Parallelized Sorting Algorithms

Ben Dewan & Youlian Simidjiyski

December 9, 2010

1 Concept

Analyzing the performance of sorting algorithms is a fairly common task, especially in introductory computer science courses. At this point the performance characteristics of CPU-based, sequential sorting algorithms are well documented and studied. However, with the advent of GPGPU programming, we felt that, since many aspects of sorting are intuitively parallel, it would be interesting to examine the performance gains of implementing several well known sorting algorithms as GPGPU applications. We used NVIDIA's CUDA to implement four different algorithms (heap sort, merge sort, quick sort and radix sort) to measure possible performance gains over CPU-based implementations of the same algorithms, as well as learn the capabilities and limits of the GPGPU platform as it exists today.

2 Implementation

Our main goal was to compare the performance of the same sorting algorithm, implemented sequentially on the CPU and parallelized on the GPU. To utilize the GPU, our code was written using 'C for CUDA,' developed by NVIDIA, as well as framework code and helper functions written with C. We now explain the specifics of each implementation. Ultimately, we were unable to write any of the algorithms in full generality. We were unable to implement the metaheaps in our heapsort algorithm, and so heapsort can only sort up to 1023 elements. Radix and quicksort (likely) have a race condition which causes nondeterministic behaviour on lists of more than around 6,000 elements.

1. Heap Sort

   GPU-based implementations of heapsort are less common than Quick, Radix, or Merge sorting, as the heapsort does not naturally fit into the "reduction" paradigm (reduce the problem into many small partitions, then recursively combine the partitions). Rather than using the GPU's multiple processors for reduction and calculation on small problem sizes, our heapsort algorithm combines the processors into a pipeline. As far as

we are aware, this is a novel idea, though we did not perform a thorough literature review (a brief search yielded only [4] as a paper on parallel heapsort.)

Heapsort consists of two steps, a heapify step, and a pop heap step. The high-level concept behind our GPU heapsort is simple. To heapify, one gpu processor grabs the first list element, and begins heapifying it as in a sequential algorithm. Once the element has percolated up two levels in our heap, the next processor grabs the next element and begins percolating. Note that so long as the two processors move in lockstep, there can be no memory conflicts due to our waiting for first the processor to perform two percolations before allowing the second processor to begin. We continue to work in this manner, and so long as we have at least $\frac{\log(n)}{2}$ procesors (one for every other level in our heap), this pipeline can be run continuously. Heap pop is implemented using the same technique. By adding a pipeline to Heapsort, the runtime is reduced from O(nlogn) to O(n).

Unfortunately, due to small shared memory size and high latency of global memory, as well as the fact that not all processors actually move in lockstep, we cannot directly perform the above implementation. Dealing with these issues introduces significant additional complexity.

To deal with the small shared memory sizes, we borrow from the reduction paradigm, and split our heap into subheaps and a metaheap. Additional layers of metaheaps can, of course, be added, but we will only use a single metaheap for clarity of explanation.
The metaheap is a heap of subheaps. Two subheaps are compared by comparing their top elements. To pop, the metaHeap pops the top element of the top subheap, then percolates a new top element of the subheap, and finally percolates the subheap itself to its proper place in the metaheap based upon the new top element. Metaheap insert is performed in the obvious manner.

A key observation at this point is that the metaheap does not need information about the entire subheap to perform metaheap pop. If we have a buffer of the top few elements of the metaheap (technically, top two is all we need, but the more the better), we can use the buffer to perform the pop and completely ignore the rest of the subheap. Note that with this setup, it somewhat rare relatively rare for a metaheap buffer to be emptied completely (this would require popping the same subheap at least 4 times) before we have a chance to refill it (we will discuss refilling shortly). In addition, the metaheap and subheap percolation can still be completely pipelined.

With the above metaheap idea in hand, we implement our design as fol-

lows. A single block on a CUDA gpu will manage the metaheap. The remaining blocks will manage the subheaps. Within each block, (meta)heap management is handled by the first warp, which, according to the CUDA specification, is guaranteed to move in lockstep. Each thread in the first warp is responsible for percolating a single element at a time. Once the thread has finished, it is free to percolate another element. With the current CUDA architecture allowing 16kb of shared memory, we can safely have subheaps with 1023 floating point elements (depth 10), since our heapsort is not an in-place sort and so we require additional shared memory to actually generate our list.

The remaining warps in the subheap blocks assist in loads and stores, but otherwise perform no work. One potential optimization at this stage would be to use the additional warps to perform a binary search for the top four elements of the heap (this takes O(logn) time – much faster than heapify) and then fill the metaheap buffer while the first element performs a partial heapify.

The remaining warps in the metaheap block keep the metaheap buffer, which contains the top four elements of each subheap, filled. As a result of the metaheap buffer, we can only store a metaheap of size 511, yielding total sorting ability on around 500,000 elements.

2. Quick Sort
   Our CPU-based implementation of Quicksort is based upon the 'Quicksort with three-way partitioning' by Robert Sedgewick[1]. This particular implementation uses a a more aggressive partitioning scheme, splitting the list into three parts based upon the pivot value (less than, equal and greater than the pivot value), which can improve performance in situations involving duplicate key values, but otherwise follows a standard implementation of quicksort.

   Our GPU-based implementation of quicksort is not particularly sophisticated. With NVIDIA GPUs that have Compute Capability lower that 2.0, kernel functions (functions executed on the GPU) cannot be called recursively. Further complications arise when attempting to sort lists that are larger than will fit within a single thread block, because there can be no data dependencies between thread blocks, and partitioning and iterating on the GPU would require determining new pivots for the sublists, that is sharing data between thread blocks. We therefore compromised, exploiting the parallel nature of comparing values against the pivot and partitioning the list on the GPU, but making recursive function calls on the Host.

3. Radix Sort
   Our CPU-based implementation of Radix Sort is an implementation of 'Radix Quicksort,' again by Robert Sedgewick[1]. The algorithm works by performing bytewise comparisons and three-way partitioning of the list based on the value of the bit in relation to a pivot value, and then recursively calling itself on the sublists. As with quicksort, once the sublists become short enough, the overhead of performing all the comparisons and partitioning becomes a hindrance, so once the sublists being sorted decrease to lengths less than 10 elements, the list is sorted using insertion sort.

   The GPU-based implementation of this algorithm is very similar to that of Quicksort, however there is no pivot value, which streamlines the comparison and partitioning process (which in turn decreases the complexity of the kernel). Given a list to be sorted, we use the host to recursively call our bitwise comparison and partition kernel function.

For more indepth information, please view the source code, available at 'http://github.com/ysimidjiyski/CS222SearchSort/'

3 Testing and Results

One of the major issues with CUDA is that the execution of kernel functions requires a noticeable amount of overhead that is unnecessary for a standard program. Despite being rated CUDA Capable (with Compute Capability 1.1), when using an NVIDIA 8400 GTS we ran into the device memory wall at approximately 2000 element lists. With lists this short, the overhead of CUDA kernels (passing data to and from device to host) far exceeds the gains made through parallelizing the code (these penalties being exacerbated by the relativiely unoptimized nature of much of our code).
NOTE: Testing Quicksort and Heapsort algorithms was done using a NVIDIA 9800M GTS GPU and Intel Core 2 Duo (2.26 GHz). Radixsort was tested using an NVIDIA 8400 GTS and AMD Phenom II x4 (3.0GHz) (with which we encountered the device memory issues).
For lists of 6000 elements, CPU-Quicksort reported an average time of 0.05 seconds. GPU-Quicksort reported an average time of 2.3 seconds.
For lists of 2000 elements, CPU-Radixsort reported an average time of 0.01 seconds. GPU-Radixsort reported an average time of 5 seconds.
For a list of 1023 elements, GPU-Heapsort reported negligible time, but we were unable to test CPU-Heapsort or GPU-Heapsort on lists of longer length.

4 Conclusions

The major conclusion we have made is simply realizing the difficulty of implementing CUDA applications due to the 'newness' of the platform.
When dealing with a slower video card, CUDA applications don't just execute slower but the differences in scheduling can hide and/or reveal various race conditions, making it very difficult to debug. The required hands on control of memory and lack of error flags from CUDA functions (without using 'cudaSafeCall' from cutil.h in the SDK, which is recommended against because it is not thread safe) makes it even more challenging.
Add to this the lack of many core coding paradigms such as recursion and the lack of other capabilities such as atomic functions in shared memory and access to printf statements withing CUDA (both of which are resolved using cards with later compute capability) has only added to the difficulty.
As we have already seen with the new features available with the Fermi Architecture (Compute Capability 2.0), many of these features are being introduced to the CUDA environment, but without backwards compatability, it does not really help CUDA become a viable programming platform for hobbyists or developers who cannot acquire the latest iterations of NVIDIA video cards.

5 Next Steps

There are two major steps to take from this point. The first being further optimization, the second being analyzing sorting algorithms designed specifically for GPGPU processing.
Optimization, never a particularly easy process, is made more difficult due to the amount of control given to the developer over memory allocation and the lack of sophisticated programming models. Only with NVIDIA devices with Compute Capability 2.0 or higher (Fermi devices or later) support recursion, and the small stack and high cost of executing conditional and branch code on devices makes improving performance on a complicated kernel difficult. Also, whereas when one writes code to a general purpose processor, memory management is usually controlled by the operating system, hardware and/or the compiler, CUDA relies entirely on the developer to efficiently manage the caches to maximize performance. Whether this is due to the relative infancy of the platform (specifically the compiler) or an explicit design decision, it makes optimizing for CUDA much more involved, as cache optimizations must be explicit and there are no safeguards in place to prevent overflow (which can crash ones graphics drivers very easily).
Another question we should consider is, even after optimizing these algorithms, we should compare the results against some of the various algorithms designed specifically as GPGPU applications. The relatively parallel nature of sorting has led to the development of algorithms such as Bitonic-Merge Sort and Even-Odd Sorting Networks specifically for GPUs, and it would be interesting to see if they are quicker than a properly adapted and optimized verion of Quicksort

(which is, on average, the quickest sequential sorting algorithm). It may be that the idea of 'parallelizing' algorithms may be particularly inefficient, and therefore is worth testing.

6 References

[1] Robert Sedgewick, *Algorithms in C*, Third Edition.
[2] Daniel Cederman & Philippas Tsigas, "A Practical Quicksort ALgorithm for Graphics Processors," 2008. http://www.cse.chalmers.se/research/group/dcs/TechReports/gpusort.pdf
[3] Nadathur Satish, Mark Harris & Michael Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," 2009. http://mgarland.org/files/papers/gpusort-idps09.pdf
[4] Sepesi, Greg, "Dualheap Sort Algorithm: An Inherently Parallel Generalization of Heapsort," 2007 arxiv.org/pdf/0706.2893