Accelerating CUDA Graph Algorithms at Maximum Warp

Sungpack Hong Sang Kyun Kim Tayo Oguntebi Kunle Olukotun

Computer Systems Laboratory
Stanford University
{hongsup, skkim38, tayo, kunle}@stanford.edu

Abstract

Graphs are powerful data representations favored in many computational domains. Modern GPUs have recently shown promising results in accelerating computationally challenging graph problems but their performance suffers heavily when the graph structure is highly irregular, as most real-world graphs tend to be. In this study, we first observe that the poor performance is caused by work imbalance and is an artifact of a discrepancy between the GPU programming model and the underlying GPU architecture. We then propose a novel virtual warp-centric programming method that exposes the traits of underlying GPU architectures to users. Our method significantly improves the performance of applications with heavily imbalanced workloads, and enables trade-offs between workload imbalance and ALU underutilization for fine-tuning the performance.

Our evaluation reveals that our method exhibits up to 9x speedup over previous GPU algorithms and 12x over single thread CPU execution on irregular graphs. When properly configured, it also yields up to 30% improvement over previous GPU algorithms on regular graphs. In addition to performance gains on graph algorithms, our programming method achieves 1.3x to 15.1x speedup on a set of GPU benchmark applications. Our study also confirms that the performance gap between GPUs and other multi-threaded CPU graph implementations is primarily due to the large difference in memory bandwidth.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – Patterns

General Terms Algorithms, Performance

Keywords Parallel graph algorithms, CUDA, GPGPU

1. Introduction

Graphs are widely-used data structures that describe a set of objects, referred to as *nodes*, and the connections between them, called *edges*. Certain graph algorithms, such as breadth-first search, minimum spanning tree, and shortest paths, serve as key components to a large number of applications [4, 5, 15–17, 22, 25] and have thus been heavily explored for potential improvement. Despite the considerable research conducted on making these algorithms efficient, and the significant performance benefits they have reaped due to ever-increasing computational power, processing large irregular graphs quickly and effectively remains an immense challenge today. Unfortunately, many real-world applications involve large irregular graphs. It is therefore important to fully exploit the fine-grain parallelism in these algorithms, especially as parallel computation resources are abundant in modern CPUs and GPUs.

The Parallel Random Access Machine (PRAM) abstraction has often been used to investigate theoretical parallel performance of graph algorithms [18]. The PRAM abstraction assumes an infinite number of processors and unit latency to shared memory from any of the processors. Actual hardware approximations of PRAM, however, have been rare. Conventional CPUs lack in number of processors, and clusters of commodity general-purpose processors are poorly-suited as PRAM approximations due to their large inter-node communication latencies. In addition, clusters which span multiple address spaces impose the added difficulty of partitioning the graphs. In the supercomputer domain, several accurate approximations of the PRAM , such as the Cray XMT [13], have demonstrated impressive performance executing sophisticated graph algorithms [5, 17]. Unfortunately, such machines are prohibitively costly for many organizations.

GPUs have recently become popular as general computing devices due to their relatively low costs, massively parallel architectures, and improving accessibility provided by programming environments such as the Nvidia CUDA framework [23]. It has been observed that GPU architectures closely resemble supercomputers, as they implement the primary PRAM characteristic of utilizing a very large number of hardware threads with uniform memory latency. PRAM algorithms involving irregular graphs, however, fail to perform well on GPUs [15, 16] due to the workload imbalance between threads caused by the irregularity of the graph instance.

In this paper, we first observe that the significant performance drop of GPU programs from irregular workloads is an artifact of a discrepancy between the GPU hardware architecture and direct application of PRAM-based algorithms (Section 2). We propose a novel virtual warp-centric programming method that reduces the inefficiency in an intuitive but effective way (Section 3). We apply our programming method to graph algorithms, and show significant speedup against previous GPU implementations as well as a multi-threaded CPU implementation. We discuss why graph algorithms can execute faster on GPUs than on multi-threaded CPUs. We also demonstrate that our programming method can benefit other GPU applications which suffer from irregular workloads (Section 4).

This work makes the following contributions:

- We present a novel virtual warp-centric programming method which addresses the problem of workload imbalance, a general issue in GPU programming. Using our method, we improve upon previous implementations of GPU graph algorithms, by several factors in the case of irregular graphs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'11, Feb 12–16,2011,San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-01190-0/11/02. . . $10.00

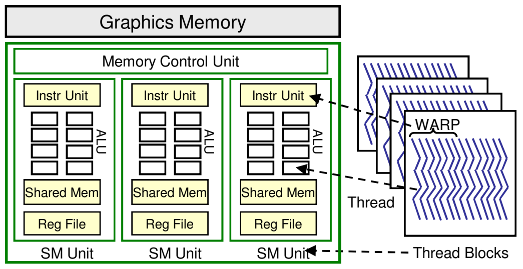


Figure 1. GPU architecture and thread execution model in CUDA.

- Our method provides a generalized, systematic scheme of warp-wise task allocation and improves the performance of GPU applications which feature heavy branch divergence or (unnecessary) scattering of memory accesses. Notably, it enables users to easily make the necessary trade-off between SIMD under-utilization and workload imbalance with a single parameter. Our method boosts the performance of a set of benchmark applications suffering from these issues by 1.3x – 15.1x.
- We provide a comparative analysis featuring a GPU and two different CPUs that examines which architectural traits are critical to the performance of graph algorithms. In doing so, we show that GPUs can outperform other architectures by providing sufficient random access memory bandwidth to exploit the abundant parallelism in the algorithm.

2. Background

2.1 GPU Architectures and the CUDA Programming Model

In this section, we briefly review the microarchitecture of modern graphics processors and the CUDA programming model approach to using them. We then provide a sample graph algorithm and illustrate how the conventional CUDA programming model can result in low performance despite abundant parallelism available in the graph algorithm. In this paper, we focus on Nvidia graphics architectures and terminology specific to their products. The concepts discussed here, however, are relatively general and apply to any similar GPU architecture.

2.2 Graph Algorithms on GPU

Figure 1 depicts a simplified block diagram of a modern GPU architecture, only displaying modules related to general purpose computation. As seen in the diagram, typical general-purpose graphics processors consist of multiple identical instances of computation units called Stream Multiprocessors (SM). An SM is the unit of computation to which a group of threads, called thread blocks, are assigned by the runtime for parallel execution. Each SM has one (or more) unit to fetch instructions, multiple ALUs (i.e., stream processors or CUDA cores) for parallel execution, a shared memory accessible by all threads in the SM, and a large register file which contains private register sets for each of the hardware threads. Each thread of a thread block is processed on an ALU in the SM. Since ALUs are grouped to share a single instruction unit, threads mapped on these ALUs execute the same instruction each cycle, but on different data. Each logical group of threads sharing instructions is called a warp. [1] Moreover, threads belonging to different warps can execute different instructions on the same ALUs, but in a different time slot. In effect, ALUs are time-shared between warps.

[1] Note that the number of ALUs sharing an instruction unit (e.g. 8) can be smaller than the warp size. In such cases, the ALUs are time-shared between threads in a warp; this resembles vector processors whose vector length is larger than the number of vector lanes.

The following summarizes the discussion above: from the architectural standpoint, a group of threads in a warp performs as a SIMD(Single Instruction Multiple Data) unit, each warp in a thread block as a SMT(Simultaneous Multithreading) unit, and a thread block as a unit of multiprocessing.

That said, modern GPU architectures relax SIMD constraints by allowing threads in a given warp to execute different instructions. Since threads in a warp share an instruction unit, however, these varying instructions cannot be executed concurrently and are serialized in time, severely degrading performance. This advanced feature, so called SIMT (Single Instruction Multiple Threads), provides increased programming flexibility by deviating from SIMD at the cost of performance. Threads executing different instructions in a warp are said to diverge; if-then-else statements and loop-termination conditions are common sources of divergence.

Another characteristic of a graphics processor which greatly impacts performance is its handling of different simultaneous memory requests from multiple threads in a warp. Depending on the accessed addresses, the concurrent memory requests from a warp can exhibit three possible behaviors:

1. Requests targeting the same address are merged to be one unless they are atomic operations. In the case of write operations, the value actually written to memory is nondeterministically chosen from among merged requests.
2. Requests exhibiting spatial locality are maximally coalesced. For example, accesses to addresses i and $i + 1$ are served by a single memory fetch, as long as they are aligned.
3. All other memory requests (including atomic ones) are serialized in a nondeterministic order.

This last behavior, often called the scattering access pattern, greatly reduces memory throughput, since each memory request utilizes only a few bytes from each memory fetch.

To best utilize the aforementioned graphics processors for general purpose computation, the CUDA programming model was introduced recently by Nvidia [23]. CUDA has gained great popularity among developers, engineers, and scientists due to its easily accessible compiler and the familiar C-like constructs of its API extension. It provided a method of programming a graphics processor without thinking in the context of pixels or textures.

There is a direct mapping between CUDA's thread model and the PRAM abstraction; each thread is identified by its thread ID and is assigned to a different job. External memory access takes a unit amount of time in a massive threading environment, and no concept of memory coherence is enforced among executing threads. The CUDA programming model extends the PRAM abstraction to include the notion of shared memory and thread blocks, a reflection of the underlying hardware architecture as shown in Figure 1. All threads in a thread block can access the same shared memory, which provides lower latency and higher bandwidth access than global GPU memory but is limited in size. Threads in a thread block may also communicate with each other via this shared memory. This widely-used programming model efficiently maps computation kernels onto GPU hardware for numerous applications such as matrix multiplication.

The PRAM-like CUDA's thread model, however, exhibits certain discrepancies with the GPU microarchitecture that can significantly degrade performance. Especially, it provides no explicit notion of warps; they are transparent to the programmers due to the SIMT ability of the processors to handle divergent threads. As a result, applications written according to the PRAM paradigm will likely suffer from unnecessary path divergence, particularly when each *task* assigned to a thread is completely independent from other tasks. One example is parallel graph algorithms, where the irregular nature of real-world graph instances often induce extreme branch

```
 1  struct graph {
 2    int nodes[N+1];//start index of edges from nth node
 3    int edges[M];//destination node of mth edge
 4    int levels[N]; // will contatin BFS level of nth node
 5  };
 6
 7  void bfs_main(graph* g, int root) {
 8     initialize_levels(g->levels, root);
 9     curr = 0; finished = false;
10     do {
11       finished = true;
12       launch_gpu_bfs_kernel(g, curr++, &finished);
13     } while (!finished);
14  }
15  __kernel__
16  void baseline_bfs_kernel(int N, int curr, int *levels,
17    int *nodes, int *edges, bool* finished) {
18      int v = THREAD_ID;
19      if (levels[v] == curr) {
20   // iterate over neighbors
21        int num_nbr = nodes[v+1] - nodes[v];
22        int* nbrs = & edges[ nodes[v] ];
23        for(int i = 0; i < num_nbr; i++) {
24          int w = nbrs[i];
25          if (levels[w] == INF) { // if not visited yet
26            *finished = false;
27            levels[w] = curr + 1;
28  } } } }
```

Figure 2. The baseline GPU implementation of BFS algorithm.

divergence problems and scattering memory access patterns, as will be explained in the next section. In Section 3, we introduce a new generalized programming method that uses its awareness of the warp concept to address this problem.

Figure 2 is an example of a graph algorithm written in CUDA using the conventional PRAM-style programming from a previous work [15]. [2] This algorithm performs a breadth-first search (BFS) on the graph instance, starting from a given *root* node. More accurately, it assigns a "BFS level" to every (connected) vertex in the graph; the level represents the minimum number of hops to reach this node from the root node.

Figure 2 also describes the graph data structure used in the BFS, which is the same as the data structures used in other related work [6, 15, 16]. This data structure consists of an array of nodes and edges, where each element in the nodes array stores the start index (in the edges array) of the edges outgoing from each node. The edges array stores the destination nodes of each edge. The last element of the nodes array serves as a marker to indicate the length of the edges array. Figure 3.(a) visualizes the data structure. [3]

For this algorithm, the level of each node is set to ∞, except for the root which is set to zero. The kernel (code to be executed on the GPU) is called multiple times until all reachable nodes are visited, incrementing the current level by one upon each call. At each invocation, each thread visits a node which has the same current_level and marks all unvisited neighbors of the node with current_level+1. Nodes may be marked multiple time within a kernel invocation, since updates are not immediately visible to all threads. This does not affect correctness, as all updates will use the same correct value. This paper mainly focuses on the BFS algorithm, but our discussion can be applied to many similar parallel graph algorithms that process multiple nodes in parallel while exploring neighboring nodes from each. We will discuss some of these algorithms in Section 4.3.

The baseline BFS implementation shown in Figure 2 suffers a severe performance penalty when the graph is highly irregular, i.e. when the distribution of degrees (number of edges per node) is highly skewed. As we will show in Section 4, the baseline algorithm yields only a 1.5x speedup over a single-threaded CPU when the graph is very irregular. Performance degradation comes from execution path divergence at lines 19, 23, and 25 in Figure 2. Specifically, a thread that processes a high-degree node will iterate the loop at line 23 many more times than other threads, stalling other threads in its warp. Additional performance degradation comes from non-coalesced memory operations at lines 21, 22, 25, and 27 since their addresses exhibit no spatial locality across the threads. In addition, a repeated single-threaded access over consecutive memory addresses (i.e., at line 21) actually wastes memory bandwidth by failing to exploit spatial locality in memory accesses.

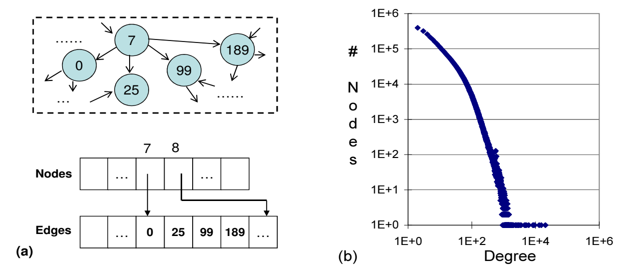


Figure 3. (a) A visualization of the graph data structure used in the BFS algorithm. (b) A degree distribution of a real-world graph instance (LiveJournal), which we used for our evaluation in Section 4.

Unfortunately, the nature of most real-world graph instances is known to be irregular [24]. Figure 3.(b) displays the degree distribution from one such example. Note that the plot is presented in log-log format. The distribution shows that although the average degree is small (about 17), there are many nodes which have degrees 10x $\sim$ 100x (and some even 1000x) larger than the average.

3. Addressing Irregular Workloads using GPUs

3.1 Virtual Warp-centric Programming Method

We introduce a novel virtual warp-centric programming method which explicitly exposes the underlying SIMD nature of the GPU architecture to achieve better performance under irregular workloads.

Generalized warp-based task allocation

Instead of assigning a different task to each thread as is typical in PRAM-style programming, our approach allocates a chunk of tasks to each warp and executes distinct tasks as serial. We utilize multiple threads in a warp for explicit SIMD operations only, thereby preventing branch-divergence altogether. More specifically, the kernel in our programming model alternates between two phases: the SISD (Single Instruction Single Data) phase, which is the default serial execution mode, and the SIMD phase, the parallel execution mode. When the kernel is in the SISD phase, only a single stream of instructions is executed by each warp. In this phase, each warp is identified by a unique warp ID and works on an independent set of tasks. The degree of parallelism is thus maintained by utilizing multiple warps. In contrast, the SIMD phase begins by entering a special function explicitly invoked by the user. Once in the SIMD phase, each thread in the warp follows the same instruction sequence, but on different data based on its warp offset, or the lane ID within the given SIMD width. Unlike the classical (CPU-based) SIMD programming model, however, our SIMD threads are allowed more flexibility in executing instructions; they

[2] The algorithm presented actually contains additional optimizations we made to the original version [15]; we eliminated unnecessary memory accesses and also eliminated an entire secondary kernel, which resulted in more than 20% improvement. We use this optimized version as our baseline.

[3] This data-structure is also known as compressed sparse row (CSR) in sparse-matrix computation domain [9].

```
29  template<int W_SZ> __device__
30  void memcpy_SIMD
31  (int W_OFF, int cnt, int* dest, int* src) {
32    for(int IDX = W_OFF; IDX < cnt; IDX += W_SZ)
33       dest[IDX] = src[IDX];
34    __threadfence_block(); }
35
36  template<int W_SZ> __device__
37  void expand_bfs_SIMD
38  (int W_SZ, int W_OFF, int cnt, int* edges,
39   int* levels, int curr, bool* finished) {
40    for(int IDX = W_OFF; IDX < cnt; IDX += W_SZ) {
41      int v = edges[IDX];
42      if (levels[v] == INF) {
43         levels[v] = curr + 1;
44         *finished = false;
45    } }
46    __threadfence_block();}
47
48  struct warpmem_t {
49    int levels[CHUNK_SZ];
50    int nodes[CHUNK_SZ + 1];
51    int scratch;
52  };
53  template<int W_SZ> __kernel__
54  void warp_bfs_kernel
55  (int N, int curr, int *levels,
56   int *nodes, int *edges, bool* finished) {
57    int W_OFF = THREAD_ID % W_SZ;
58    int W_ID = THREAD_ID / W_SZ;
59    int NUM_WARPS = NUM_THREADS / W_SZ;
60    extern __shared__ warp_mem_t SMEM[];
61    warpmem_t* MY = SMEM + (LOCAL_THREAD_ID / W_SZ);
62
63  // copy my work to local
64    int v_ = W_ID * CHUNK_SZ;
65    memcpy_SIMD<W_SZ> (W_OFF, CHUNK_SZ,
66      MY->levels, &levels[v_]);
67    memcpy_SIMD<W_SZ> (W_OFF, CHUNK_SZ+1,
68      MY->nodes, &nodes[v_]);
69
70  // iterate over my work
71    for(int v = 0; v < CHUNK_SZ; v++) {
72      if (MY->levels[v] == curr) {
73        int num_nbr = MY->nodes[v+1] - MY->nodes[v];
74        int* nbrs = &edges[ MY->nodes[v] ];
75        expand_bfs_SIMD<W_SZ> (W_OFF, num_nbr,
76          nbrs, levels, curr, finished)
77  } } }
```

Figure 4. BFS kernel written in virtual warp-centric programming model

can perform scattering/gathering memory accesses, execute conditional operations independently, and process dynamic data width. This is all done while taking advantage of the underlying hardware SIMT feature.

The proposed programming method has several advantages:

1. Unless explicitly intended by the user, this approach never encounters execution-path divergence issues. Intra-warp workload imbalance is therefore never unaware.
2. Memory access patterns can be more coalesced than the conventional thread-level task allocation in applications where concurrent memory accesses within a task exhibit much higher spatial locality than across different tasks.
3. Many developers are already familiar with our approach, since it resembles, in many ways, the traditional SIMD programming model for CPU architectures. However, the proposed approach is even simpler and more powerful than SIMD programming for CPU, since CUDA allows users to describe custom SIMD operations with C-like syntax.
4. This method allows for each task to allocate a substantial amount of privately-partitioned *shared memory* per task. This is because there are fewer warps than threads in a thread block.

In order to generally apply our programming method within current GPU hardware and compiler environments, we take simple means of replicated computation: during the SISD phase, every thread in a warp executes exactly the same instruction on exactly the same data. We enforce this by assigning the same warp ID to all threads in a warp. Note that this does not waste memory bandwidth since accesses from the same warp to the same destination address are merged into one by the underlying hardware.

Virtual Warp Size

Although naive warp-granular task allocation provides several merits aforementioned, it suffers from two potential drawbacks, where in both cases, unused ALUs within a warp limit the parallel performance of kernel execution:

1. If the native SIMD width of the user application is small, the underlying hardware will be under-utilized.
2. The ratio of the SIMD phase duration to the SISD phase duration imposes an Amdahl's limit on performance.

We address these issues by logically partitioning a warp into multiple virtual warps. Specifically, instead of setting the warp size parameter value to be the actual physical warp size of 32, we use a divisor (i.e. 4, 8, and 16). Multiple virtual warps are then co-located in one physical warp, with each virtual warp processing a different task. Note that all previous assumptions on a warp's execution behavior – synchronized execution and merged memory accesses for the threads inside a warp – are still valid within virtual warps. Thus, the parallelism of the SISD phase increases as a result of having multiple virtual warps for each physical warp, and the ALU utilization improves as well due to the logically narrower SIMD width.

Using virtual warps leads to the possibility of execution path divergence among different virtual warps, which in turn serializes different instruction streams among the warps. The degree of divergence among virtual warps, however, is most likely much less than among threads in a conventional PRAM warp. In essence, the virtual warp scheme can be viewed as a trade-off between execution-path divergence and ALU underutilization by varying a single parameter, the virtual warp size.

BFS in the Virtual Warp-centric Programming Method

Figure 4 displays the implementation of the BFS algorithm using our virtual warp-centric method. While the underlying BFS algorithm is fundamentally identical to the baseline implementation in Figure 2, the new implementation divides into SISD and SIMD phases. The main kernel (lines 54-77) executes the same instruction and data pattern for every thread in the warp, thus operating in the SISD phase. Functions in lines 30-46 operate in the SIMD phase, since distinct partitions of data are processed. Each warp also uses a private partition of shared memory; the data structure in lines 48-51 illustrates the layout of each private partition.

Lines 57-61 of the main kernel define several utility variables. The virtual-warp size (`W_SZ`) is given as a template parameter; the warp ID (`W_ID`) of the current warp and the warp offset (`W_OFF`) of each thread is computed using the warp size. Warp-private memory space is allocated by setting the pointer (`MY`) to the appropriate location in the shared memory space.

The virtual warp-centric implementation copies its portion of work to the private memory space (lines 64-68) before executing the main loop. As the function name implies, the memory copy operation is performed in a SIMD manner. After the memory copy operation finishes, the kernel executes the iterative BFS algorithm

```
78  BEGIN_SIMD_DEF(memcpy, int* dest, int* src)
79  { dest[IDX] = src[IDX]; } END_SIMD_DEF
80
81  BEGIN_SIMD_DEF (expand_bfs, int* edges,
82    int* level, int curr, bool*finished)
83  { int v = edges[IDX];
84    if (level[v] == INF) {
85      level[v] = curr + 1;
86      *finished = false;
87  } } END_SIMD_DEF
88
89  BEGIN_WARP_KERNEL(warp_bfs_kernel,
90     int N, int curr, int *level,
91     int *nodes, int * edges, bool* finished) {
92     USE_PRIV_MEM(warp_mem_t);
93  // copy my_work
94     int v_ = N / NUM_WARPS * W_ID;
95     DO_SIMD(mempcy, CHUNK_SZ, MY->levels, &level[v_]);
96     DO_SIMD(mempcy, CHUNK_SZ+1, MY->nodes, &nodes[v_]);
97
98  // iterate over my_work
99     for(int v = 0; v < CHUNK_SZ; v++) {
100      if (level[v] == curr) {
101        int num_nbr = MY->nodes[v+1] - MY->nodes[v];
102        int* nbrs = &edges[ MY->nodes[v] ];
103        DO_SIMD(expand_bfs, num_nbr,
104           nbrs, begin, level, curr, finished)
105 } } } END_WARP_KERNEL
```

Figure 5. Same code as Figure 4 using macro-expansion. Type definition of `warp_mem_t` is same as before and omitted.

sequentially (lines 71-77), with the exception of explicitly-called SIMD functions. The expansion of BFS neighbors (line 75) is an explicit SIMD function call to the one defined at line 37, whose functionality is equivalent to lines 23-27 of the baseline algorithm Figure 2.

For a detailed explanation of how SIMD functions are implemented, consider the simple `memcpy` function in line 30. Each thread in a warp enters the function with a distinct warp offset (`W_OFF`), which leads to a different range of indices (`IDX`) of the data to be copied. The SIMT feature of CUDA enables the width of the SIMD operation to be determined dynamically. Although the SIMT feature guarantees synchronous execution of all threads at the end of the `memcpy` function, `__threadfence_block()` at line 34 is still required for intra-warp visibility of any pending writes before returning to SISD phase.[4] The second SIMD function, `expand_bfs` (line 37), is structured similarly to `memcpy`. The if-then-else statement in line 42 is an example of a conditional SIMD operation, automatically handled by SIMT hardware.

Using the virtual warp-centric method, the BFS code exhibits no execution-path divergence other than intended dynamic widths and conditional operations, as shown in Figure 4. Moreover, memory accesses are coalesced except the final scattering at line 42 and 43, which are inherent to the nature of the BFS algorithm.

Abstracting the Virtual Warp-centric Programming Method

As evident in the BFS example, the virtual warp-centric programming method is intuitive enough to be manually applied by GPU programmers. Closer inspection of the code in Figure 4, however, reveals some structural repetition in patterns that serve the programming method itself, rather than the user algorithm. Thus, providing an appropriate abstraction for the model can further reduce programmer effort as well as potential for error in the structural part of the program.

To this end, we introduce a small set of syntactic constructs intended to facilitate use of the programming model. Figure 5 illustrates how these constructs can simplify our previous warp-centric BFS implementation. For example, the SIMD function `memcpy` (line 30-33) in Figure 4 can be concisely expressed as line 78-79 in Figure 5. The constructs `BEGIN_SIMD_DEF` and `END_SIMD_DEF` automatically generate the function definition and outer-loop for work distribution. The user invokes the SIMD function using the `DO_SIMD` construct (line 95), where the function name, dynamic width, and other arguments are specified. Similarly, the `BEGIN_WARP_KERNEL` and `END_WARP_KERNEL` constructs indicate and generate the beginning and end of a warp-centric kernel, while the `USE_PRIV_MEM` construct allocates a private partition of shared memory.

[4] Although intra-warp visibility is attainable without the fence in some GPU generations (e.g. GT200), it is not guaranteed in general by the CUDA specification. Also note that the fence guarantees threadblock-wide visibility, which is larger than required; however, the performance impact of the overhead is negligible.

The current set of constructs are implemented as C-macros, which is adequate to demonstrate how these constructs can generate desired routines and simplify programming. However, future compiler support of such virtual warp-centric constructs, or similar, could provide further benefits. For example, the compiler may choose to generate codes for SISD regions such that only a single thread in a warp is actually activated, rather than replicating computation. This eliminates unnecessary allocation of duplicated registers which are used only in the SISD phase and can also save power wasted by replicated computation.

3.2 Other Techniques

In this subsection, we discuss two other general techniques for addressing work imbalance. These techniques do not necessarily rely on the new programming model but can accompany it.

Deferring Outliers

The first technique is deferring execution of exceptionally large-sized tasks, which we term 'outliers'. Since there are a limited number of such tasks which induce load imbalance, we identify these tasks during main-kernel execution and defer their processing by placing them in a globally-shared queue, rather than processing them on-line. In subsequent kernel calls, each of the deferred tasks is executed individually, with its work parallelized across multiple threads. Figure 6 illustrates this idea.

In the BFS algorithm, the amount of work is proportional to the degree of each node, which is obtainable in 0(1) time given our data structure. For this technique, therefore, we simply defer processing of any node having degree greater than a predetermined threshold. Results in Section 4 explore the effects on performance when one varies this threshold.

This optimization technique requires the implementation of a global queue, a challenging task on a GPU in general. It is relatively simple, however, to implement a queue that always grows (or shrinks) during a kernel's execution. The code below exemplifies such an implementation using a single atomic operation:

```
AddQueue(int* q_idx, type_t* q, type_t item) {
  int old_idx = AtomicAdd(q_idx, 1);
  q[old_idx] = item; }
```

In our case, the overhead of the atomic operations is negligible compared to overall execution time, since queuing of deferred outliers is rare. However, this technique presents additional overhead via subsequent kernel invocations to process the deferred outliers.

Dynamic Workload Distribution

The virtual warp-centric programming method addresses the problem of workload imbalance inside a warp. However, there still exists the possibility of workload imbalance between warps: a single warp processing an exceptionally large task can stall the entire thread block (mapped to an SM), wasting computational resources. To solve this problem, we apply a dynamic workload distribution

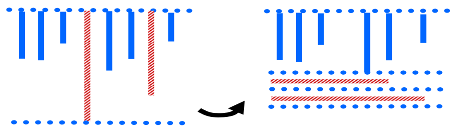


Figure 6. Visualization of deferring outliers: Each solid bar in the figure represents the amount of workload of each task. Tasks that have particularly large workload are deferred and later processed individually in parallel.

technique to our GPU graph algorithm. First, we instantiate only as many warps as there are physical resources. Second, each warp dynamically fetches a chunk of work from a shared work queue and processes it. All warps repeat this process until the work queue is emptied.

Note that this dynamic workload distribution technique is completely ineffective in the conventional PRAM-style programming; each thread in a warp cannot fetch another task until its fellow threads in the warp complete theirs. The same global queue implementation used for the deferring of outliers technique can be used for the work queue, since it only shrinks during a kernel execution. However, atomic operations during SISD phase should be handled with care as in following example:

```
if (WARP_OFFSET==0){
 MY->scratch = AtomicAdd(work_q, CHUNK_SZ);
 __threadfence_block();}
v_ = MY->scratch;
```

That is, since the GPU hardware does not merge, but serializes, atomic accesses that have the same destination address, only one thread should execute it for semantic correctness. The result of the single threaded atomic instruction can be propagated to all the other threads in the warp by means of shared memory without explicit synchronization.

For task parallelization, there is a well-known trade-off between static and dynamic workload distribution: Static distribution is vulnerable to load imbalance for tasks with varying workload, while dynamic distribution often imposes considerable overhead. In our case, the overhead is the atomic mutation of the work queue. The better scheme thus depends upon the size of the work chunk. Results in Section 4 will study this effect.

4. Experimental Results

In this section, we present experimental results in three categories. First, we explore the effectiveness of the methods we designed to address workload imbalance. Second, we study how different traits of various processor architectures affect the performance of graph algorithms. Third, we apply the virtual warp-centric programming method to other types of GPU applications, including two other graph algorithms.

4.1 Effect of Optimization Techniques

In the previous section, we introduced three techniques to avoid execution-path divergence and improve performance. In this section, we evaluate our proposed schemes by applying them in various combinations to the BFS algorithm with graph instances from different sources. Table 1 summarizes the experimental environments used for evaluation. For all results, speedup is measured against single-threaded execution time on the Out-of-Order (OOO) CPU machine described in the table, unless stated otherwise. The single-threaded execution used the same algorithm as the baseline in Figure 2, as it has been shown to be faster than other popular library such as Boost [16]. CPU and GPU BFS codes were compiled using gcc with the '-O3 -m32' flags and nvcc with the '-O3' flag, respectively.

Table 2 summarizes the properties of the graph instances used in the experiment. Of the four used, two of them are synthetically-generated graphs: RMAT and Random. RMAT [11] generates a scale-free graph which follows a power law degree distribution like many real world graphs [24], such as the world-wide web or a social networks. Random is a uniformly distributed graph instance created by randomly connecting *m* pairs of nodes out of a total *n* nodes. The average degree is set to 12 for both instances. Our experiment also includes two real world graph instances from a public large dataset collection [1]. LiveJournal exhibits a very irregular structure; its degree distribution was the example given in Figure 3.(b). Patents is relatively regular and has a smaller average degree.

Figure 7 illustrates the effectiveness of our proposed methods. Graph (a) compares five different configurations used on the GPU. The leftmost bar represents the speedup of the baseline GPU implementation (Figure 2). As evident in the graph, the performance of the baseline algorithm is particularly poor for irregular graph instances (RMAT, LiveJournal) due to the problem of workload imbalance. The next striped bar shows the performance when the deferring method is applied to the baseline. Deferring outliers produces a 1.9x improvement over the baseline implementation for RMAT, which features the worst workload imbalance, but does not provide significant speedup for any other graph instance.

The center solid bar represents the speedup of the BFS implementation using our virtual warp-centric programming method with virtual warp-size equal to physical warp-size (i.e. 32). RMAT and LiveJournal, in particular, exhibit dramatic performance improvements, as their workload imbalance is well-handled by this method The regular graph instance of Random, however, gained only minor improvements. This small speedup came from slightly better-coalesced memory access patterns in the warp-centric method (Section 3.1). For Patents, the warp-centric method resulted in performance degradation. This is not only because Patents is quite regular, but its average degree is small (~ 6). The naive warp-centric method therefore suffered from underutilization of warp resources, since the average SIMD width is determined by the average degree. A more detailed discussion follows shortly.

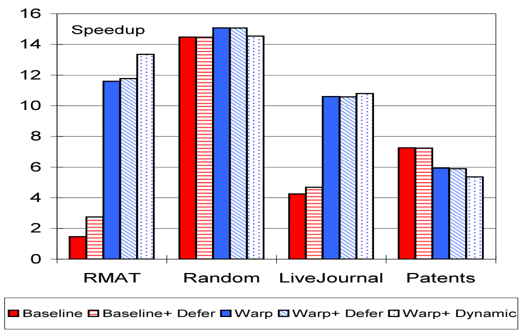
The two rightmost bars in Figure 7.(a) display the performance of the deferred outlier and dynamic workload distribution methods, when combined with the warp-centric method. Deferring outliers yields only a marginal performance gain since the warp-centric programming method already handles load imbalance reasonably well. In contrast, dynamic workload distribution allows for further speedup in severely imbalanced graphs such as RMAT. If the workload is evenly distributed, however, this method yields more overhead than benefit, as shown in the Random and Patent instances.

We now explore the trade-off between execution path divergence and ALU underutilization, by varying the virtual warp size. The result is shown in Figure 7.(b). The figure depicts the different optimal trade-off points for the graph instances. The regular graph instances (Random and Patents) achieve maximum average utilization of the SIMD lanes when the virtual warp size is set to approximately the average degree value. For RMAT, whose degree distribution is severely skewed, the best performance was achieved when avoiding execution path divergence as much as possible; a virtual warp size of 32 yielded optimal results. These results support the argument of having an intelligent runtime system that dynamically adjusts the virtual warp size according to the nature of the graph instance. With a properly selected virtual warp size, the virtual warp-centric method outperforms the baseline algorithm in all graph instances.

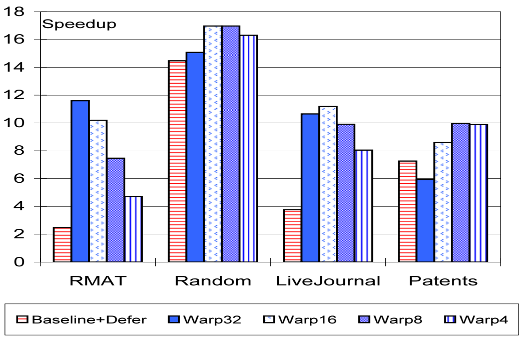
Figure 8 summarizes a sensitivity study of how performance was affected by the chosen parameter values and graph instance sizes. Figure 8.(a) shows the effect of changing the threshold value in the deferring outlier method; this value dictates the minimum

Architecture Name	Detailed Name	SMP sockets	Cores per chip	SMT per core	Last level cache size	Clock freq	Memory size, type
OOO CPU	Intel Xeon E5345	2	4	1	8 MB	2.3 Ghz	32GB, FB-DIMM DDR2
GPU	Nvidia Tesla GTX260	1	24	16	-	1.2 Ghz	1GB, GDDR3
SMT CPU	Sun UltraSPARC T2+	4	8	8	4 MB	1.6 Ghz	128GB, FB-DIMM DDR2
GPU2	Nvidia Tesla GTX275	1	30	16	-	1.4 Ghz	1GB, GDDR3

Table 1. Specifications of machines used in the experiments. GPU2 is only used to create Figure 10.

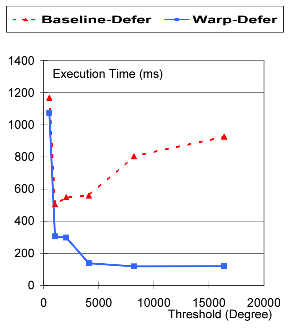


(a) Comparison of optimization techniques.

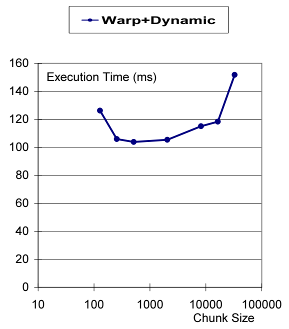


(b) Effect of virtual warp size.

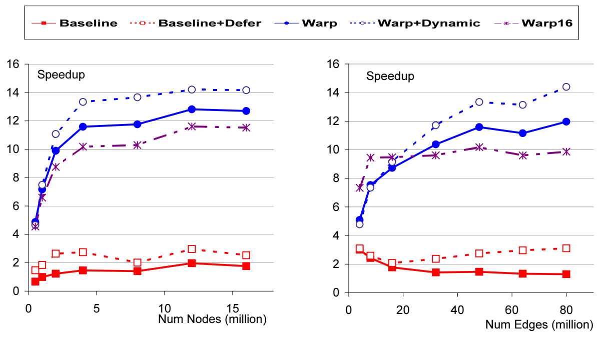
Figure 7. Effect of optimizations with different graph instances.



(a) Deferment threshold



(b) Chunk-size



(c) Number of nodes (d) Number of edges

Figure 8. Sensitivity analysis on optimization parameters and input graph instance sizes: (a) and (b) used the same RMAT graph instance as in previous experiments and display the absolute execution time (lower is better) , while (c) and (d) used RMAT graph instances generated with different size parameters and display speedup against CPU execution (higher is better).

Name	Source	Nodes	Edges	Shape
RMAT	generate [7, 11]	$4 * 10^6$	$4.8 * 10^7$	irregular
Random	generate [7]	$4 * 10^6$	$4.8 * 10^7$	regular
LiveJournal	Real-world data [1]	4,308,451	68,993,773	irregular
Patents	Real-world data [1]	1,765,311	10,564,104	regular

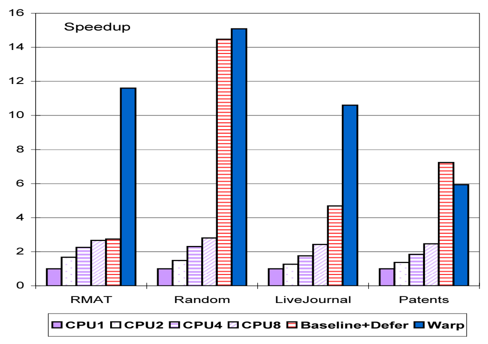
Table 2. Characteristics of graph instances used in the experiments.

size of workload to be deferred. When applied to the baseline implementation (dashed line), the graphs shows that the resulting execution time is highly sensitive to the threshold value. Large values cause workload imbalance to persist; small ones result in too many additional kernel invocations for the deferred work. When combined with the virtual warp-centric method, however, the execution time is insensitive to the threshold value since our method adequately handles load imbalance already.

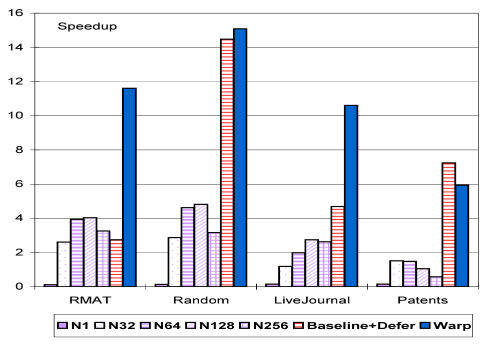
Figure 8.(b) shows the impact of workload chunk-size for the dynamic workload distribution method. This is a classical tradeoff where small chunk-sizes increase the queue overhead, while large chunk-sizes tend to yield workload imbalance. Yet, the overall performance remains relatively insensitive to the chunk-size for a wide range (256 - 2048).

Finally, the plots in Figure 8.(c) and (d) show the scalability of each algorithm, as determined by the input size. Figure 8.(c) sweeps the number of nodes in the RMAT graph instance while fixing the average degree to 12. The relative speedup for the small graph instances in this GPU-based algorithm is low, because the CPU performed well due to increased cache hit rates. The plot clearly shows, however, that virtual warp-centric implementations provide superior speedup as the graph instances become large.

Figure 8.(d) varies the number of edges while keeping the number of nodes constant (4 million). This affects RMAT graph generation in two ways: (i) the average degree changes, and (ii) the graph instance becomes more skewed as the average degree increases. As a result, Figure 8.(d) confirms that when the average degree is small (especially when smaller than the warp size), the naive warp-centric implementations suffer from underutilization in the SIMD operations. In such cases, using a smaller virtual warp size (i.e. warp16) provides meaningful benefit. When the average degree is large, however, the load imbalance becomes more severe due to the increased level of degree skew; dynamic load balancing provides a larger benefit for such cases.



(a) Comparison against an out-of-order SMP.



(b) Comparison against an SMT SMP.

Figure 9. Comparison against CPU execution.

4.2 Study of Architectural Effects

As seen in the previous subsection, the GPU implementations of our graph algorithm significantly outperform their CPU counterparts. We now explore which GPU architectural traits bring about this result and how.

In general, there are a few key architectural characteristics which account for whether performance improvements are delivered by GPU execution for a user application:

(1) GPU architectures enable massively parallel execution by providing many physical cores (SMs) and many ALUs per core.
(2) GPUs use a large number of warps to hide memory latency.
(3) GPU execution takes advantage of the directly-attached GDDR3 memory, which has higher bandwidth and lower latency than FB-DIMM based CPU main memory.
(4) GPUs do not spend bandwidth on coherence traffic.

To explore the effect of these traits on the graph algorithms, we first compared GPU performance to that of other multi-threaded processor architectures; we used the same BFS algorithm [5] on all the machines listed in Table 1 and measured performance.

Figure 9.(a) compares the speedup result on the OOO machine with the GPU results from Figure 7. The four leftmost bars represent the OOO machine execution with varying number of threads. The OOO execution results exhibit similar speedup values for every graph instance, since execution path divergence is not an issue for CPUs. However, the speedup was limited to around 2x with eight cores. Recall that there is no parallel overhead such as locks or atomic operations in our algorithm. Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm. The OOO performance for the Patents instance was particularly better than others, due to relatively higher cache hit rates resulting from the small graph size.

Figure 9.(b) shows the result of the same experiment on the SMT machine. The measured single thread performance (T1) was very poor compared to OOO execution, although the clock frequency of the SMT machine was more than half that of the OOO machine. This implies that the out-of-order execution successfully exploited the high degree of instruction level parallelism within the most time-consuming loop (line 23-27 in Figure 2). Once an adequate number of threads are used, however, the SMT machine soon outperforms the OOO machine due to higher throughput as shown in the 64-thread (T64) results. More interestingly, utilizing more sockets of the SMT machine (two sockets for T128 and four for T256) does not necessarily result in better performance. This indicates that communication cost across chip boundaries, including bandwidth used for coherence traffic, is the major limiting factor for scalability on this machine.

Next, we explore the effect of bandwidth utilization and latency hiding in GPUs. Our experiment involves varying the number of warps and SMs when executing the dynamic load-balancing version of the warp-centric BFS. The result is shown in the two plots of Figure 10.(a); both plots are aligned to the number of active warps. The upper plot explores how latency hiding affects speedup by setting the number of SMs to the maximum (24) and changing the number of warps per SM. The performance gain from adding more than eight warps per SM is marginal, which indicates that performance is limited by the memory bandwidth after that point. The lower plot varies the number of SMs while fixing the number of warps per SM (to 16). Since each SM has limited internal memory bandwidth, more SMs performed better for the same number of warps.

Figure 10.(b) repeats the same experiment on a different GPU machine (GPU2 in Table 1). The plots also suggest that the performance is limited by the bandwidth, since GPU2, whose memory bandwidth is $\sim$10% larger than GPU1 [2], resulted in about 10% performance improvement. We also measured the memory bandwidth of random read accesses on these machines (omitted for brevity) and confirmed that the GPU has 3x $\sim$ 16x larger bandwidth than OOO and SMT CPUs, depending on the cache hit ratio.

To summarize, the graph algorithm has abundant parallelism, primarily in memory operations rather than computation; this parallelism can be exploited by using deep OOO execution, SMT hardware, or scattering SIMD operations as well as by using multiple cores and chips. Regardless of the hardware mechanism chosen, graph algorithms such as BFS are bound by memory bandwidth and require as much as possible in order to realize peak performance. The GPU had the largest memory bandwidth of all the machines we tested and therefore showed the best performance.

One common criticism of graph algorithm computation on GPUs is lack of support for very large graphs that do not fit in the GPU's limited memory. It would therefore be interesting to consider how the GPU would perform if the directly attached GDDR memory is replaced by FB-DIMM memory. This would allow GPUs to accommodate larger graph instances at the expense of a reduction in memory bandwidth. In such a case, the GPU performance should decrease, as per the reduced memory bandwidth, although the non-coherence and massive parallelism of the GPU enables it to utilize its bandwidth more efficiently. The merits of the CPU's cache will also diminish as data size grows.

Nevertheless, many real-world graphs (e.g. all instances in [1, 6]) still fit in GPU memory, and many interesting queries on these

[5] We also tried another multi-threaded CPU implementation [6] that uses spinlocks and private temporary work queues. However, the presented algorithm showed better performance in every configuration.

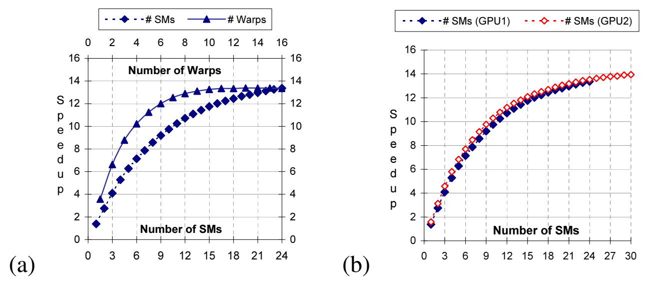


Figure 10. Effect of number of warps and number of SMs for GPU execution (Warp+Dynamic) with RMAT input.

graphs require a considerable amount of time, e.g calculating the average BFS number of all nodes from all other nodes requires O(N) repeated execution of single BFS number algorithm. Thus, fast GPU execution is practically very beneficial for many real-world problems.

4.3 Other Applications

In this subsection, we demonstrate that the virtual warp-centric programming method can be applied to other graph algorithms and provides the same merits as in the BFS algorithm. In addition, we apply our method to GPU applications other than graph algorithms to verify the benefits claimed in Section 3.1.

Figure 4.3 presents the speedup of our virtual warp-centric implementations. Note that in these graphs the baseline is the original PRAM-style GPU implementation, rather than any CPU version; our interest lies only in the effectiveness of our programming method rather than in the applications themselves.

The first two applications, SSSP (Single Source Shortest Path) and STCON(S-T Connectivity), are two other CUDA graph algorithms from [15, 16]. Although the detailed implementation of these algorithm differ from each other, they share a common algorithmic pattern as in the BFS algorithm to achieve parallelism: each node is visited in parallel to explore its neighbors. Thus analysis similar to the ones in the previous subsections can be made on these results. We used the RMAT graph instance as the input of these algorithms.

The next four applications are selective GPU benchmarks from the literature [8, 12, 20] that have some degree of branch path divergence and scattering memory access issues; however, they do not exhibit workload imbalance as severe as in the graph algorithms. Merge sort [20] recursively merges pairs of sorted sublists in parallel. After each merge operation, the size of sorted sublists doubles and the number of sublist halves. Thus, the application has no workload imbalance (except possibly at the end). However, memory divergence becomes an issue since each thread works on different sublists that are spaced apart. The warp-centric execution method eliminates memory divergence by assigning a pair of sublists to each warp. LU [20] implements a naive version of the LU decomposition. In this implementation, each thread computes one element in a KxK sub-matrix; thus, threads that handle boundary elements incur intra-warp memory divergence. Our programming method avoids this divergence by assigning a row partition to each warp. Backprop [12], a neural network application, performs vector-matrix multiplication in its first kernel. The original GPU implementation uses the entire thread block to compute the partial sum for a sub-matrix which requires column-wise reduction and synchronization. The warp centric method lets each warp independently accumulate partial sums from a chunk of rows, which obviates the synchronization and reduction. NN [8], another neural network application, processes separate data streams in each thread. Although the data streams are consecutive in memory, the start addresses are virtually randomized, causing memory divergence. The warp-centric approach assigns a data stream to each warp, using the threads within a warp to fetch data in parallel.

Finally, Kmeans [12] represents applications where the conventional PRAM-style programming performs well. This application contains very little execution path divergence, and every memory access is well aligned across threads in a warp. Thus, the virtual warp-centric programming method yields only pure overhead: the SISD phase, SIMD width underutilization, as well as increased instruction count. This resulted in 40% overhead with warp-size 32, which has the worst resource utilization. Since our programming method does not require any change to the underlying hardware or compiler system, the user can simply choose not to use our method on non-suitable applications: applications without branch or memory divergence issues. Such applications are relatively easy to identify though code inspection, profiling, or simulation [8].

5. Related Work

There have been numerous implementations of parallel graph algorithms using various computer architectures, including distributed memory supercomputers [25], shared memory supercomputers [5, 17], multi-core SMP machines [6], and GPUs [15, 16]. Our work improves upon previous GPU implementations by several folds, introducing a new programming method that better considers the traits of modern GPU architectures.

Although naive warp-granular task allocation had been used ad hoc in some domains such as sparse-matrix multiplication [9], we proposed a generalized programming method for it using explicit SISD and SIMD phases as well as introduced the new concept of virtual-warp which enables trade-offs between underutilization and divergence.

Our programming method is a low-level construct that resides just on top of the GPU programming APIs. On the other side of the programming model spectrum are high-level approaches that abstract away architectural traits from the user; instead, the compiler/runtime system intelligently generates or selects proper execution codes based on high-level user description, targeting any processor architecture [10, 14]. Thus, these systems are actually complementary to our low-level method; they may decide when a virtual warp-centric method would be beneficial or what virtual warp-size should be utilized based on high-level information such as data access pattern analysis.

Unlike our virtual warp method, which prevents unnecessary divergence without hardware modification, Meng et al. proposed to modify the GPU architecture in order to address these issues [20]. When combined, these approaches complement our virtual warp method by allowing sub-warps to execute independently; thus, there is no need to pay the cost of divergence when using virtual warp sizes.

In this paper, we have not explored the effect of virtual warp-centric programming method with a GPU architecture that includes cache memory, e.g. Nvidia's Fermi [21]. Having cache memory may alleviate memory divergence issues for applications where each thread pursues an independent data stream, since the sequential locality is automatically exploited via cache. However, even with a cache, our warp-centric programming method still provides substantial benefits, since the cache pressure of having data streams per warp is considerably less than that of managing data per thread.

Very recently, Agarwal et al. [3] proposed a scalable BFS algorithm for multi-core CPU. Their optimization technique concurs with our observations in Section 4.2; they saved memory bandwidth by using bitmaps and intentionally avoided coherence traffic as much as possible. Note that there are still rooms to further improve GPU BFS algorithms through algorithmic enhancement. For example, we only applied a frontier-expansion method which maximizes sequential memory access; however it is better to apply

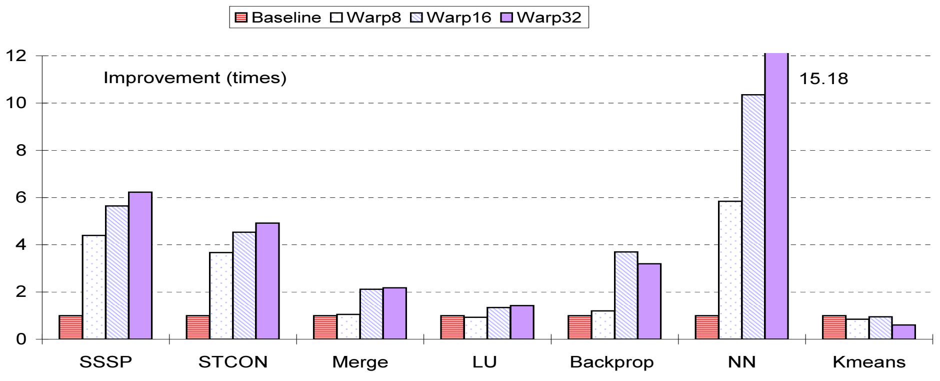


Figure 11. Performance improvement results of GPU applications using warp-centric programming method. Comparison is against original GPU implementation.

a fixed-point method, which manages a work-queue holding next-level nodes, for first and last few stages where only a few nodes are explored. It will also be interesting to apply the recent CPU techniques [3] for GPUs having last-level cache, which does not support coherence at all. We also refer interested readers to other studies on the impact of architectural traits on various applications [19, 21], other than graph algorithms.

6. Conclusion

Parallel execution of graph algorithms on GPUs suffered from workload imbalance for real-world graph instances. To address this issue, we proposed a novel virtual warp-centric programming method, a general strategy that prevents branch divergence and unnecessary scattering memory access. Users can also trade-off between SIMD utilization and branch divergence by varying the virtual warp size.

When applied to graph algorithms, our method resulted in up to 9x speedup over previous GPU implementations. In addition, a set of GPU benchmark applications that suffer from branch divergence and scattering memory accesses was accelerated by 1.3x to 15.1x over the baseline GPU implementations. Finally, we showed that the GPU executions of graph algorithms can outperform implementations on other architectures, mainly because of substantially larger memory bandwidth.

Acknowledgments

This work is supported by DOE contract, Sandia order 942017; Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order 630003198; and Stanford PPL affiliates program, Pervasive Parallelism Lab: NVIDIA, Oracle/Sun, AMD, and NEC.

References

[1] Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`, 2009.

[2] `http://en.wikipedia.org/wiki/GeForce_200_Series`, 2010.

[3] V. Agarwal, F. Petrini, D. Pasetto, and D. Bada. Scalable Graph Exploration on Multicore Processors. In *ACM/IEEE SC*, 2010.

[4] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory bfs algorithms. In *ACM-SIAM SODA*, 2006.

[5] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *IEEE ICPP*, 2006.

[6] D. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IEEE IPDPS*, 2008.

[7] D. A. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite. `http://www.cc.gatech/edu/~kamesh/GTgraph/`, 2006.

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *IEEE ISPASS*, 2009.

[9] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. In *Proc. Conf. Supercomputing (SC'09)*.

[10] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proc. Conf. OOPSLA '10*, 2010.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE IISWC*, 2009.

[13] Cray, Inc. Cray xmt. `http://www.cray.com/products/xmt/`.

[14] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC*, 2006.

[15] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, 2007.

[16] P. harish, V. Vineet, and P. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, India, 2009.

[17] B. Hendrickson and J. Berry. Graph analysis with high-performance computing. *Computing in Science Engineering*, 10(2), march 2008.

[18] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley, 1992.

[19] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ISCA*, 2010.

[20] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.

[21] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2), 2010.

[22] R. Niewiadomski, J. Amaral, and R. Holte. A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations. In *IEEE ICPP*, 2006.

[23] Nvidia. Cuda. `http://www.nvidia.com/cuda/`.

[24] D. J. Watts. *Small Worlds: the dynamics of Networks between Order and Randomness*, chapter 1–2. Princeton University Press, 1999.

[25] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *ACM/IEEE SC*, 2005.