Ed Karrels, edk@illinois.edu
CS 484, Project report

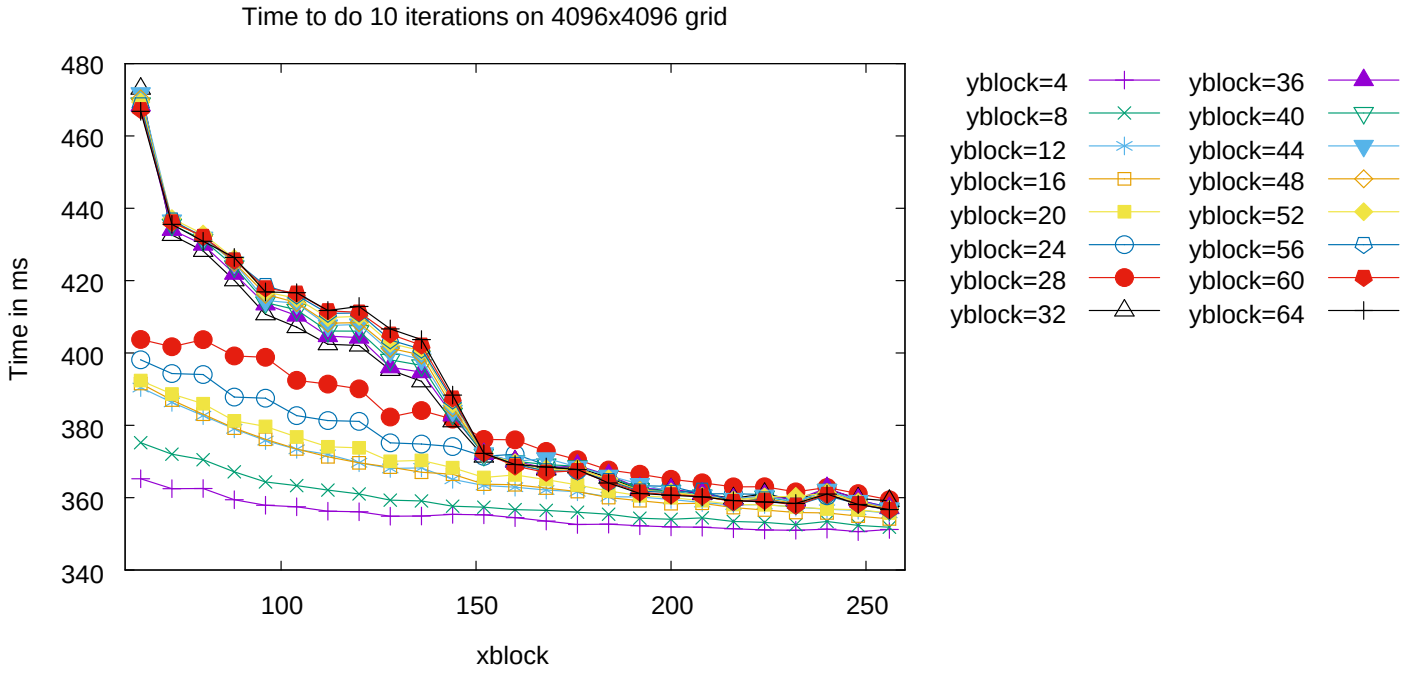
Jacobi Red-Black
CUDA vs Xeon vs Xeon Phi

One of our machine problems for this course was a multithreaded implementation of the Jacobi Red-Black algorithm. It was easily parallelized on the CPU and on a Xeon Phi coprocessor, but the performance on the Xeon Phi was disappointing. For my project I tried to improve the Xeon Phi performance, and I implemented the algorithm in CUDA. I wanted to see the difference in performance between the three solutions, and the difference in programmer productivity.

Baseline

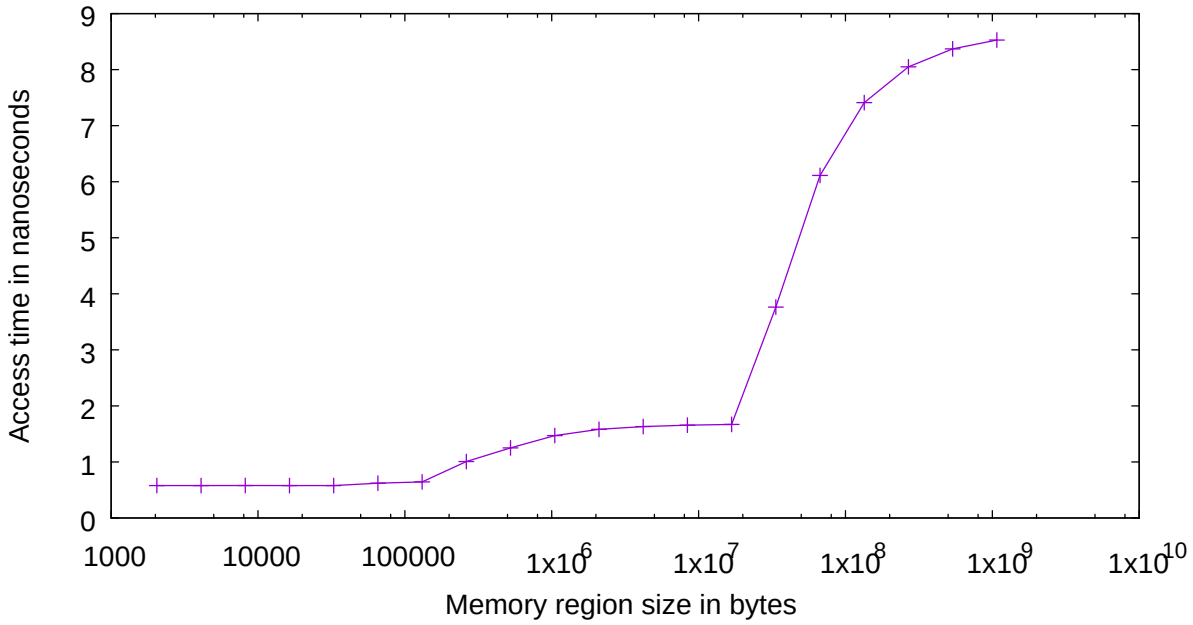
The algorithm is memory-bound. For each iteration of the algorithm, the computation for each cell reads 5 words of memory, writes 1, and does 5 floating point operations. Assuming the data is accessed in a cache-friendly manner, the computation should be limited by memory bandwidth.

I used the multithreaded cache-aware CPU version as a performance baseline. To maximize cache re-use, the two-dimensional grid of data is processed in tiles. I tested many different tile sizes to see what performed best. Here are the results.



The optimal tile size was of height 4 and maximum width. This make sense because with 4 byte, single-precision floats, each row of data in a 4096x4096 grid is just 16KB. Processing one row of tiles of height 4 will access 6 rows of data (since each cell is computed based on the adjacent cells), for a total of 96KB. This is larger than the L1 cache, but still fits in the L2 cache, which is not much slower.

To quantify memory latency, I wrote a program that accesses memory randomly from a region of memory. As the size of that region increases, the access pattern outgrows layers of the cache. The source for this is in "memspeed_random.cc", and it was run on the Taub cluster.

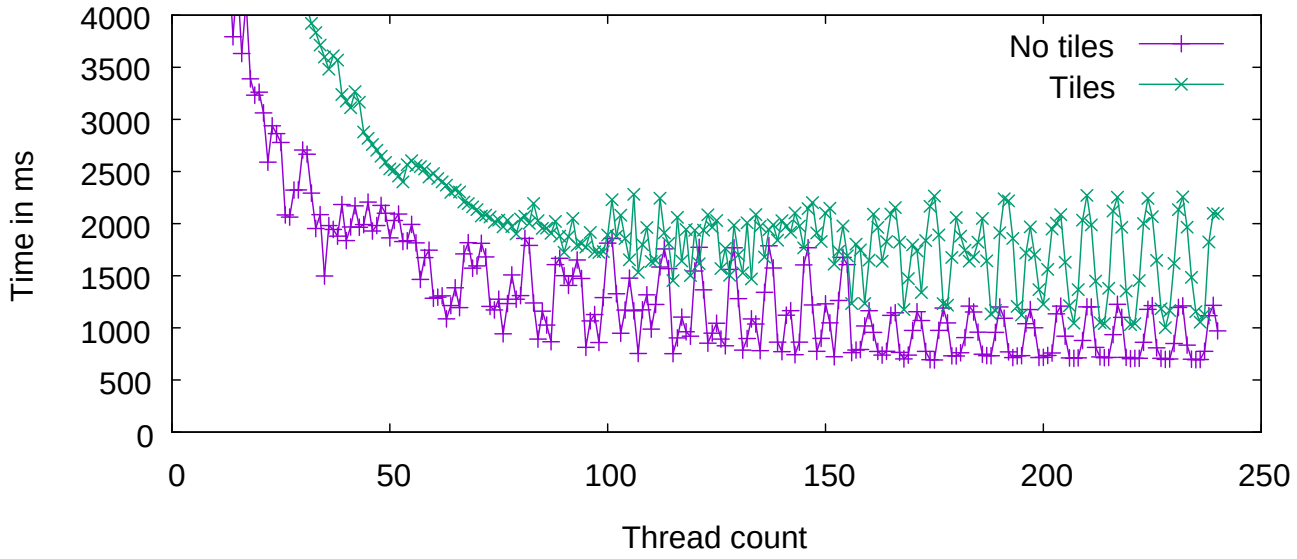


Using tiles for this problem doesn't really help much because the data access is already very localized. Even if no tiling is done, the memory accesses proceed linearly with three active read points: the current row, the row above, and the row below. Assuming the memory subsystem is intelligently prefetching for each of those read points, most of those memory accesses should be cache hits.

As part of the original machine problem, I did improve a performance over the simple linear approach by unrolling the row loop by four and the column loop by two, processing the red or black cells in a 4x4 mini-tile with each iteration.

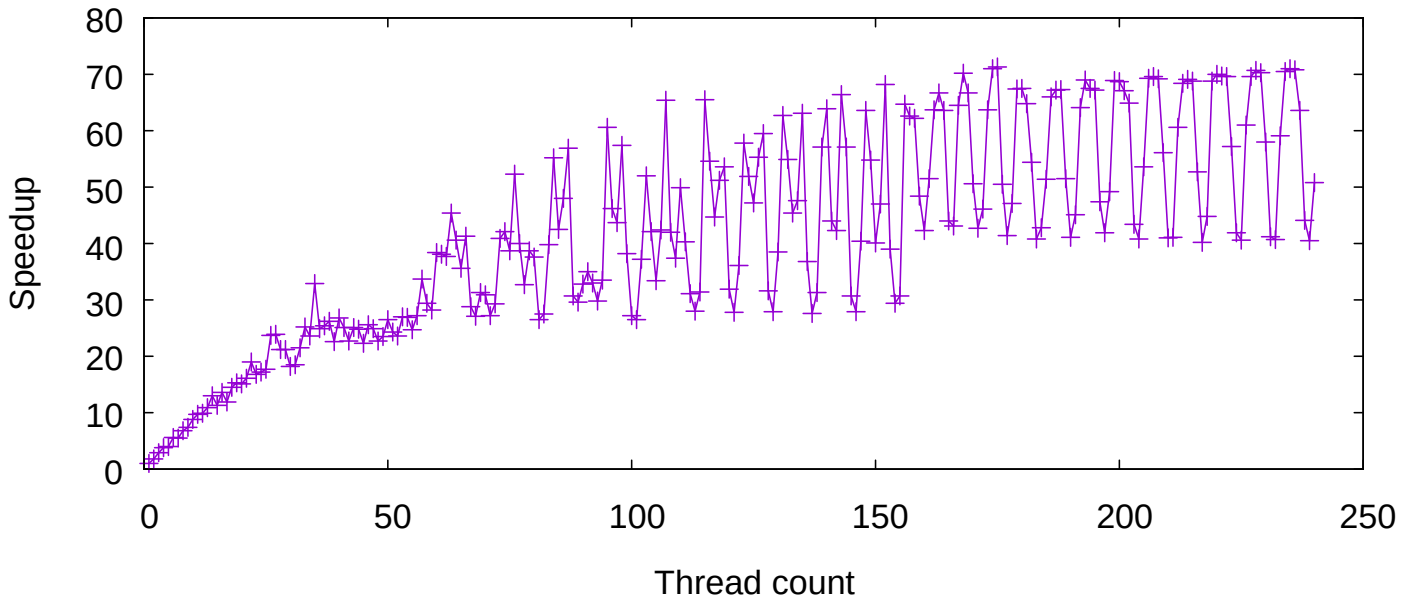
Xeon Phi

Using tiles didn't help on the Xeon Phi because it was unable to vectorize my unrolled mini-tile code.



I tried a number of transformations on my code, but none of them improved the performance on the Xeon Phi coprocessors. The only significant performance improvement I got was by tuning the thread count. With four

threads running each of the 60 cores, I expected 240 threads to get the best performance, but the pattern was more complicated than that.



The performance hit a plateau at about 160 threads, but it varied quite a bit, in a pattern with a cycle 7. The best performance was found with a thread count greater than 16 and congruent to 4 modulo 7. I was unable to figure out why that cycle of 7 in thread counts is so significant in the performance.

CUDA

I also implemented the algorithm on CUDA using a variety of approaches. I used an NVIDIA Titan Z in my testing.

Version 1: simple

For my baseline implementation, each thread computes the result for one cell, reading the adjacent cells directly from global memory. The problem maps naturally to a two-dimensional grid of thread blocks, and I got the best performance from thread blocks with 8 rows and 64 columns. This performed quite well, nearly twice as fast as the Xeon Phi version. The memory accesses are naturally coalesced, since adjacent threads read adjacent cells.

Version 2: multiple cells per thread

Rather than having each thread just compute one cell, I increased the tile size so each thread computes multiple cells. This had performance nearly identical to that of the simple version. I thought the performance would improve by reducing the number of threads and thread startup/shutdown overhead, but apparently that was not a significant factor.

Version 3: shared memory

Rather than using global memory all the data accesses, I tried reading each tile of data into shared memory. The new value for each cell is computed, and the result written to shared memory. When all threads have finished updating the cells, the tile is written back out to global memory. This access pattern guarantees all reads from and writes to global memory will be coalesced, and every memory access inside the computations will be to shared memory. It challenging to implement, mostly due to off-by-one errors introduced by the halo cells needed around each tile. The halo must be read on each iteration, but not written back to global memory. The performance of this version was about 25% slower than that of version 1. I suspect this is because the GPU I

was using already has an efficient cache, and the overhead of synchronizing all the threads to do the data copies in parallel was significant.

Version 4: read into shared memory, write to global

The last variation I tried was to read the tile of data into shared memory, but rather than writing out the tile after the computations completed, each computation directly writes its result to global memory. This produced a 2% speedup over the simple version. I suspect this performs better mostly because no synchronization is needed before writing the tile to memory. Given the difficulty of getting the tiled code working, in the future I would stick with the simple version.

Multiple GPUs

The Titan Z GPU card I was using contain two GPUs. I made a version of the code that splits the work across the two GPUs. The upper half of the grid is processed on the first GPU and the lower half on the second. By controlling the two GPUs using independent control streams, data transfers can be done to both cards at the same time. This reduced the time to copy the data from the CPU to the GPU from 27.4ms to 21.2ms. All the computations can be done independently on the two GPUs, but between each iteration the GPUs need to exchange a row of halo data on the boundary between them. This limits the speedup to a factor of 1.6, from 421ms to 258ms. Copying the data from the GPUs back to host memory was also faster than the single GPU version, 20.9ms, down from 22.0 ms.

Page-locked memory

I was able to achieve one last significant speedup in the CUDA implementation by using page-locked memory on the host. CUDA supports using DMA (direct memory access) to transfer data between host memory and device memory, bypassing the CPU. This only works if the memory is at a fixed physical address. Normal memory is fixed at a virtual address, but can be moved around or out of physical memory without informing the process. CUDA provides a function that allocates memory (on the host, not the device) that will not be moved or swapped out of physical memory. When copying data between that memory and the GPU, CUDA will use DMA to do the transfer, which supports a much higher throughput. Making this change improved the transfer rate from 6.5 GiB/sec to 10.2 GiB/sec.

Summary

Solution	Compute time in milliseconds	Speedup
Xeon CPU, single thread	8045	1
Xeon CPU, 12 threads	1083	4.5x
Xeon Phi coprocessor	692	11.6x
NVIDIA Titan Z, one GPU	421	19x
NVIDIA Titan Z, two GPUs	258	31x

Using the Xeon Phi coprocessor, I was able to get a nice speedup as compared to the Xeon CPU, but the performance was very sensitive to the number of threads, and the performance bottlenecks were not easy to find. Using CUDA-capable hardware with a similar acquisition cost I was able to get much better performance, but with much more time spent testing different strategies.