Gaussian Elimination Based Algorithms on the GPU

Aydın Buluç[a,*,1], John R. Gilbert[a,1], Ceren Budak[a]

[a] *Computer Science Department, University of California, Santa Barbara, CA 93106-5110*

Abstract

We implemented and evaluated several Gaussian elimination based algorithms on Graphic Processing Units (GPUs). These algorithms, LU decomposition without pivoting, all-pairs shortest-paths, and transitive closure, all have similar data access patterns. The impressive computational power and memory bandwidth of the GPU make it an attractive platform to run such computationally intensive algorithms. Although improvements over CPU implementations have previously been achieved for those algorithms in terms of raw speed, the utilization of the underlying computational resources was quite low. We implemented a recursively partioned all-pairs shortest-paths algorithm that harnesses the power of GPUs better than existing implementations. The alternate schedule of path computations allowed us to cast almost all operations into matrix-matrix multiplications on a semiring. Since matrix-matrix multiplication is highly optimized and has a high ratio of computation to communication, our implementation does not suffer from the premature saturation of bandwidth resources as iterative algorithms do. By increasing temporal locality, our implementation runs more than two orders of magnitude faster on an NVIDIA 8800 GPU than on an Opteron. Our work provides evidence that programmers should rethink algorithms instead of directly porting them to GPU.

Key words: All-pairs shortest paths, Gaussian elimination, graphical processing units, semirings, matrix multiplication, graph algorithms

1. Introduction

The massively parallel nature of GPUs makes them capable of yielding theoretically much higher GFlops rates than current state-of-the-art CPUs. GPU performance also grows much faster than CPU performance due to specialized explicit parallelism. The amount of computational power to be harvested has

*Corresponding author

Email addresses: aydin@cs.ucsb.edu (Aydın Buluç), gilbert@cs.ucsb.edu (John R. Gilbert), cbudak@cs.ucsb.edu (Ceren Budak)

[1] The research of these authors was supported in part by the Department of Energy under award number DE-FG02-04ER25632, in part by NSF grant CNS-0709385, and in part by MIT Lincoln Laboratory under contract number 7000012980

also attracted the high-performance computing (HPC) community, and we have seen many scientific applications successfully implemented with significant performance gains on the GPU [5, 30].

Implementing HPC applications to run on a GPU requires significant expertise, even with the recently introduced C-like APIs such as Nvidia's Cuda platform [25]. The key to performance is to hide the data access latency by having many threads on the fly. The performance is usually fragile and requires careful craftmanship from the programmer's side. It is up to the programmer to make sure that the registers and other levels of cache are neither underutilized nor over-pressured. Several papers are devoted to the issue of achieving the right balance to get optimal performance on GPUs [28, 37], relying on novel programming techniques that are not necessarily intuitive to the existing HPC programmer.

Gaussian elimination (GE) based algorithms have triple nested loops and very similar data access patterns. Examples include LU decomposition without pivoting, Cholesky factorization, all-pairs shortest paths (APSP), and transitive closure. The similarity among those problems has led researchers to approach them in a unified manner. For example, the Gaussian Elimination Paradigm of Chowdhury and Ramachandran provides a cache-oblivious framework for these problems [8]. In this paper, we specifically focus on the APSP problem because it usually operates on single precision floating point data, making it suitable to current generation GPUs. On the contrary, factorizations such as LU and Cholesky require double precision arithmetic that was not available on the GPUs until very recently (with AMD FireStream 9170 and Nvidia GeForce GTX 280). Even now, the double precision performance is 4-8 times slower than single precision, and the limited global memory of current generation GPUs discourage the use of double precision floating point numbers. Furthermore, numerical LU decomposition without pivoting is unstable [17] at best (it may not even exist), and pivoting strategies on the GPU are beyond the scope of this paper. Volkov and Demmel did an excellent job of implementing LU, QR, and Cholesky factorizations on the GPU, albeit in single precision [37]. It is worth noting that our implementations compute only the distance version of the APSP problem, i.e. they do not explicitly compute the paths, only their lengths. This is due to memory limitations of the GPUs.

Our two main contributions in this paper are:

1. Recursive partitioning is used as a tool to express a different schedule of path computations that allows extensive use of highly optimized matrix-matrix operations. Specifically, we use matrix multiplication on semirings as a building block for GE based algorithms. By doing so, we increase data locality, which is even more important for high performance computing on the GPU than on the CPU
2. As a proof of concept, we provide an efficient implementation of the APSP algorithm on the GPU that is up to 480x faster than our reference CPU implementation, and up to 75x faster than an existing GPU implementation on a similar architecture.

Locality of reference has always been an issue in algorithm design, and it will be even more important with GPUs. Our work provides evidence that some fundamental algorithmic techniques that have not been popular in the HPC community, such as recursion, may become mainstream. This is because stream processors, such as GPUs, achieve efficiency through locality [11], and recursive algorithms naturally exploit locality.

As minor contributions, we give an alternate (arguably simpler) proof of correctness based on path expressions for the recursively partitioned APSP algorithm. On the GPU, we compare iterative, and recursive versions of the same algorithm and provide insights into their performance difference through micro benchmarks. Therefore, we provide evidence that BLAS-3 routines on semirings can be used to speed up certain graph algorithms. Finally, we compare different CPUs and GPUs on their power efficiency in solving this problem.

2. Gaussian Elimination Based Algorithms

Gaussian elimination is used to solve a system of linear equations $Ax = b$, where A is an $n \times n$ matrix of coefficients, x is a vector of unknowns, and b is a vector of constants. Recursive blocked LU factorization is an efficient way of performing Gaussian elimination on architectures with deep memory hierarchies [12, 35]. This is mostly due to its extensive use of matrix-matrix operations (Level 3 BLAS) that are optimized for the underlying architecture. Let A be partitioned as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}. \tag{1}$$

Then, the in-place version of the recursive blocked algorithm without pivoting can be written as

$$\begin{aligned} A_{11} &\leftarrow \texttt{LU}(A_{11}) \\ A_{12} &\leftarrow A_{12} \backslash A_{11} \\ A_{21} &\leftarrow A_{21} / A_{11} \\ A_{22} &\leftarrow \texttt{LU}(A_{22} - A_{21} A_{12}). \end{aligned} \tag{2}$$

In this pseudocode, `LU` is the recursive call to the function itself, \ and / denote triangular solve operations with multiple right hand sides.

LU factorization operates on the field of real numbers, but the same algorithm can be used to solve a number of graph problems, albeit using a different algebra. Specifically, closed semirings provide a general algebraic structure that can be used to solve a number of path problems on graphs [4, 33]. A semiring has all the properties of a ring, except that there might be elements without an additive inverse. One practical implication is that fast matrix multiplication algorithms that use additive inverses, such as the Strassen algorithm [32] and the Coppersmith-Winograd algorithm [9], do not apply to matrices over semirings.

A closed semiring is formally denoted by $(\mathbb{S}, \oplus, \otimes, 0, 1)$, where $\oplus$ and $\otimes$ are binary operations defined on the set $\mathbb{S}$ with identity elements 0 and 1 respectively. Fletcher [15] gives a complete definition of a closed semiring. Two important semirings used in this work are the *Boolean semiring* $(\{0,1\}, \vee, \wedge, 0, 1)$ and the *tropical semiring* $(\mathbb{R}^+, \min, +, \infty, 0)$. A closed semiring is said to be *idempotent* if $a \oplus a = a$ for all $a \in \mathbb{S}$. Although idempotence of the semiring is not a requirement for the solution of path problems on graphs [15], the correctness of our in-place algorithms relies on idempotence. Both the Boolean semiring and the tropical semiring are idempotent, as $\min(a, a) = a$ for all $a \in \mathbb{R}^+$, and $0 \vee 0 = 0$, $1 \vee 1 = 1$.

2.1. The All-Pairs Shortest-Paths Problem

The all-pairs shortest-paths (APSP) is a fundamental graph problem. Given a directed graph $G = (V, E)$ with vertices $V = \{v_1, v_2, ..., v_n\}$ and edges $E = \{e_1, e_2, ..., e_m\}$, the problem is to compute the length of the shortest path from v_i to v_j for all (v_i, v_j) pairs. APSP corresponds to finding the matrix closure $A^* = I \oplus A \oplus A^2 \oplus ...$ on the tropical semiring.

APSP is the focus of this paper among the set of GE based algorithms due to its practical importance and the lack of fast implementations on the GPU. All the algorithms discussed in this paper take the adjacency matrix A of the graph, where $A(i, j)$ represents the length of the edge $v_i \rightarrow v_j$, as the input. They output A^*, where $A^*(i, j)$ represents the length of the shortest path from v_i to v_j. Edge weights can be arbitrary (positive, negative, or zero), but we assume that there are no negative cycles in the graph. Also, the cost of staying at the same vertex is zero, i.e., $A(i, i) = 0$. If not, we can delete any edge of the form $A(i, i) \neq 0$ as it will certainly not contribute to any shortest path. This is because shortest paths are simple when there are no negative cycles.

The standard algorithm for solving the APSP problem is the Floyd-Warshall (FW) algorithm. It is especially well-suited for dense graphs due to its $\Theta(n^3)$ complexity. It is a dynamic programming algorithm that consists of a triply nested loop similar to matrix multiplication. In fact, computing the APSP problem is computationally equivalent to computing the product of two matrices on a semiring [4]. However, the order of the loops cannot be changed arbitrarily as in the case of matrix multiplication. In the linear algebra sense, the algorithm computes the outer product of the k^{th} row and the k^{th} column, and does rank-1 updates on the whole matrix, for $k = 1, 2, ..., n$. The order of the outer product updates cannot be changed, but one is free to compute the outer product in any order. This means that the k-loop should be the outermost loop, and the other loops can be freely interchanged. Although the added constraint on the order of loops hinders some of the loop-interchange optimizations that are applied to matrix multiplication, automatic program generators for the FW algorithm have been shown to provide formidable speedups [19]. The pseudocode for the FW algorithm, in standard notation and in linear algebra notation, are given in Figures 1 and 2.

For sparse graphs, Johnson's algorithm [24], which runs Dijkstra's single-source shortest paths algorithm from every vertex (after some preprocessing

```
A* : R^{N×N} = FW(A : R^{N×N})
1  for k ← 0 to N − 1
2      do for i ← 0 to N − 1
3          do for j ← 0 to N − 1
4              do A(i, j) ← min(A(i, j), A(i, k) + A(k, j))
5  A* ← A
```

Figure 1: FW algorithm in the stardard notation

```
A* : R^{N×N} = FW(A : R^{N×N})
1  for k ← 0 to N − 1
2      do A ← A ⊕ A(:, k) ⊗ A(k, :)      ▷ Algebra on the (min,+) semiring
3  A* ← A
```

Figure 2: FW algorithm in linear algebra notation

that lets the algorithm run on graphs having edges with negative weights), is probably the algorithm of choice for an implementation on the CPU. However, as we demonstrate in Section 4, the GE based algorithm clearly outperforms both the FW algorithm and Johnson's algorithm when implemented on the GPU.

For unweighted graphs, it is possible to embed the semiring into the ring of integers and use a fast, $o(n^3)$, matrix multiplication. For an undirected and unweighted graph, Seidel [29] gives a $O(M(n) \lg n)$ algorithm, where $M(n)$ is the time to multiply two $n \times n$ matrices on the ring of integers. This elegant algorithm repeatedly squares the adjacency matrix of the graph. However, it is not currently known how to generalize Seidel's algorithm to weighted or directed graphs [38].

2.2. Recursive In-Place APSP Algorithm

The closure of a matrix can be computed using an algorithm similar to recursive Gaussian elimination without pivoting. It is guaranteed to terminate on a closed semiring like the tropical semiring. The only subroutine of this algorithm is matrix multiplication on a semiring. The n-by-n adjacency matrix is recursively partitioned into four equal-sized $n/2$-by-$n/2$ submatrices as before; the pseudocode for the algorithm is shown in Figure 3. We use juxtaposition (AB) to denote the multiplication of A and B on the semiring. The algorithm does not require n to be even. If n is odd, the same decomposition in (1) works with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.

$$
\begin{aligned}
A_{11} &\leftarrow A_{11}^* \\
A_{12} &\leftarrow A_{11} A_{12} \\
A_{21} &\leftarrow A_{21} A_{11} \\
A_{22} &\leftarrow A_{22} \oplus A_{21} A_{12} \\
A_{22} &\leftarrow A_{22}^* \\
A_{21} &\leftarrow A_{22} A_{21} \\
A_{12} &\leftarrow A_{12} A_{22} \\
A_{11} &\leftarrow A_{11} \oplus A_{12} A_{21}.
\end{aligned}
\tag{3}
$$

Figure 3: Pseudocode for recursive in-place APSP

Recursive formulations of APSP have been presented by many researchers over the years [10, 27, 34]. The connection to semiring matrix multiplication was shown by Aho et al. [4], but they did not present a complete algorithm. Ours is a modified version of the algorithm of Tiskin [34] and R-Kleene algorithm [10]. Especially, the in-place nature of the R-Kleene algorithm helped us avoid expensive global memory to global memory data copying. As the algorithm makes use of matrix multiplication as a subroutine, it has a much higher data reuse ratio while having asymptotically the same operation count.

The correctness of the recursive algorithm has been formally proven in various ways before [10, 27]. Here we present an simpler proof based on algebraic paths. As in Aho et al. [4], we partition the set of vertices into $V_1 = \{v_1, ..., v_{n/2}\}$ and $V_2 = \{v_{n/2+1}, ..., v_n\}$. Submatrix A_{11} represents the edges within V_1, submatrix A_{12} the edges from V_1 to V_2, submatrix A_{21} the edges from V_2 to V_1, and submatrix A_{22} the edges within V_2.

Now, consider the paths in A_{11}^*. They can either travel within V_1 only or move from V_1 to V_2 following an edge in A_{12}, and then come back to V_2 through an edge in A_{21}, possibly after traveling within V_2 for a while by following edges in A_{22}. The regular expression for the latter path is $A_{12}A_{22}^*A_{21}$. This partial path can be repeated a number of times, possibly going through different vertices each time. An example path from v to w is shown in Figure 4. The complete regular expression becomes

$$
A_{11}^* = (A_{11} \mid A_{12}A_{22}^*A_{21})^*. \tag{4}
$$

On the other hand, the regular expression we get after the recursive algorithm terminates is

$$
A_{11}^* = A_{11}^* \mid (A_{11}^* A_{12}(A_{22} \mid A_{21}A_{11}^*A_{12})^* A_{21}A_{11}^*). \tag{5}
$$

These two regular expressions define the same language, hence represent the same set of paths [33]. By converting these regular expressions into deterministic

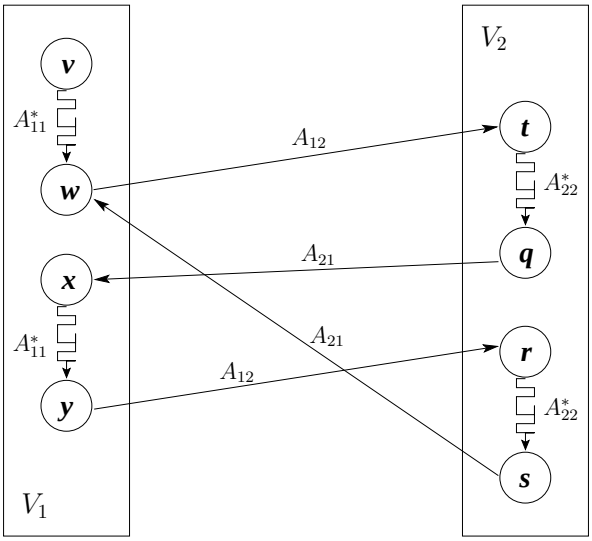

Figure 4: An example path in A_{11}^*

finite automata (DFA), and minimizing them [22], we see that both have the same minimum-state DFA shown in Figure 5. Since the minimum-state DFA is unique for a language, this proves that the algorithm computes the correct set of paths.

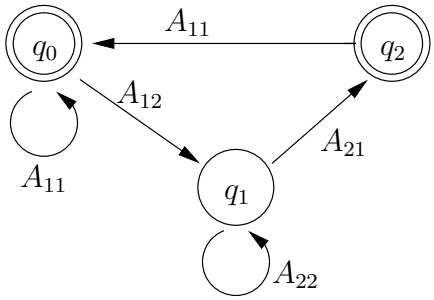

Figure 5: Minimum-state DFA for the path expressions in A_{11}^*

It is also possible to implement this algorithm in a blocked iterative way as previously done for transitive closure [36]. The percentage of work done iteratively (without using matrix multiplication) is the same, and corresponds to the block diagonal part of the matrix. However, the multiplications in the blocked algorithm are always between matrices of size $B \times B$, where B is the blocking factor. This is potentially a limiting factor on GPUs because multiplication

tends to get drastically faster as matrices get bigger (less than 20 GFlops/s when N=64 versus 200 GFlops/s when N=1024) [37]. With the recursive formulation, on the other hand, more work can be done during multiplication of large matrices.

Furthermore, the recursive algorithm does fewer kernel launches than the block iterative one. The block iterative algorithm launches $O((N/B)^3)$ kernels for matrix multiplications and $O(N/B)$ kernels for computing closures of $B \times B$ blocks on the diagonal. On the other hand, at each level of the recursion tree, the recursive algorithm launches 6 kernels for matrix multiplications, and does 2 recursive calls. This makes a total of only $O(N/B)$ kernel launches because the height of the recursion tree is $\lg(N/B)$, and the number of kernel launches doubles at each level ($\{6, 12, 24, ..., 6(N/B)\}$). The $O((N/B)^2)$ factor of improvement can be quite significant, as kernel launches incur significant overhead in CUDA.

One important feature of our implementation is that it is performed in place, overwriting the input with the output without constraining the order of loops in the matrix multiplication. For the matrix multiply-add operations $A_{22} \leftarrow A_{22} \oplus A_{21}A_{12}$ and $A_{11} \leftarrow A_{11} \oplus A_{12}A_{21}$, there are no issues of correctness. However, for other multiplications of the form $B \leftarrow BA$ or $B \leftarrow AB$, the order of evaluation (whether it is an ijk loop or an kji loop) matters on a general semiring. This is because updating the output automatically updates the input, and the algorithm will now use a different input for the rest of the computation. As proved by D'Alberto and Nicolau [10], this is not a problem as long as the semiring is idempotent and A is a closure. The intuition is that if the algorithm prematurely overwrites its input, this just makes the algorithm find shortest paths quicker. In other words, it speeds up the information dissemination, but the correctness is preserved thanks to idempotence.

Note that four of the six multiplications at any level of the recursion tree are of the form $B \leftarrow BA$ or $B \leftarrow AB$. In other words, they perform multiply instead of multiply-add operations. Using $B \leftarrow B + BA$ or $B \leftarrow B + AB$ would be equally correct, but unnecessary. Remember that the cost of staying in a vertex is zero, i.e. $A(i, i) = 0$. Consider $B \leftarrow AB$: If B contains a path $v_i \Rightarrow v_j$ before the operation, AB generates a cost-equivalent path $v_i \Rightarrow v_i \Rightarrow v_j$ and safely overwrites B.

3. GPU Computing Model with CUDA

More and more applications that traditionally run on the CPU are now being reimplemented to run on the GPU, a technique called general-purpose computing on graphics processing units (GPGPU). Both Nvidia and AMD offer programming interfaces for making GPGPU accessible to programmers who are not experts in computer graphics [1, 3]. Nvidia's Compute Unified Device Architecture (Cuda) offers a higher level C-like API, whereas AMD's Close-to-Metal (CTM) allows the programmers to access lower levels of hardware. As opposed to CTM, the Cuda platform is unified in the sense that it has no architectural division for vertex and pixel processing.

3.1. GPU Programming

The new generation of GPUs are basically multithreaded stream processors. They offer tremendous amounts of bandwidth and single-precision floating point arithmetic computation rates. In stream processing, a single data parallel function (kernel) is executed on a stream of data, and that is exactly how the Cuda programming model works. A Cuda program is composed of two parts: A host (CPU) code that makes kernel calls, and a device (GPU) code that actually implements the kernel. The host code is conceptually a serial C program, but the device code should be massively parallel in order to harness the power of the GPU.

The fundamental building block of Nvidia 8 and 9 series is the streaming multiprocessors (SMs), sometimes called the GPU chips. Each SM consists of 8 streaming processors (cores), but only one instruction fetch/decode unit. The implies that all 8 cores must simultaneously execute the same instruction. This is why divergence in the device code should be avoided as much as possible. The memory hierarchy consists of multiple levels. Each SM has 8192 registers and 16KB on-chip shared memory, which is as fast as registers provided that bank conflicts are avoided. A high-latency (200-300 cycles) off-chip global memory provides the main storage for the application on the GPU. Part of the off-chip memory, called the local memory, is used for storing variables that are spilled from registers.

A kernel is executed by many threads on the GPU. These threads are organized as a grid of thread blocks, which are batches of threads that can cooperate/communicate through on-chip shared memory and synchronize their execution. Each thread block is executed by only one SM, but each SM can execute multiple thread blocks simultaneously.

The main scheduling unit in Cuda is a *warp*, a group of 32 threads from the same thread block. All threads in a warp execute the same instruction, and execution of an arithmetic instruction for the whole warp takes 4 clock cycles. The number of active warps in a block is an important factor in tolerating global memory access latency.

3.2. Experiences and Observations

Some limitations exist for the device code. For example, recursion and static variables are not allowed. These limitations do not apply to the host code, as it is just a regular C code running on the CPU. In fact, recursion in the host code is a powerful technique, since it naturally separates the recursion stack from the floating-point intensive part of the program. Although recursive divide-and-conquer algorithms are naturally cache efficient [18], they have traditionally not achieved their full performance due to the overheads associated with recursion. We do not have such a limitation with CUDA because the recursion stack, which is on the CPU, does not interfere with the kernel code on the GPU.

Code optimization on a GPU is a tedious job with many pitfalls. Performance on a GPU is often more fragile than performance on a CPU. It has been observed that small changes can cause huge effects on the performance [28]. For

example, in the optimized GEMM routine of Volkov [37], each thread block is 16×4 and each thread uses 32 registers. This allows $8192/32 = 256$ threads and $256/64 = 4$ thread blocks can simultaneously be active on each SM. As there are two warps per thread block and it takes 4 cycles to execute an instruction for the whole warp, a latency of $8 \times 4 = 32$ cycles can be completely hidden. In the case that an extra variable is required, the compiler can either choose to spill it out to local memory and keep the register count intact, or increase the register usage per thread by one. In the latter case, the number of active thread blocks decreases to 3. This introduces a 25% reduction in parallelism, but the former option may perform worse if the kernel has few instructions because access to a local variable will introduce one-time extra latency of 200-300 cycles. Whichever option is chosen, it is obvious that performance is fragile: by just adding one extra line, it is possible to drastically slow down the computation.

Another pitfall awaiting the programmer is bandwidth optimizations. In Cuda, peak bandwidth can only be achieved through memory coalescing, i.e. by making consecutively numbered threads access consecutive memory locations. One can heavily underutilize the GPU bandwidth by not paying attention to memory coalescing. However, the way memory coalescing works is quite counter-intuitive to a multicore programmer. Assume that one wants to scan a $16 \times N$ matrix stored in row-major order. On an SMP system with 16 cores, the most bandwidth-friendly way is to let each processor scan a different row of the matrix; in this case, each processor makes at most N/B cache misses, which is optimal. On an Nvidia GPU, on the other hand, this will create multiple memory accesses per warp since these threads do not access contiguous range of memory addresses. An example with $N = 8$ is shown in Figure 6. However, if the matrix were stored in column-major order, having each thread scan a different row would be optimal on an Nvidia GPU. This is because memory accesses at each step would be coalesced into a single access by the NVCC compiler [26]. Consequently, the right programming practices for achieving high bandwidth are quite different for the GPU than for traditional parallel programming.

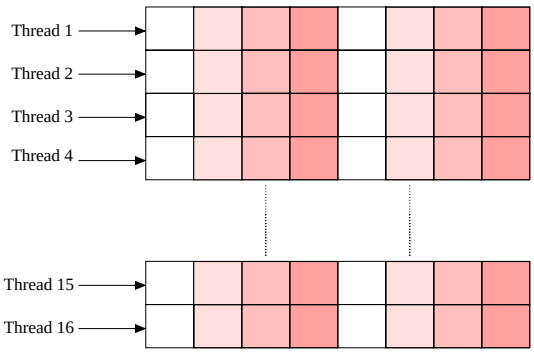

Figure 6: Stride-1 access per thread (row-major storage)

As a result, we advocate the use of optimized primitives as much as possible on the GPU. Harris et al. provide an excellent optimized scan primitive with Cuda and encourage its use as a building block for implementing parallel algorithms on Nvidia GPUs [21]. Here, we advocate the use of matrix-matrix multiplication as an important primitive, not only for solving systems of linear equations, but also for graph computations. In terms of performance, matrix multiplication has been claimed to be unsuitable to run on GPUs due to the lack of sufficient bandwidth [13]. The new generation GPUs, however, offer a tremendous bandwidth of more than 100 GB/s. Moreover, alternate implementations that are not bandwidth bound achieved close to peak performance [37]. It would be wise to take advantage of such an efficient primitive whenever possible.

4. Implementation and Experimentation

4.1. Experimental Platforms

We ran our GPU code on an Nvidia GeForce 8800 Ultra with Cuda SDK 1.1 and GCC version 4.1. The graphics card driver installed in our system is Nvidia Unix x86_64 kernel module 169.09. The GeForce 8800 Ultra has 768 MB DRAM, a core clock of 612 MHz, a stream processor clock of 1.5 GHz, a memory clock of 1080 MHz, and an impressive bandwidth of 103.7 GB/s. It consists of 16 SMs, each containing 8 cores, making up a total of 128 cores. Each core can perform a multiply-add operation in a single cycle, which accounts for two floating-point operations (Flops). Therefore, it offers a peak multiply-add rate of $2 \times 1.5 \times 128 = 384$ GFlops/s (not counting the extra MUL operation that cores can issue only under certain circumstances).

For comparison, we ran our CPU experiments in three different settings:

1. Serial C++ code on Intel Core 2 Duo T2400 1.83 Ghz with 1 GB RAM running Windows XP.
2. Serial C++ code on Opteron 2.2 Ghz with 64 GB RAM running Linux kernel 2.6.18
3. Parallel Cilk++ code on a Numa machine (Neumann) with 64 GB RAM, and 8 dual-core Opteron processors clocked at 2.2 Ghz.

4.2. Implementation Details

We implemented both the recursive and the iterative algorithm on the GPU using Cuda. For the recursive algorithm, we experimented with two different versions: one that uses a simple GEMM kernel, and one that uses the optimized GEMM routine of Volkov [37]. When reporting experimental results, we call the latter *recursive optimized*. Both recursive codes implement the same algorithm given in Figure 3. Our recursive Cuda code is freely available at `http://gauss.cs.ucsb.edu/~aydin/apsp_cuda.html`.

Our iterative APSP implementation uses a logical 2D partitioning of the whole adjacency matrix. Such a decomposition was previously employed by

Jenq and Sahni on a hypercube multiprocessor [23], and found to be more effective than 1D partitioning. However, keep in mind that there is no explicit data partitioning, only a logical mapping of submatrices to thread blocks. Host code invokes the kernel n times, where each thread block does a rank-1 update to its submatrix per invocation. An initial snapshot of the execution is illustrated in Figure 7 from the viewpoint of $(2, 2)$ thread block.

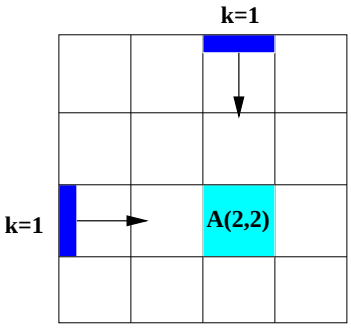

Figure 7: A shapshot from the execution of the iterative algorithm

Our serial iterative and recursive implementations run on the CPU as references. The iterative implementation is the standard implementation of FW, as shown in Figure 1. The recursive implementation is based on our recursive formulation shown in Figure 3. The recursive implemention stops the recursion when the submatrices completely fit into L1-cache to achieve better results.

Our reference parallel implementation runs on Neumann, a Numa machine with a total of 16 processor cores (8 dual-core 2.2 Ghz Opterons). We used Cilk++ [2] to parallelize our code, which enabled speedups up to 15x.

4.3. Performance Results

Timings for our APSP implementations on Cuda are given in Table 1. Please note the orders of magnitude difference among implementations.

Table 1: GPU timings on GeForce 8800 Ultra (in milliseconds)

Num. of Vertices	Iterative	Recursive	Recursive Optimized
512	2.51×10^2	1.62×10^1	6.43×10^0
1024	2.42×10^3	1.00×10^2	2.44×10^1
2048	4.60×10^4	7.46×10^2	1.41×10^2
4096	4.13×10^5	5.88×10^3	1.01×10^3
8192	5.47×10^6	5.57×10^4	7.87×10^3

Among our reference implementations, the best CPU performance is obtained on the Intel Core 2 Duo, even though the processor had a slower clock

Table 2: Speedup on 8800 Ultra w.r.t. the best CPU implementation

Num. of Vertices	Iterative	Recursive	Recursive Optimized
512	3.1	48.1	121.4
1024	3.0	73.4	301.5
2048	1.3	79.6	420.7
4096	1.2	81.5	473.2
8192	0.7	67.7	479.3

speed than the Opteron. We attribute this difference to the superior performance of MS Visual Studio's C++ compiler. Full listings of timings obtained on two different CPUs and various compilers can be found in Appendix A. Table 2 shows the speedup of various GPU implementations with respect to the best CPU performance achieved for the given number of vertices. The results are impressive, showing up to 480x speedups over our reference CPU implementation. Using an iterative formulation, only a modest 3.1x speedup is achieved for relatively small inputs.

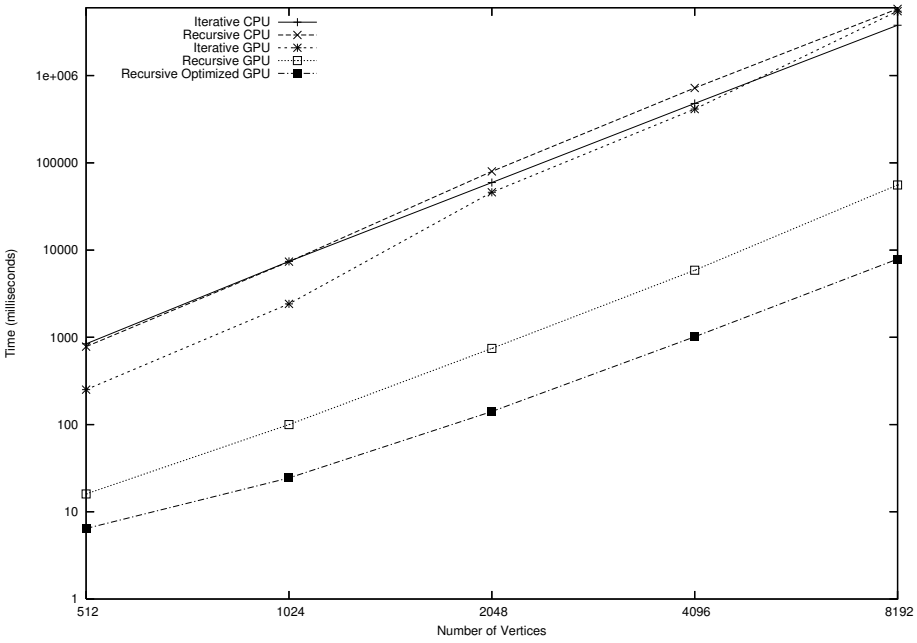

Figure 8: Log-log plot of absolute running times

Figure 8 shows a log-log plot of running times of 5 different implementations. Iterative CPU and recursive CPU are timings obtained by our serial code

Table 3: Observed exponents and constants for the asymptotic behaviour of our APSP implementations with increasing problem size

$t = c\|V\|^n$	CPU (Intel Core 2 Duo)		GPU (GeForce 8800 Ultra)		
	Iterative	Recursive	Iterative	Recursive	Recur. Optimized
Exponent (n)	3.02	3.23	3.62	2.94	2.59
Constant (c)	5.5×10^{-6}	1.4×10^{-6}	3.6×10^{-8}	1.5×10^{-7}	4.7×10^{-7}

Table 4: Performance comparison of our best (optimized recursive) GPU implementation with parallel Cilk++ code running on Neumann, using all 16 cores

Num. of Vertices	Best GPU (secs)	Parallel CPU (secs)	GPU Speedup
512	0.00643	0.113	17.5×
1024	0.0244	0.708	29×
2048	0.141	5.146	36.5×
4096	1.01	40.36	40×
8192	7.87	354.9	45×

running on Intel Core 2 Duo. For the rest of this section, we will be referring to the recursive optimized code as our best GPU code.

The observed exponent of the recursive GPU implementation turned out to be slightly different than theoretical values. To reveal that, we performed a least-squares polynomial data fit on the log-log data. The input size($|V|$) - running time(t) relationship is of the form $t = c|V|^n$. This can be converted to $\lg t = \lg c + n \lg |V|$, on which we can do linear data fitting. The observed exponents and constants are reported in Table 3.

Our best GPU implementation still outperforms the parallelized CPU code by a factor of 17-45x, even on 16 processors. Timings are listed in Table 4. The economic advantage of the GPU is clear. Each of the dual-core sockets cost more than $500, with a total cost of more than $4000, where Nvidia 8800 costs less than $600. These estimates ignore the costs of other supporting hardware.

4.4. Comparison with Earlier Performance Results

We compare the performance of our code with two previously reported results. One is an automatically generated, highly optimized serial program running on a 3.6 Ghz Pentium 4 CPU [19]. The other is due to Harish and Narayanan on a GPU platform very similar to ours [20]. Our GeForce 8800 Ultra is slightly faster than the GeForce 8800 GTX used by Harish and Narayanan, so we underclocked our GPU to allow a direct comparison in terms of absolute values.

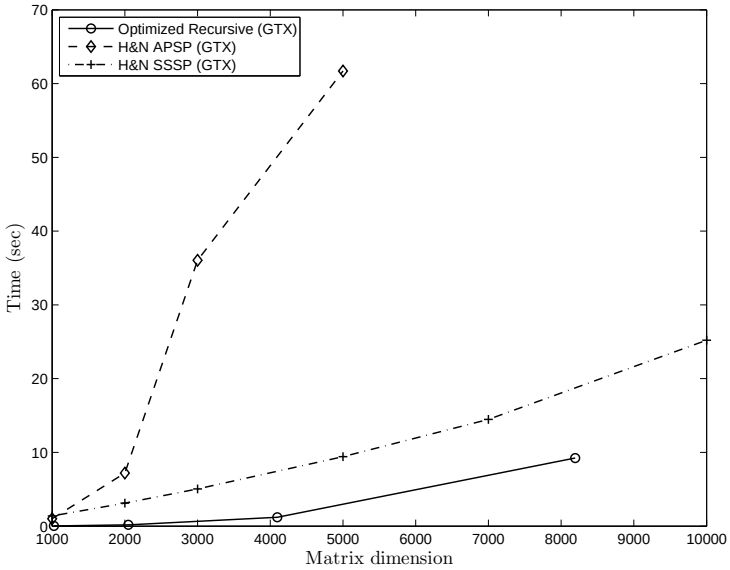

Figure 9: Comparison of different GPU implementations on 8800 GTX settings

On the GPU, Harish and Narayanan implemented two variants of APSP: one that uses the FW algorithm and one that runs Dijkstra's single source shortest paths (SSSP) algorithm for every vertex. For sparse graphs with $m = O(n)$, the latter is theoretically faster than both the FW algorithm and our recursive formulation in the classical RAM model of computation [4]. It runs in $O(n^2 \lg n + nm)$ time using Fibonacci heaps [16]. However, as seen in Figure 9, our recursive implementation significantly outperforms both their FW implementation (H&N APSP) and Dijkstra based implementation (H&N SSSP) when implemented on a GPU. The running times for the H&N SSSP code are observed for randomly generated Erdős-Rényi graphs with an average vertex degree of 6. The running times of the other two implementations are not sensitive to sparsity. When timing our algorithm, we underclocked our GPU's clocks down to the speed of 8800 GTX for a head-to-head comparison. Due to the adjacency matrix representation, our algorithm runs on graphs of at most 8192 vertices. Therefore, the H&N SSSP implementation is currently more favorable for large sparse graphs, although it lags behind in terms of raw speed. We plan to implement an out-of-core version of our algorithm for larger graphs.

The performance results for our iterative algorithm, given in Section 4.3, agree with the 2x-3x speedup over a CPU implementation achieved by H&N APSP. That implementation was also limited to 4096 vertices, while ours extends to 8192 with only a slowdown. Our best APSP code is faster than H&N APSP by a factor of 35-75x.

Comparing our results with the timings reported by Han et al. for the optimized code obtained using their auto generation tool Spiral [19], we also see

significant speedups achieved by our best (optimized recursive) GPU implementation. Our comparisons are against their vectorized code (typically 4-5x faster than scalar code), and we see speedups up to 28x against Pentium 4, and 42x against Athlon 64. A detailed comparison can be found in Table 5. Those results also show that the GPU implementation scales better with increasing problem size, because the speedup we get over Spiral increases as the problem size increases.

Table 5: Comparisons of our best GPU implementation with the timings reported for Han et al. 's auto generation tool Spiral

Num. of Vertices	GFlops/s			Speedup of GeForce	
	GeForce 8800	Pentium 4	Athlon 64	Pentium 4	Athlon 64
512	38.6	5.08	3.17	7.6x	12.2x
1024	82.0	5.00	2.77	16.4x	29.6x
2048	113.5	4.78	2.73	23.7x	41.6x
4096	126.7	4.47	2.96	28.3x	42.8x

4.5. Scalability and Resource Usage

In this section, we try to identify the bottlenecks in our implementation in terms of resource usage and scalability. By using the NVIDIA Coolbits utility, we tweaked the frequencies of both the GPU core clock and the memory clock. The results reveal that our recursive implementation is not limited by the memory bandwidth to global GPU DRAM. For this implementation, the timings and GFlops/s rates with different clock rates are given in Table 6. When the memory clock is fixed, the slowdown of the computation closely tracks the slowdown of the GPU core clock (0-50% with increments of 12.5%). On the other hand, when the GPU core clock is fixed, little slowdown is observed when we underclock the memory clock. Coolbits reported the default clock speeds of 8800 Ultra as 648 Mhz for cores, and 1152 Mhz for memory, which are slightly different than the values reported in NVIDIA factsheets.

The peak rate observed was 130 GFlops/s for $|V| = 8192$, compared to the theoretical peak of 384 GFlops. However, the theoretical peak counts 2 Flops for each fused multiply-add operation, which is not available on the tropical semiring our algorithm operates on. Therefore, the actual theoretical peak in the absence of fused multiply-add operations is 192 GFlops. Our implementation achieves more than 67% of that arithmetic peak rate for APSP.

The iterative implementation, on the other hand, is observed to be completely bandwidth bound. Even when the GPU cores are underclocked to half, no slowdown was observed. Underclocking the memory to half, however, slowed down the computation by exactly a factor of two. Exact timings can be seen in Figure 7. We conclude that the iterative formulation is putting too much stress on GPU memory bandwidth, consequently not harnessing the available computation power of the GPU. This is indeed expected, because the iterative

Table 6: Scalability of our optimized recursive GPU implementation. We tweaked core and memory clock rates using Coolbits.

$\|V\| = 4096$	GPU Clock	Memory Clock	Time (ms)	GFlops/s	Slowdown (%)
Default values	648	1152	1028.3	124.4	-
	567	1152	1190.8	107.5	13.6
Memory	486	1152	1362.9	93.9	24.5
clock fixed	405	1152	1673.1	76.5	38.5
at 1152 Mhz	324	1152	2093.7	61.1	50.8
	648	1008	1036.2	123.5	0.7
GPU core	648	864	1047.3	122.2	1.8
clock fixed	648	720	1096	116.8	6.1
at 648 Mhz	648	576	1124.9	113.8	8.5

Table 7: Scalability of our iterative GPU implementation. We tweaked core and memory clock rates using Coolbits.

$\|V\| = 4096$	GPU Clock	Memory Clock	Time (ms)	Slowdown (%)
Default values	648	1152	417611.4	-
Core clock halved	324	1152	418845.7	0.3
Memory clock halved	648	576	856689.7	51.2

formulation accesses $O(n^2)$ data and does $O(n^2)$ work in every iteration. The recursive algorithm, on the other hand, does almost all of its work in matrix multiplications, which access $O(n^2)$ data for doing $O(n^3)$ work. Therefore, it clearly has better locality of reference.

As it was not possible to disable a subset of GPU cores in the NVIDIA 8800, we do not report any scalability results with increasing number of processors.

4.6. Power Consumption

Power efficiency is becoming an important consideration when comparing different architectures [14]. The Green500 list ranks supercomputers according to their Flops/Watts/s (or Flops/Joule) ratio. In this section, we compare the power efficiency of different architures for the APSP problem. Due to the lack of equipment we were not able to measure exact power usages during the computation. Thus, we will refer to these values provided by the manufacturers. The results, however, are orders of magnitude different, so that the uncertainty in the actual power consumption when running the application is acceptable.

Nvidia reports a peak power consumption of 175 Watts for its GeForce 8800 Ultra video card. Our dual-core Opteron (model number 8214) is reported to consume a peak power of 95 Watts, but we are using only a single core of it during serial computation. The machines used in the reported timings of automatically tuned CPU implementations are Pentium 4 (model number 560)

Table 8: Efficiency comparison of different architectures (running various codes), values in MFlops/Watts/s (or equivalently MFlops/Joule)

$\|V\|$	Nvidia GPU	Athlon	Pentium 4	Core 2 Duo	Neumann (Opteron)
	Best Cuda code	Spiral Code		Reference FW	Cilk++ (p=16)
512	173	35.6	44.1	19.1	2.9
1024	368	31.1	43.7	17.4	3.7
2048	510	30.6	41.5	17.3	4.1
4096	569	33.2	38.8	17.2	4.2

and Athlon 64 (model 4000+). They consume 115 and 89 Watts, respectively. The Intel Core Duo T2400, the most power efficient CPU in this comparison, has a maximum power consumption of only 31 Watts even when both cores are active.

This comparative study should be considered very preliminary, because we are not running the same code in every architecture. The GPU code is assumed to use $175 + 95/2 = 222.5$ Watts as it also uses one of the CPU cores to assist the computation. This is also a rough estimate as it is likely that when one core is idle, the whole processor's power consumption is more than half of its maximum. However, our rationale is that it is possible to use the other core to perform the same computation on a different input.

The results, outlined in Table 8, show that the Nvidia Cuda implementation is not only powerful, but also efficient. The closest competitor is the auto generated Spiral [19] code that runs on Pentium 4. Note that Pentium 4 is not a particularly power efficient processor. Therefore, it is plausible that an auto generated code on more power efficient hardware would get closer to the efficiency of the GPU. A couple of factors contribute to the inefficiency of Neumann. The most important one being that the Opterons we use are not HE versions, but rather high-performance Opterons. A single Opteron core in Neumann consumes more than three times the power that is consumed by Core 2 Duo, while still giving worse performance in this particular problem.

5. Conclusions and Future Work

We have considered the efficient implementation of Gaussian elimination based algorithms on the GPU. Choosing the right algorithm that efficiently maps to the underlying hardware has always been important in high-performance computing. Our work shows that it is even more important when the hardware in question is a GPU. Our proof-of-concept implementation runs more than two orders of magnitude faster than a simple porting of the most popular algorithm to the GPU. The key to performance was to choose an algorithm that has good locality of reference and makes the most use of optimized kernels.

We made extensive comparisons with our reference implementations on single processor and shared memory multiprocessor systems, as well as with previously reported results obtained on various CPUs and GPUs. Future work includes identifying and implementing crucial kernels that are likely to speed up a large class of applications. Specifically, we are working on implementing an efficient sparse matrix-matrix multiplication algorithm on the GPU, which is to be used as a building block for many graph algorithms [6, 7].

Acknowledgments

We acknowledge the kind permission of Charles Leiserson and CilkArts to use an alpha release of the Cilk++ language. We also thank P.J.Narayanan and Pawan Harish for providing us the exact timings from their experiments. Sivan Toledo helped us improve the presentation of the paper with various comments. Also, thanks to Fenglin Liao and Arda Atali for their help during the initial implementation on Cuda.

References

[1] AMD Stream Computing. `http://ati.amd.com/technology/streamcomputing`.
[2] Cilk Arts. `www.cilk.com/`.
[3] NVIDIA CUDA. `http://www.nvidia.com/cuda`.
[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, Boston, MA, USA, 1974.
[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schrder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, 2003.
[6] A. Buluç and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *The 37th International Conference on Parallel Processing (ICPP'08)*, pages 503–510, Portland, Oregon, USA, September 2008.
[7] A. Buluç and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, April 2008.
[8] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *SODA '06: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 591–600, New York, NY, USA, 2006. ACM.
[9] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.
[10] P. D'Alberto and A. Nicolau. R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, 2007.
[11] W. J. Dally. Keynote address: "Stream programming : Parallel processing made simple". In *ICPP '08: Proc. of the Intl. Conf. on Parallel Processing*. IEEE Computer Society, September 2008.
[12] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[13] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 133–137, New York, 2004. ACM.

[14] W. Feng, X. Feng, and R. Ge. Green supercomputing comes of age. *IT Professional*, 10(1):17–23, 2008.

[15] J. G. Fletcher. A more general algorithm for computing closed semiring costs between vertices of a directed graph. *Communications of the ACM*, 23(6):350–351, 1980.

[16] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[18] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997.

[19] S.-C. Han, F. Franchetti, and M. Püschel. Program generation for the all-pairs shortest path problem. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation techniques*, pages 222–232, New York, 2006. ACM.

[20] P. Harish and P.J.Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International Conference on High Performance Computing (HiPC 2007)*, 2007.

[21] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

[22] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[23] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *ICPP '87: Proc. of the Intl. Conf. on Parallel Processing*, pages 713–716, 1987.

[24] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[25] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[26] NVIDIA. CUDA Programming Guide 1.1, 2007. `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[27] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[28] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[29] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[30] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, 2007.

[31] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual (With CD-ROM)*. Addison-Wesley Professional, 2001.

[32] V. Strassen. Gaussian elimination is not optimal. *Numerical Math.*, 13:354–356, 1969.

[33] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[34] A. Tiskin. Synchronisation-efficient parallel all-pairs shortest paths computation (work in progress), 2004. `http://www.dcs.warwick.ac.uk/~tiskin/pub/2004/apsp.ps`.

[35] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[36] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3(2-4):331–360, 1991.

[37] V. Volkov and J. Demmel. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[38] U. Zwick. Exact and approximate distances in graphs - a survey. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 33–48. Springer-Verlag, 2001.

Appendix A. Additional Timing Results

Table 9 shows the timings obtained on Intel Core 2 Duo, using MS Visual Studio 2003's C++ compiler. For small inputs ($|V| \leq 1024$), the recursive implementation performs better due to its cache friendliness. For larger inputs, however, the overhead of recursion starts to dominate the running time. We have also experimented with the Boost Graph Library's Floyd-Warshall implementation [31] but found it to be consistently slower than our implementations. This might be due to the overheads coming from the genericity of Boost. Therefore, we excluded its running times from our plots in the main text.

Table 9: Serial timings on Intel Core 2 Duo (in milliseconds)

Num. of Vertices	Iterative	Recursive	Boost
512	8.43×10^2	7.81×10^2	1.37×10^3
1024	7.40×10^3	7.35×10^3	1.16×10^4
2048	5.94×10^4	7.98×10^4	9.19×10^4
4096	4.79×10^5	7.20×10^5	7.27×10^5
8192	3.77×10^6	5.82×10^6	N.A.

In Table 10, we list the performance of our reference implementations, compiled both with GCC and Intel C/C++ compiler version 9.1 (ICC). Although Intel's compiler consistently outperformed GCC, its performance still lags behind the performance achieved by MS Visual Studio on Intel.

Table 10: Serial timings on Opteron (in milliseconds)

Num. of Vertices	Iterative		Recursive	
	GCC	ICC	GCC	ICC
512	1.30×10^3	9.90×10^2	1.60×10^3	1.14×10^3
1024	1.07×10^4	8.31×10^3	1.34×10^4	9.74×10^3
2048	8.41×10^4	6.41×10^4	1.32×10^5	1.03×10^5
4096	6.66×10^5	5.03×10^5	1.24×10^6	1.00×10^6
8192	N.A.	3.94×10^6	N.A.	1.58×10^7