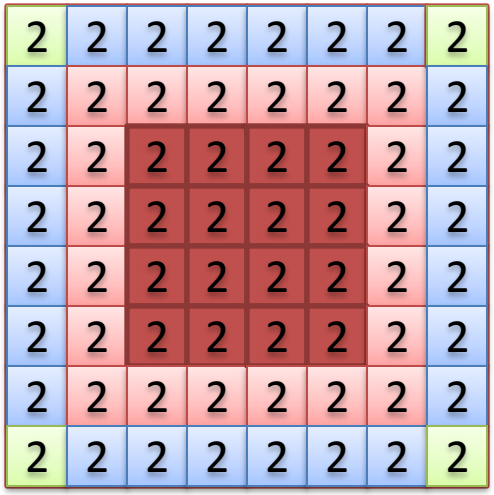
15: Compute 1 iteration in B2, writing to B1.
16: sync_threads



Global Memory

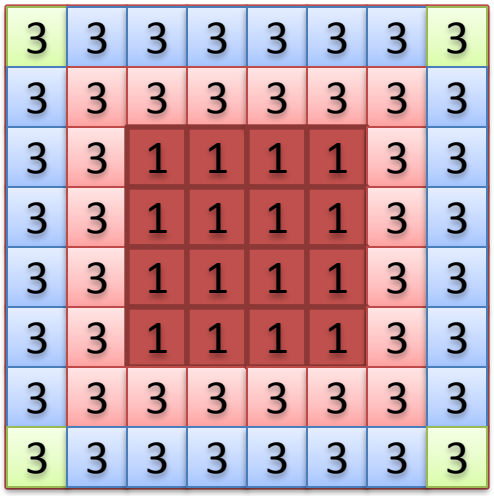


B1

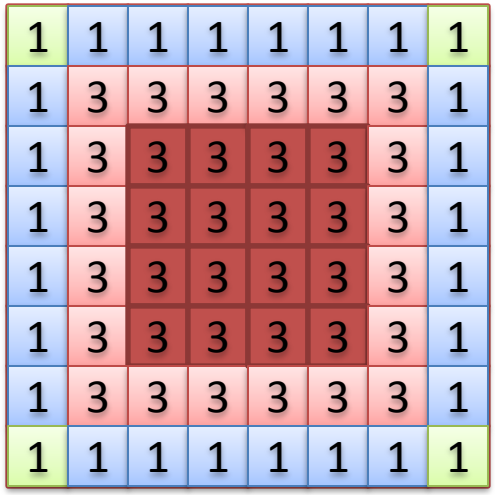


B2

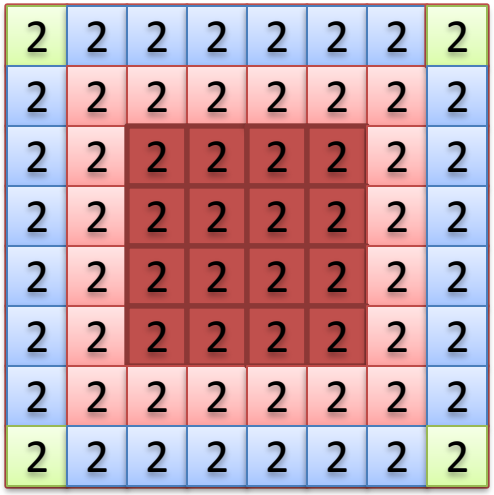
17: Write penultimate fringe from B1 to device memory.
18: sync_threads



Global Memory

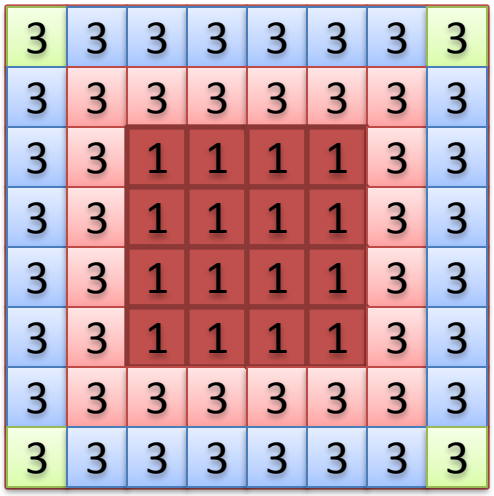


B1

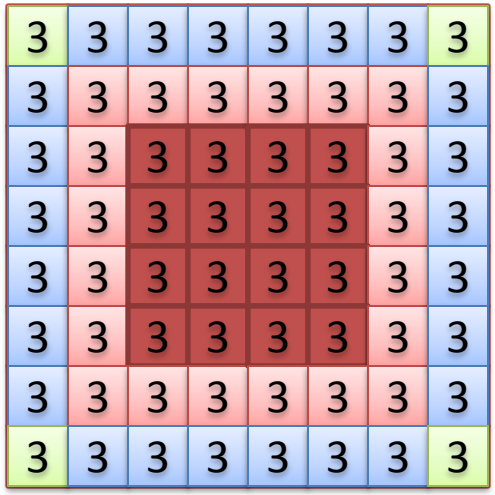


B2

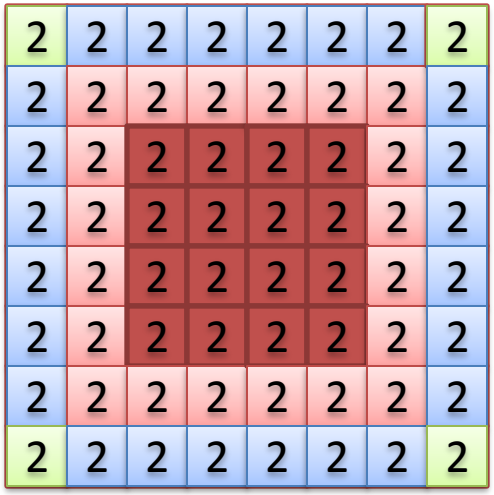
19: Fetch fringe elements from device memory to B1.
20: sync_threads



Global Memory

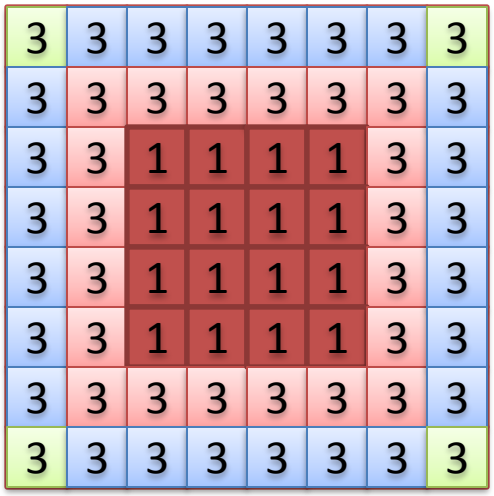


B1

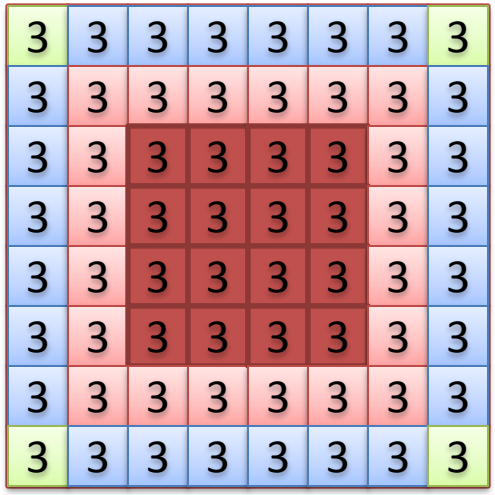


B2

22: Compute one Jacobi step with elements in B1.



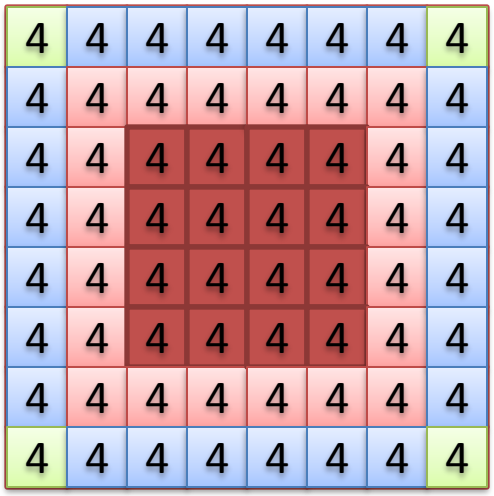
Global Memory



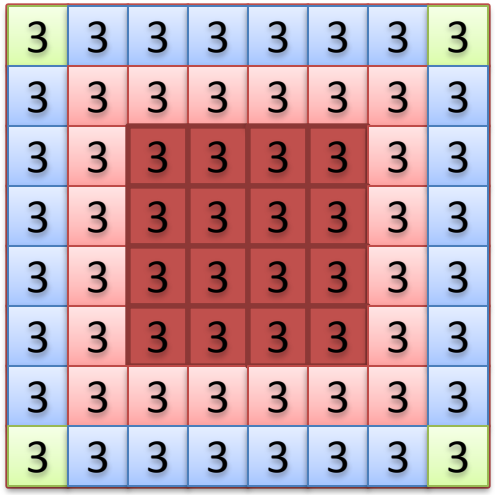
B1

B2

23: Write results back the results to the device memory.



Global Memory



B1

B2

Code Design and Tuning

- Hybrid implementations tuning:
  - Asynchronous variations 1

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 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.

Code Design and Tuning

- Hybrid implementations tuning:
  - Asynchronous variations 2

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 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.

Code Design and Tuning

- Hybrid implementations tuning:
  - Asynchronous variations 3

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.

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.

Results and Discussion

- Evaluation platforms
  - gcc 4.3.2 with "-O4 -mtune=native" flags
  - CUDA 2.0 SDK

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."

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."

Results and Discussion

- GPU

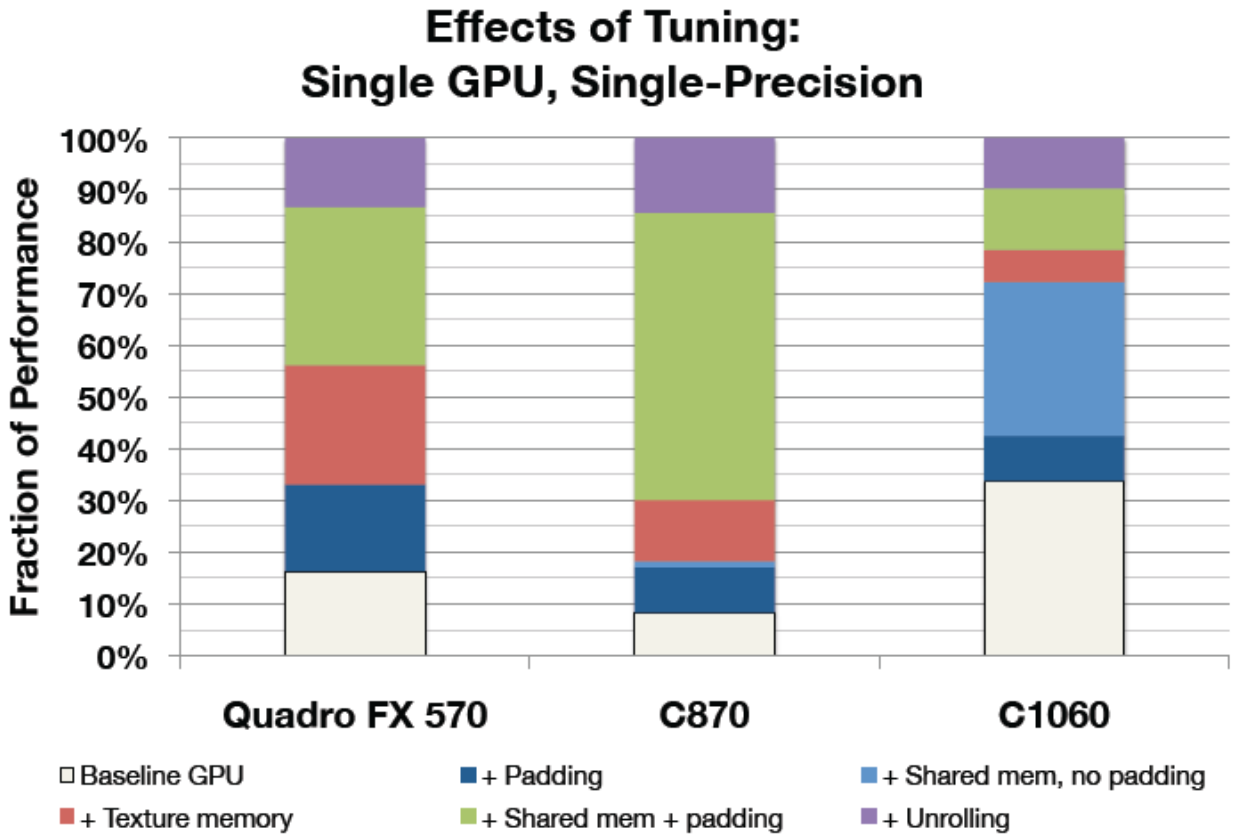
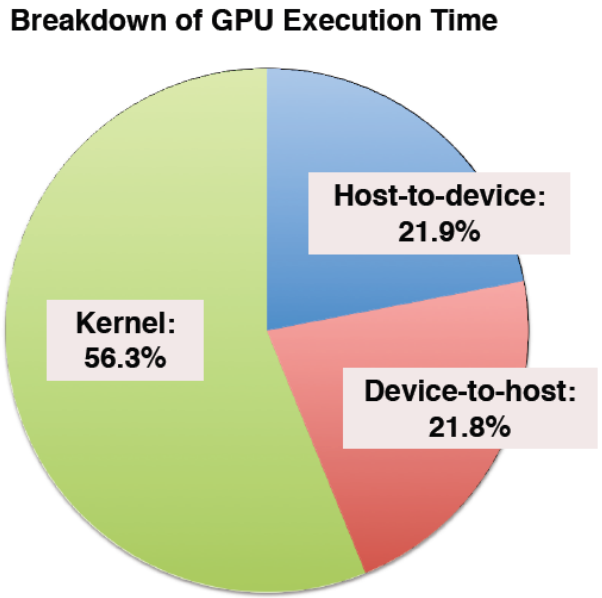


Figure 12: Cumulative impact of various tuning techniques on our implementation's final (best) performance.

Results and Discussion

- GPU



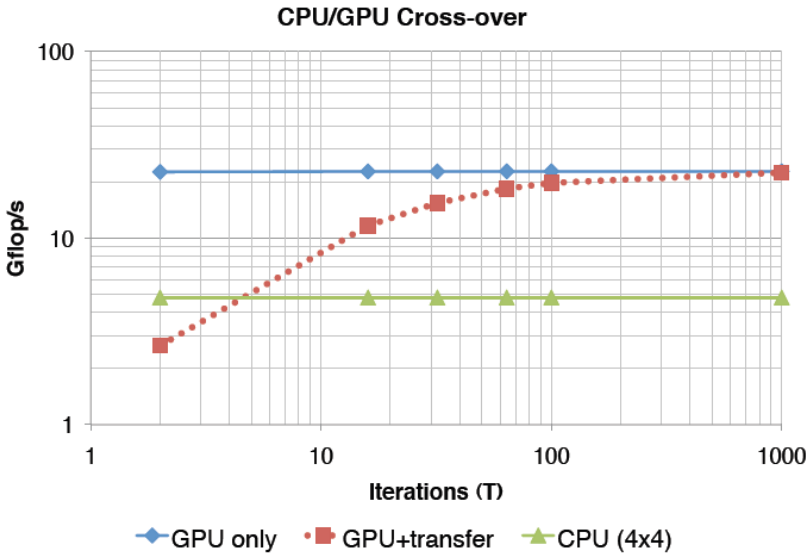
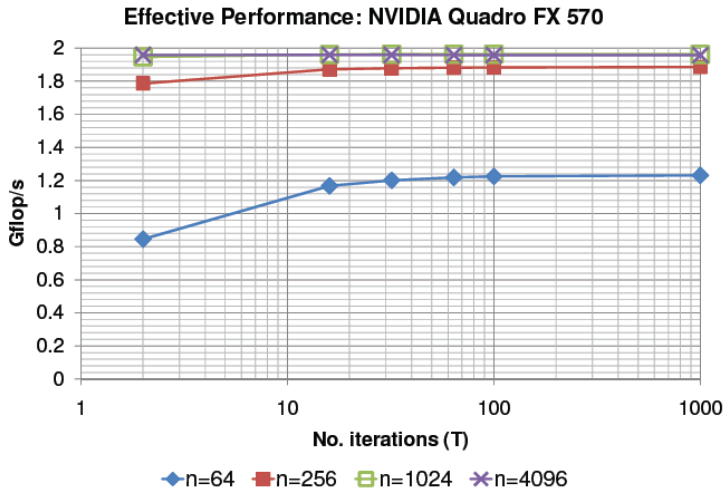
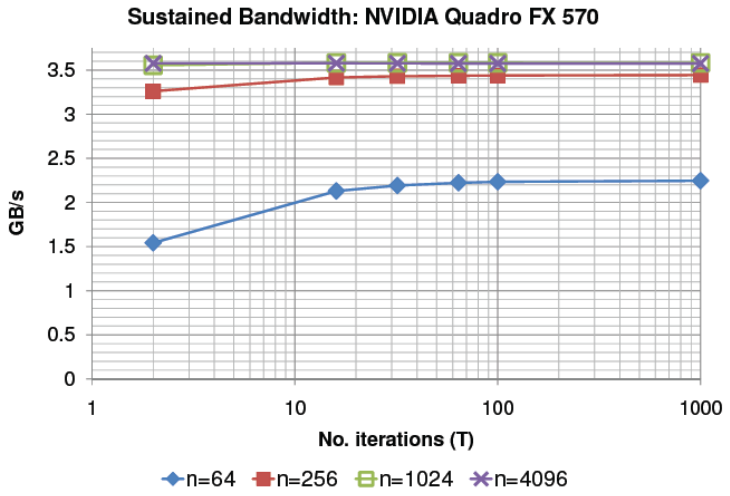


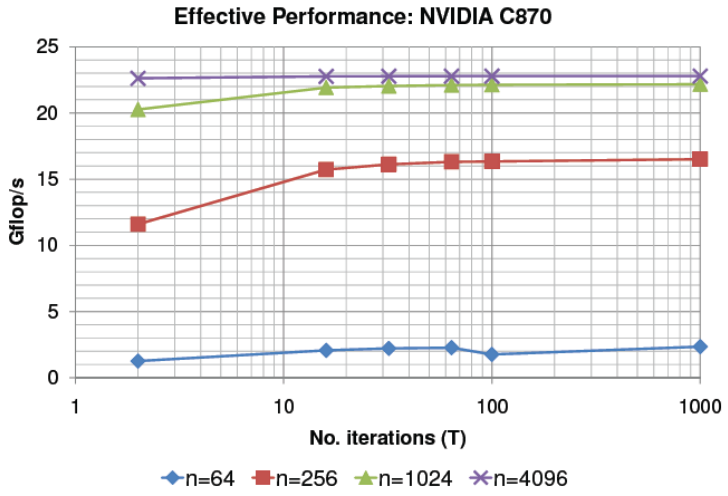
Figure 13: (*Left*) Breakdown of the total execution time using a single GPU (NVIDIA C870). Grid size is $n = 4096$, and number of iterations $T = 32$. A substantial amount of time is spent just transfering data between the host memory and GPU memory. (*Right*) Number of iterations needed for the single-GPU (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 $n = 4096$.

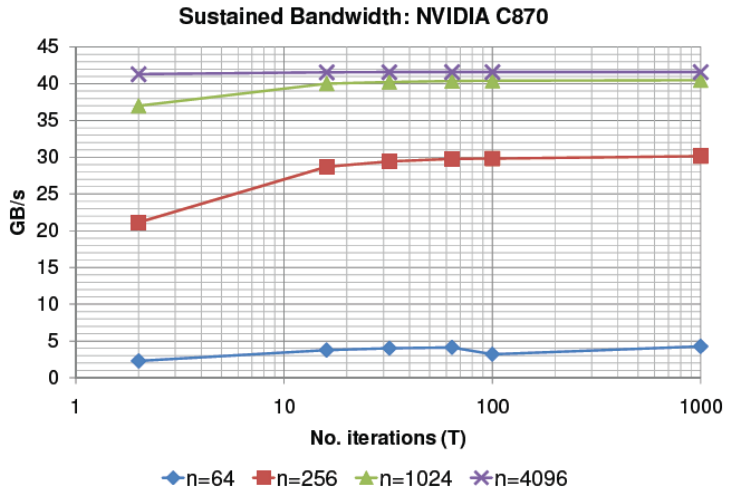
Results and Discussion

- GPU



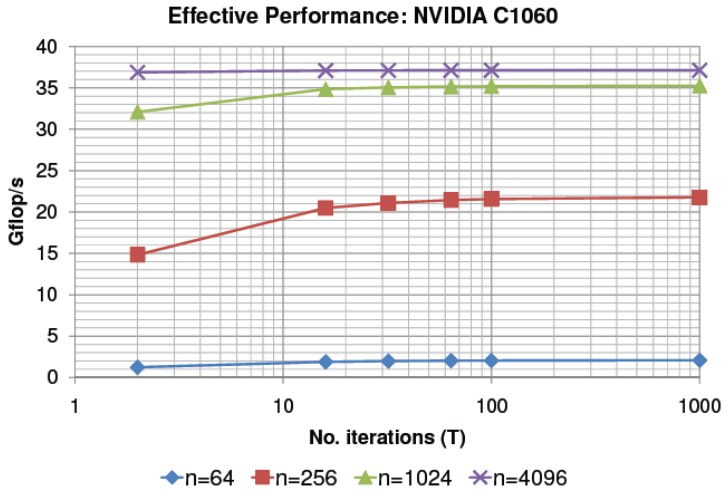


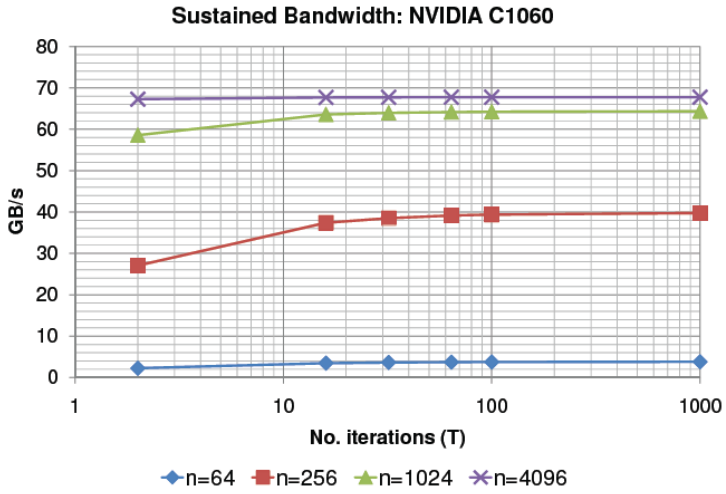


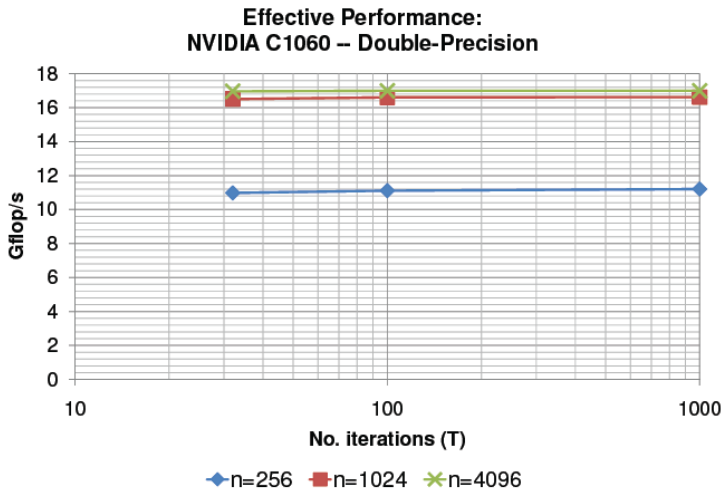


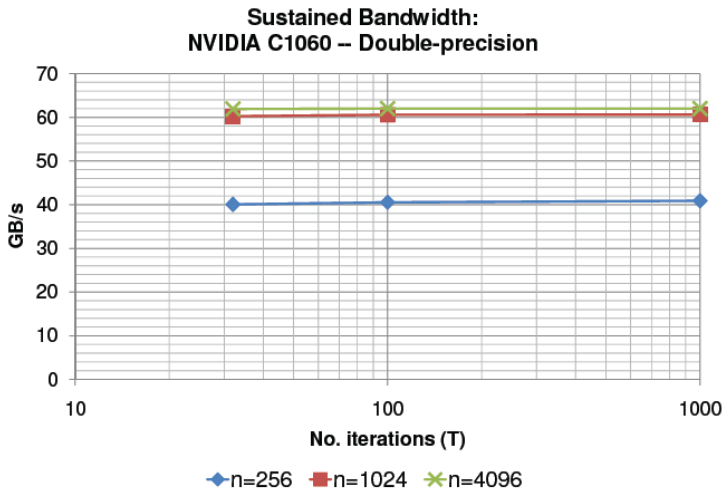
Results and Discussion

- GPU



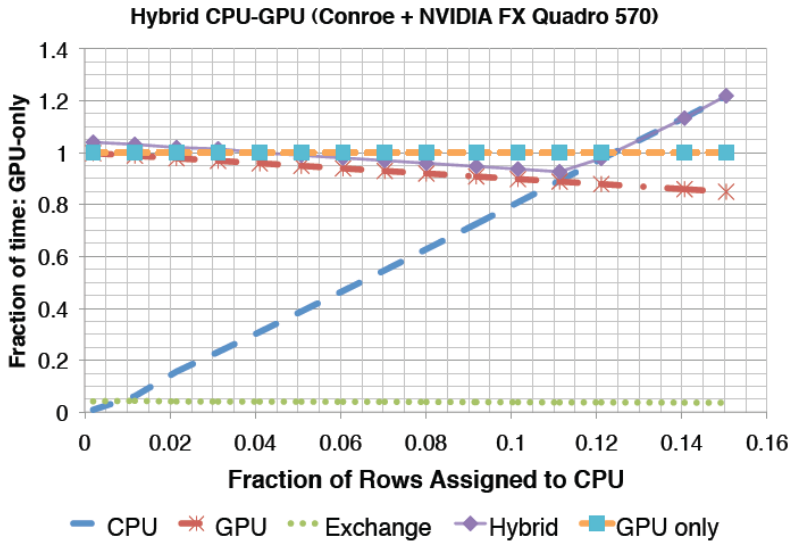






Results and Discussion

- Hybrid CPU/GPU execution



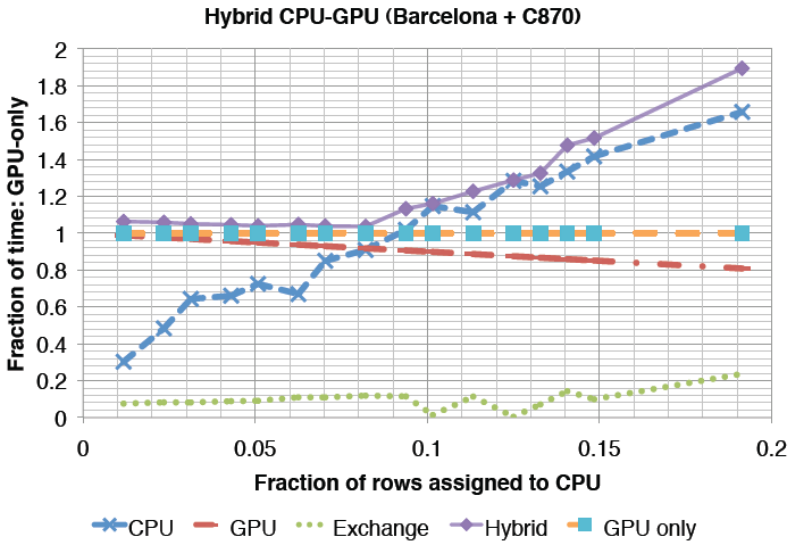
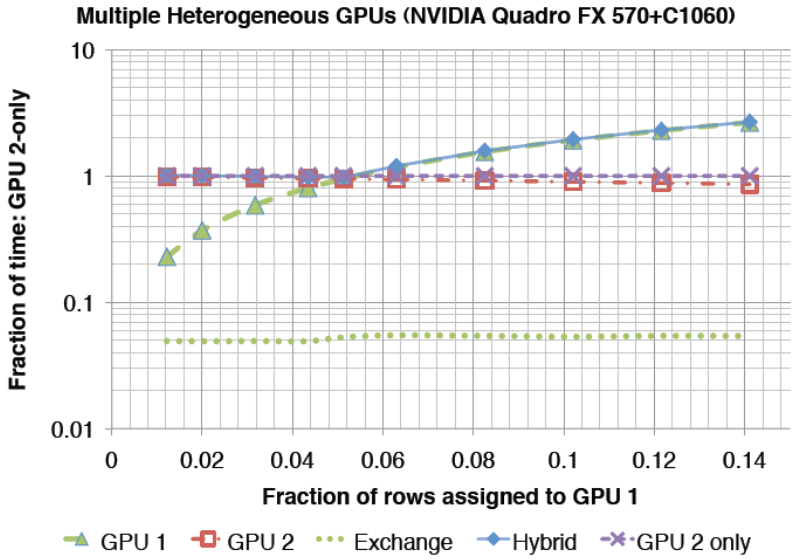


Figure 14: Measured time for hybrid multi-CPU and single-GPU implementations, normalized to GPU-only time. Compare to our hybrid performance model, illustrated in Figure 6. (*Left*) Intel single-socket dual-core Conroe + NVIDIA Quadro FX 570. At approximately 11% of rows assigned to the CPU, there is 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.

Results and Discussion

- Multi-GPU



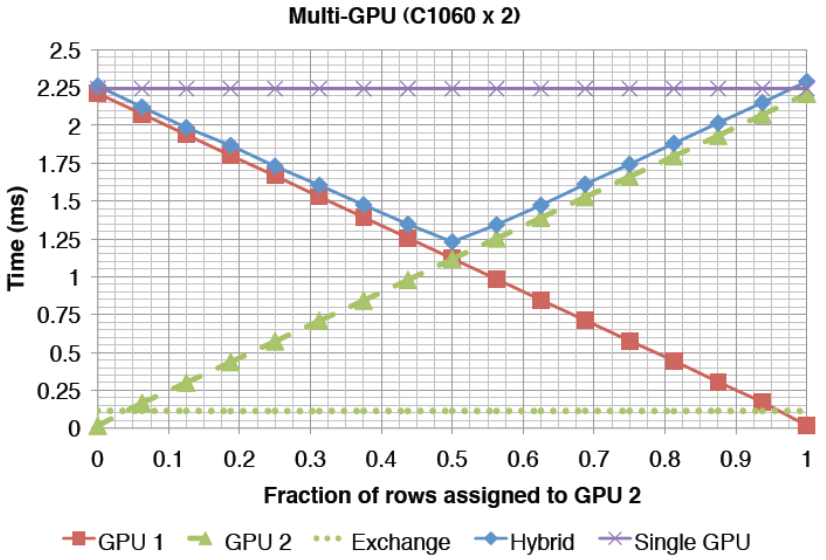
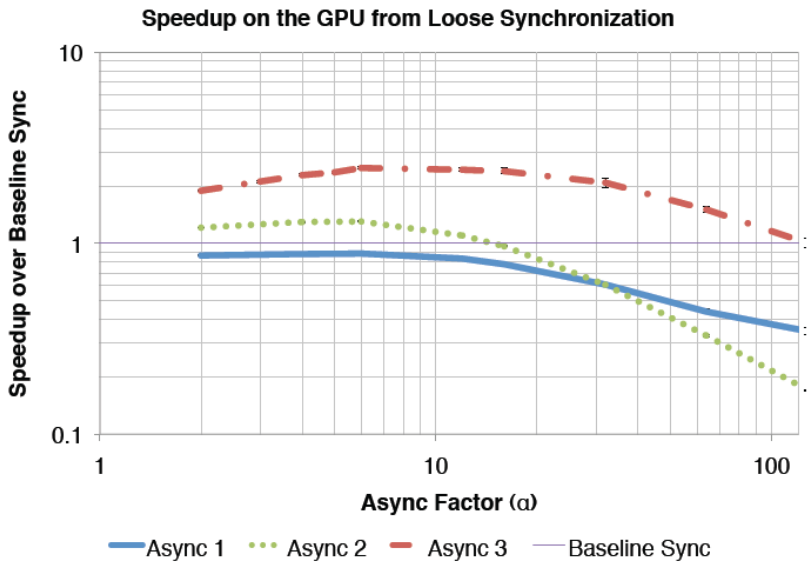


Figure 15: Measured multiple GPU performance, (*Left*) One NVIDIA Quadro FX 570 ("GPU 1") and One NVIDIA C1060 ("GPU 2"). Because of the large gap between the performance of the two GPUs ($\approx 18\times$ difference, not shown), the optimal fraction does not beat the GPU 2-only code. (Compare to Figure 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.

Results and Discussion

- Wildly asynchronous execution



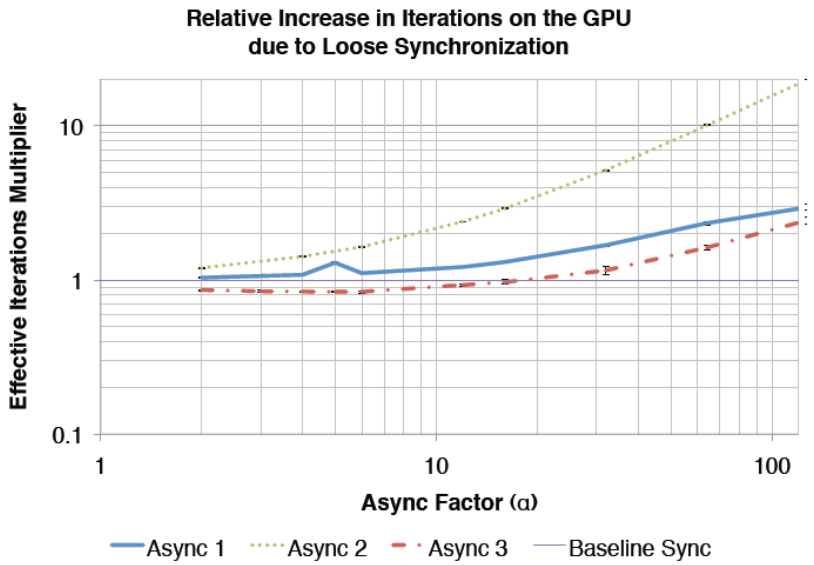


Figure 16: Comparison of asynchronous implementations on the NVIDIA C1060, for $n = 4096$ and $T = 1000$. (*Left*) Speedup relative to the synchronized baseline. (*Right*) Relative increase in effective iterations required to reach the same level of accuracy as the tuned synchronized GPU baseline (synchronized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

Conclusions and Future Work

- Hybrid performance models
  - Host-to-device transfer time, that have sometimes been omitted in prior work.
- Wildly Asynchronization
  - Now is an appropriate time to take a fresh look into this area of research.

My Comments

- Hybrid performance models
  - 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.
- It may be tricky to keep the accuracy
- Comparison for double-precision

Thank you!