On the Limits of GPU Acceleration

Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat (Efe) Guney
{richie,aparna,jee,efe}@gatech.edu

Abstract

This paper throws a small "wet blanket" on the hot topic of GPGPU acceleration, based on experience analyzing and tuning both multithreaded CPU and GPU implementations of three computations in scientific computing. These computations—(a) iterative sparse matrix linear solvers; (b) sparse Cholesky factorization; and (c) the fast multipole method—exhibit complex behavior and vary in computational intensity and memory reference irregularity. In each case, algorithmic analysis and prior work might lead us to conclude that an idealized GPU can deliver better performance, but we find that for at least equal-effort CPU tuning and consideration of realistic workloads and calling-contexts, we can with two modern quad-core CPU sockets roughly match one or two GPUs in performance.

Our conclusions should not dampen interest in GPU acceleration; on the contrary, they do quite the opposite: they partially illuminate the boundary between multicore CPU and GPU performance, and ask architects to consider larger application contexts in the design of future coupled on-die CPU/GPU processors.

1 Introduction and Scope

Our group has over the past year been engaged in the analysis, implementation, and tuning of a variety of irregular computations arising in computational science and engineering applications, for both multicore CPUs and GPGPU platforms [5, 12, 6, 16, 1]. In reflecting on this experience, the following question arose:

> What is the boundary between computations that can and cannot be effectively accelerated by GPUs, relative to general-purpose multicore CPUs within a roughly comparable power footprint?

Though we do not claim to have definitively answered this question, we believe that our preliminary findings might surprise the broader community of application development teams whose charge it is to decide whether and how much effort to expend on GPGPU code development.

Position. Our central aim is to provoke a more critical view of the role of GPGPU accelerators in applications. In particular, we argue that, for a moderately complex class of "irregular" computations, even well-tuned GPGPU accelerated implementations on currently available systems will deliver only comparable performance, when compared to *well-tuned* general-purpose multicore CPU systems within a roughly comparable power footprint. Put another way, adding a GPU is equivalent in performance to simply adding one or perhaps two more multicore CPU sockets. Thus, one might reasonably ask whether this level of performance increase is worth the potential "productivity loss" due to having to adopt a new programming model and re-tune for the accelerator.

Our computations of interest are (a) iterative solvers for sparse linear systems; (b) direct solvers for sparse linear systems; and (c) the fast multipole method for particle systems. These computations appear in traditional high-performance scientific computing applications, but are also of increasing importance in graphics, physics-based games, and large-scale machine learning problems.

Threats to validity. Our conclusions represent our interpretation of the data, and indeed our position is intended to provoke discussion. By way of "full disclosure" upfront, we acknowledge at least the following three major weaknesses in our position.

- (Threat 1) *Our perspective comes from a relatively narrow classes of applications.* These computations come from traditional HPC applications.
- (Threat 2) *Some conclusions are drawn from partial results.* Our work is very much on-going, and we are carefully studying our GPU codes to ensure that we have not missed additional tuning opportunities.
- (Threat 3) *Our results are limited to today's platforms.* We consider current widely available

and state-of-the-art NVIDIA offerings (Tesla C1060/S1070 and GTX285 systems), but have not evaluated the latest ATI offerings. Moreover, we recognize that NVIDIA's upcoming Fermi processor could drastically change the conclusions as well [13, 14]. Also, some of the performance limits we have encountered stem in part from the limits of PCIe. If CPUs and GPUs move onto the same die, this limitation may become irrelevant.

Having acknowledged these limitations, we make the following counter-arguments.

Regarding Threat 1, we claim this class has at least two interesting features. First, as stated previously, we believe our target computations will have an impact in increasingly sophisticated emerging applications in graphics, gaming, and machine learning. Secondy, the computations are non-trivial, going beyond just a single "kernel," like matrix multiply or sparse matrix-vector multiplication. Since they involve additional context, the computations begin to approach larger and more realistic applications. Thirdly, they have a mix of regular and irregular behavior, and may therefore live near the "boundaries" of what we might expect to run well on a GPU vs. a CPU.

Regarding Threat 2, we would claim that we achieve extremely high levels of absolute performance in all our codes, so it is not clear whether there is much room left for additional improvement, without resorting to entirely new algorithms.

Regarding Threat 3, it seems to us that just moving a GPU-like accelerator unit on the same die as one or more CPU-like cores will not solve all problems. For example, the high-bandwidth channels available on a GPU board would presumably have to be translated to a future same-die CPU/GPU socket as well, in order to deliver the same level of performance we enjoy today when the entire problem can reside on the GPU.

2 Iterative Sparse Solvers

We first consider the class of iterative sparse solvers. Given a sparse matrix A, we wish either to solve a linear system (i.e., compute the solution x of $Ax = b$) or compute the eigenvalues and/or eigenvectors of A, using an *iterative* method, such as the conjugate gradients or Lanczos algorithms [7]. These algorithms have the same basic structure: they iteratively compute a sequence approximate solutions that ultimately converge to the solution within a user-specified error tolerance. Each iteration consists of multiplying the sparse A by a dense vector, which is called a *sparse matrix-vector multiply (SpMV)* operation. Algorithmically, an SpMV computes $y \leftarrow A \cdot x$, given A and x. To first order, an SpMV is dominated simply by the time to stream the matrix A, and within an iteration, SpMV has no temporal locality. That is, we expect SpMV, and thus the solver overall, to be largely memory-bandwidth bound.

We have for many years studied autotuning of SpMV for single- and multicore CPU platforms [16, 15, 11]. The challenge is that although SpMV is bandwidth bound, a sparse matrix must be stored using a graph data structure, which will lead to indirect and irregular memory references to the x and/or y vectors, depending on the specific data structure used to store A. Nevertheless, the main cost for typical applications on *cache-based* machines is the bandwidth-bound aspect of reading A.

Thus, GPUs are attractive for SpMV because they deliver much higher raw memory bandwidth than a multisocket CPU system within a (very) roughly equal power budget. We have extended our autotuning methodologies for CPU-tuning [15] to the case of GPUs [6]. We do in fact achieve a considerable $2\times$ speedup over the CPU case, as Figure 1 shows for a variety of finite-element modeling problems (x-axis) in double-precision: our autotuned GPU SpMV on a single NVIDIA GTX285 system achieves 12–19 Gflop/s, compared to an autotuned dual-socket quad-core Intel Nehalem implementation that achieves 7–8 Gflop/s, with 1.5–2.3$\times$ improvements. This improvement is roughly what we might expect, given that the GTX285's peak bandwidth is 159 GB/s compared to the aggregate peak bandwidth on the dual-socket Nehalem system of 51 GB/s, a $3.1\times$ difference.

However, this performance assumes the matrix is already on the GPU. In fact, there will be additional costs for moving the matrix to the GPU combined with GPU-specific *data reorganization*. That is, the optimal implementation on the GPU uses a different data structure than either of the the optimal or baseline implementations on the CPU. Indeed, this data structure tuning is even more critical on the GPU, due to the performance requirement of coalesced accesses; without it, the GPU provides no advantage over the CPU [3].

The host-to-GPU copy is also not negligible. To see why, consider the following. Recall that, to first order, SpMV streams the matrix A, and performs just 2 flops per matrix entry. If SpMV runs at P Gflop/s in double-precision, then the "equivalent"

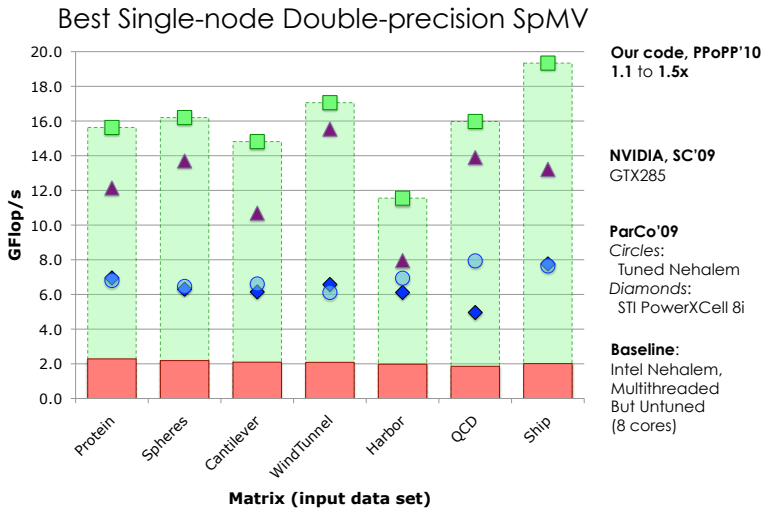

Figure 1: The best GPU implementation of sparse matrix-vector multiply (SpMV) ("Our code", on one NVIDIA GTX 285) can be over 2× faster than a highly-tuned multicore CPU implementation ("Tuned Nehalem", on a dual-socket quad-core system). Implementations: ParCo'09 [16], SC'09 [3], and PPoPP'10 [6]. Note: Figure also to appear elsewhere [2].

effective bandwidth in double-precision is at least (8 bytes) / (2 flops) * P, or $4P$ GB/s. Now, decompose the GPU solver execution time into three phases: (a) data reorganization, at a rate of β_{reorg} words per second second; (b) host-to-GPU data transfer, at β_{transfer} words per second, without increasing the size of A; and finally (c) q iterations of SpMV, at an effective rate of β_{gpu} words per second. On a multicore CPU, let β_{cpu} be the equivalent effective bandwidth, also in words per second. For a matrix of k words, we will only observe a speedup if the CPU time, τ_{cpu}, exceeds the GPU time, τ_{gpu}. With this constraint, we can determine how many iterations q are necessary for the GPU-based solver to beat the CPU-based one:

$$\tau_{\text{cpu}} \geq \tau_{\text{gpu}} \quad (1)$$

$$\Rightarrow \frac{k \cdot q}{\beta_{\text{cpu}}} \geq k \cdot \left(\frac{1}{\beta_{\text{reorg}}} + \frac{1}{\beta_{\text{transfer}}} + \frac{q}{\beta_{\text{gpu}}} \right) \quad (2)$$

$$\Rightarrow q \geq \frac{\frac{1}{\beta_{\text{reorg}}} + \frac{1}{\beta_{\text{transfer}}}}{\frac{1}{\beta_{\text{cpu}}} - \frac{1}{\beta_{\text{gpu}}}} \quad (3)$$

From Figure 1, we might optimistically take β_{gpu}= (4 bytes/flop) * (19 Gflop/s) = 76 GB/s, and pessimistically take β_{cpu}= (4 bytes per flop) * 6 Gflop/s = 24 GB/s; both are about half the aggregate peak on the respective platforms. Reasonable estimates of β_{reorg} and β_{transfer}, based on measurement (not peak), are 0.5 and 1 GB/s, respectively. The solver must, therefore, perform $q \approx 105$ iterations to breakeven; thus, to realize an actual 2× speedup on the whole solve, we would need $q \approx 840$ iterations. While typical iteration counts reported for standard problems number in the few hundreds [7], whether this value of q is "large" or not is *highly* problem- and solver-dependent, and we might not know until run-time when the problem (matrix) is known. The developer must make an educated guess and take a chance, raising the question of what she or he should expect the real pay-off from GPU acceleration to be.

Having said that, our analysis may also be pessimistic. One could, for instance, improve effective β_{transfer} term by pipelining the matrix transfer with the SpMV. Or, one might be able to eliminate the β_{transfer} term altogether by assembling the matrix on the GPU itself [4]. The main point is that making use of GPU acceleration even in this relatively simple "application" is more complicated than it might at first seem.

3 Direct Sparse Solvers

The iterative solvers described above have a simple structure, but robustness (rate of convergence to a desired accuracy) is always an issue. A more robust alternative approach is to directly compute the solution using an explicit matrix factorization, rather than iterate toward a solution. In the direct approach, one is guaranteed a certain number of operations, at the cost of significantly more complex task-level parallelism, more storage, and irregular memory reference behavior, when compared to the

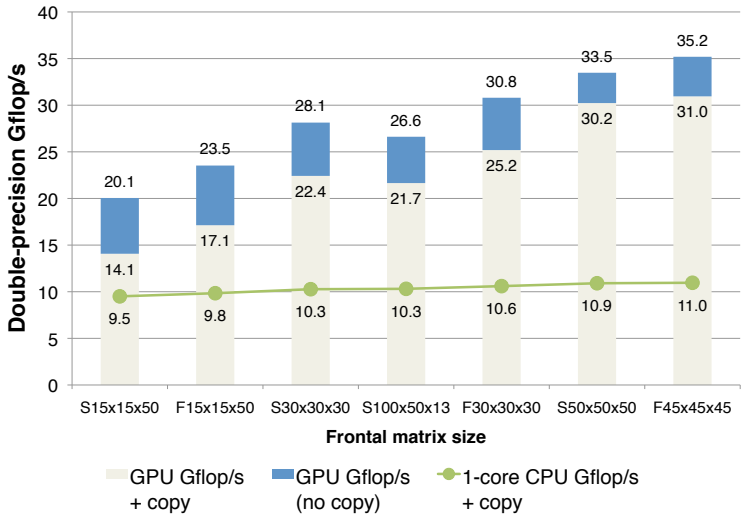

Figure 2: Single-core CPU vs. GPU implementations of sparse Cholesky factorization, both including and excluding host-to-GPU data transfer time. Though the GPU provides a speedup of up to $3\times$, this is compared to just a *single* CPU core of an 8-core system. Note: Figure also to appear elsewhere [10].

largely data-parallel and streaming behavior of the iterative case (Section 2).

We have been interested in such sparse direct solvers, particularly so-called multifrontal methods for Cholesky factorization, which we tune specifically for structural analysis problems arising in civil engineering [10]. The most relevant aspect of a sparse direct solver from the perspective of GPU acceleration is that the workload consists of many dense matrix subproblems (factorization, triangular multiple-vector solves, and rank-k update matrix multiplications). Generally speaking, we expect a GPU to easily accelerate such subcomputations.

In reality, however, the size of these subproblems changes as the computation proceeds, and the subproblems themselves may execute asynchronously together, depending on the input problem. That is, the input matrix determines the distribution of subproblem sizes, and moreover dictates how much cross-subproblem task-level parallelism exists. Thus, though the subproblem "kernels" map well to GPUs in principle, in practice the structure demands CPU-driven coordination, and the cost of moving data from host to GPU will be critical.

Figure 2 makes this point explicitly. We show the performance (double-precision Gflop/s) of a preliminary implementation of partial sparse Cholesky factorization, on benchmark problems arising in structural analysis problems. Going from left-to-right, the problems roughly increase in problem size. The different implementations are (a) a well-tuned, single-core CPU implementation, running on a dual-socket quad-core Nehalem system with dense linear algebra support from Intel's Math Kernel Library (MKL); and (b) a GPU implementation, running on the same Nehalem system but with the cores just for coordination and the GPU acceleration via an NVIDIA Tesla C1060 with CUBLAS for dense linear algebra support. Furthermore, we distinguish two GPU cases: one in which we ignore the cost of copies (red bar), and one in which we include the cost of copies (blue bar). The GPU speedup over the single CPU core is just $3\times$, meaning a reasonable multithreaded parallelization across all 8 Nehalem cores is likely to win.

4 Generalized N-body Solvers

The third computation we consider is the fast multipole method (FMM), a hierarchical tree-based approximation algorithm for computing all-pairs of forces in a particle system [8, 18, 17]. Our interest derives from the fact that not only can physics problems be solved by the FMM, but large classes of methods in statistical data analysis and mining, such as nearest neighbor search or kernel density estimation (and other so-called *kernel methods*), also have FMM-like algorithms. Thus, a good FMM implementation accelerated by a GPU will have broad applicability in multiple domains.

In short, the FMM approach reduces an exact $O(N^2)$ algorithm for N interacting particles into an approximate $O(N)$ or $O(N \log N)$ algorithm with an error guarantee. The FMM is based on two key

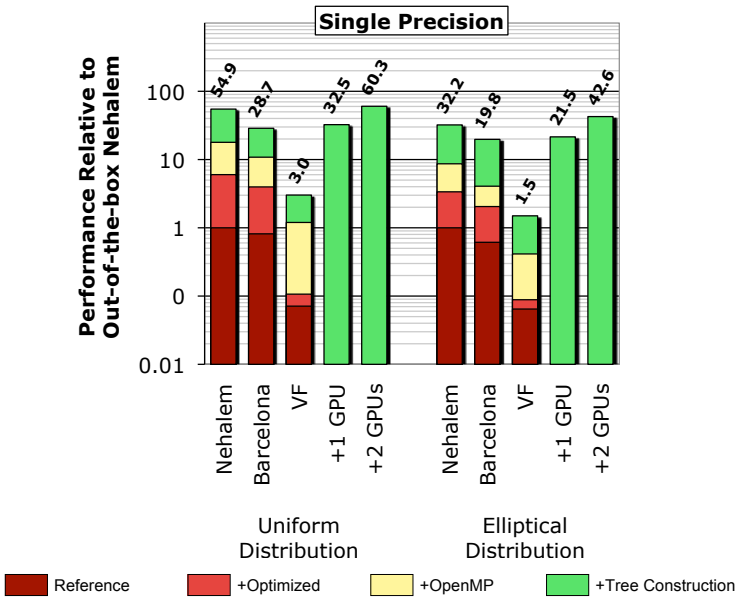

Figure 3: Cross-platform comparison of the fast multipole method. All performance is shown relative to an "out-of-the-box" 1-core Nehalem implementation; each bar is labeled by this speedup. VF = Sun's Victoria Falls multithreaded processor. Note: Figure also to appear elsewhere [5].

ideas: (a) a *tree representation* for organizing the points spatially; and (b) *fast approximate evaluation*, in which we compute summaries at each node using a constant number of tree traversals with constant work per node. The dominant cost is the evaluation phase, which is not simple: it consists of 6 distinct components, each with its own computational intensity and varying memory reference irregularity.

All components essentially amount either to tree traversal or graph-based neighborhood traversals. Like the case of sparse direct solvers, the computation within each component is regular and there is abundant parallelism. However, the cost of each component varies depending on the particle distribution, shape of the tree, and desired accuracy. Thus, the optimal tuning has a strong run-time dependence, and mapping the data structures and subcomputations to the GPU is not straightforward.

Figure 3 summarizes the results of our current cross-platform comparison, which includes both CPU and one- and two-GPU implementations. Prior work by others had suggested we should expect significant speedups (30–60×) from GPU acceleration compared to a single CPU core [9]. As Figure 3 shows, our own GPU implementation did in fact yield this range of speedups compared to a baseline code on a single Nehalem core [12]. However, we also found that explicit parallelization and tuning of the multicore CPU implementation could yield an implementation on Nehalem that nearly *matched* the *dual*-GPU code, within about 10%. Like both of the previous computation classes, the same issues arise: (a) there is overhead from necessary GPU-specific data structure reorganization and host-to-GPU copies; and (b) variable workloads, which results in abundant but irregular parallelism as well as sufficiently irregular memory access patterns.

5 Concluding Remarks

In short, the intent of this paper is to consider much of the recent work on GPU acceleration and ask for CPU comparisons in more realistic application contexts. Such comparisons are critical for applications like the ones we consider here, which lie between completely regular computations (e.g., dense matrix multiply) and wildly irregular applications (tree-, linked list-, and graph-intensive computations). Our observations suggest that, for our computations, adding a GPU to a conventional system is like adding roughly an additional one or two sockets of performance. This raises broader questions about the boundary between when a GPU "wins" over a CPU, and whether any productivity loss (if any) of tuning specifically for a GPU is outweighed by the performance gained.

References

[1] N. Arora, A. Shringarpure, and R. Vuduc. Direct n-body kernels for multicore platforms. In *Proc. Int'l. Conf. Parallel Processing (ICPP)*, Vienna, Austria, September 2009. [71/220=32.3%].

[2] N. Bell, J. Choi, M. Garland, L. Oliker, R. W. Vuduc, and S. Williams. Sparse matrix vector multiplication on multicore and accelerator systems. In J. Dongarra, D. A. Bader, and J. Kurzak, editors, *Scientific Computing with Multicore Processors and Accelerators*. CRC Press, 2010.

[3] N. Bell and M. Garland. Implementing a sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Portland, OR, USA, November 2009. (*to appear*).

[4] C. Cecka, A. J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *Int'l. J. Numerical Methods in Engineering*, 2009.

[5] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Atlanta, GA, USA, April 2010. [127/527=24.1%] (*accepted*).

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, Bangalore, India, January 2010. [29/173=16.8%] (*accepted*).

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

[8] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, 73:325–348, 1987.

[9] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *J. Comp. Phys.*, 227:8290–8313, 2008.

[10] M. E. Guney. *Sparse direct solvers for structural engineering problems on multicore systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, May 2010. (*expected*).

[11] E.-J. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *Int'l J. of High Performance Computing Applications (IJHPCA)*, 18(1):135–158, February 2004.

[12] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Portland, OR, USA, November 2009. [59/261=22.6%]
Finalist, Best Paper.

[13] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi™, v1.1. Whitepaper (electronic), September 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[14] D. A. Patterson. The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. `http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf`, September 2009.

[15] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Phys.: Conf. Series*, volume 16, pages 521–530, 2005.

[16] S. Williams, R. Vuduc, L. Oliker, J. Shalf, K. Yelick, and J. Demmel. Optimizing sparse matrix-vector multiply on emerging multicore platforms. *Parallel Computing (ParCo)*, 35(3):178–194, March 2009. Extends conference version: `http://dx.doi.org/10.1145/1362622.1362674`. *Most downloaded paper*, Q1 2009.

[17] L. Ying, G. Biros, D. Zorin, and H. Langston. A new parallel kernel-independent fast multipole method. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Phoenix, AZ, USA, November 2003.

[18] L. Ying, D. Zorin, and G. Biros. A kernel-independent adaptive fast multipole method in two and three dimensions. *J. Comp. Phys.*, 196:591–626, May 2004.