A Compiler for Throughput Optimization of Graph Algorithms on GPUs

Sreepathi Pai Keshav Pingali
The University of Texas at Austin, USA
sreepai@ices.utexas.edu pingali@cs.utexas.edu

Abstract

Writing high-performance GPU implementations of graph algorithms can be challenging. In this paper, we argue that three optimizations called *throughput optimizations* are key to high-performance for this application class. These optimizations describe a large implementation space making it unrealistic for programmers to implement them by hand.

To address this problem, we have implemented these optimizations in a compiler that produces CUDA code from an intermediate-level program representation called IrGL. Compared to state-of-the-art handwritten CUDA implementations of eight graph applications, code generated by the IrGL compiler is up to 5.95x times faster (median 1.4x) for five applications and never more than 30% slower for the others. Throughput optimizations contribute an improvement up to 4.16x (median 1.4x) to the performance of unoptimized IrGL code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors

Keywords Graph applications, amorphous data-parallelism, GPUs, compilers, optimization, throughput

1. Introduction

There is growing interest in raising the abstraction level for GPU programmers through the use of high-level programming notations and compiler technology.

For dense array programs with affine loop nests for example, Baskaran *et al.* have shown that polyhedral compilation techniques can be used to generate high quality CUDA code [6]. The PPCG compiler [59] implements this approach for C programs annotated with directives. DSL compilers can target PPCG's PENCIL intermediate representation [3, 4].

For data-parallel programs, directives-based programming notations like OpenACC [43] and OpenMP [44] permit standard compilers to be used to produce high quality GPU code [32]. Venkat et al. [58] have implemented a compiler to translate data-parallel sparse matrix-vector multiplication (SpMVM) code to CUDA; their compiler also supports data transformations between sparse matrix formats.

For graph algorithms on the other hand, the state of the art of high-level notations and compilation technology is much less advanced. Graph algorithms were first implemented for the GPU by Vineet *et al.* [60]. NVIDIA's Fermi architecture, introduced in 2010, supported efficient fine-grain synchronization for the first time, permitting efficient breadth-first search (BFS) [36] and single-source shortest-path (SSSP) [18] programs to be written in CUDA. Burtscher *et al.* implemented the LonestarGPU benchmark suite, which contains CUDA implementations of complex irregular algorithms such as Delaunay mesh refinement [10, 37, 38]. Since then, C++ template libraries have been developed, such as Gunrock [61], VertexAPI [20], MapGraph [23], TOTEM [24], and Falcon [14]. However, high-performance graph algorithms are still written by hand because existing frameworks cannot match the performance of hand-written code. Furthermore, most frameworks support only vertex programs, a very restricted class of graph algorithms.

In this paper, we argue that three optimizations, which we call *throughput optimizations*, are key to high-performance GPU implementations of graph algorithms. Of course not all graph algorithms benefit from these optimizations, but our study of handwritten CUDA/OpenCL implementations shows that these optimizations are often not being used even for algorithms that do benefit from them. One reason might be that programmers are not aware of these optimizations or may not be sure that these optimizations are useful for their algorithm. Another reason is that these optimizations are difficult to implement, so programmers may be reluctant to make the effort to implement them if the pay-off is unclear.

Obviously, these problems go away if the optimizations can be performed by a compiler while generating CUDA or OpenCL from a high-level notation.

This paper makes the following contributions:

- We identify three *throughput bottlenecks* that may limit the performance of GPU implementations of graph algorithms, and describe three *throughput optimizations* for circumventing these bottlenecks (Section 2).
- We show how these optimizations can be automated within a compiler that translates intermediate-level descriptions of graph algorithms to CUDA code. The intermediate-level notation, called IrGL, is intended to be the target language for front-ends for higher-level notations for graph algorithms such as Green-Marl [29] or GraphLab [33] (Sections 3, 4).
- We evaluate the performance of CUDA code produced by the IrGL compiler for eight graph algorithms: Breadth-First-Search (BFS), Single-Source Shortest Paths (SSSP), Minimum-Spanning Tree (MST), Connected Components (CC), Page Rank (PR), Triangle Counting (TRI), Maximal Independent Set (MIS) and Delaunay Mesh Refinement (DMR) (Section 5).

  IrGL speeds up implementations from 1.24x to 5.95x compared to the fastest GPU codes available, which mostly do not implement these optimizations, as mentioned above.

2. GPU Bottlenecks for Graph Algorithms

The simplest graph algorithms make multiple sweeps over a graph. In each sweep, every node of the graph is visited and an *operator* is applied to the node to update the labels of that node and its neighbors. The Bellman-Ford algorithm for the single-source shortest-path (SSSP) problem is an example; in this algorithm, the operator is the well-known *relaxation operator* [16]. These *topology-driven* algorithms [48] may not be work-efficient because there is usually no work to do at most nodes in a given sweep.

2.1 Data-driven Graph Algorithms

In contrast, *data-driven* algorithms maintain worklists of active nodes in the graph where there is work to be done; in each sweep, only the active nodes are processed, and the worklist for the next sweep is populated. These algorithms terminate when there are no active nodes in the graph. We illustrate the key ideas of data-driven algorithms using breadth-first search (BFS). Initially, the level of each node is set to ∞. The algorithm begins by setting the level of the source node `src` to zero, and initializing the worklist for the first sweep to contain the source node `src`. In each sweep, nodes at level l are on the worklist, and their neighbors in the graph are visited; if the label of a neighbor is ∞, its label is updated to $l+1$ and the node is placed on the worklist for the next sweep. The algorithm terminates when the worklist is empty because all nodes have been labeled .

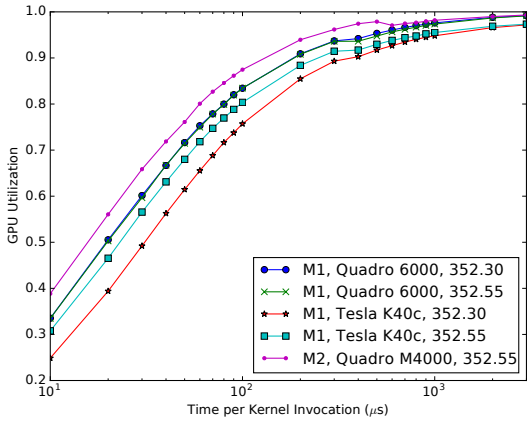


Figure 1. GPU Utilization for a loop containing a fixed-time kernel + `cudaMemcpy` that iterates 10000 times. Kernel time, R, is varied from 10μs to 3000μs across runs, but is fixed during the 10000 iterations. GPU Utilization is defined as $(R * 10000)/Time$ where $Time$ is time taken to execute the loop. Results are across two machines, three graphics card generations, and two device driver versions.

Compared to topology-driven algorithms, data-driven graph algorithms are more difficult to implement efficiently both on CPUs and GPUs. Topology-driven algorithms can be implemented by iterating over the representation of the graph; in contrast, data-driven algorithms require worklists to keep track of active nodes. Implementing worklists efficiently is challenging: because the amount of computation at each active node is relatively small, the overhead of worklist manipulation cannot be amortized as it would be if tasks were more coarse-grained.

In the rest of this section, we describe three performance bottlenecks that must be overcome to match handwritten program performance for data-driven graph algorithms. One of these – kernel launch throughput – is identified for the first time in this work.

2.2 Kernel Launch Throughput

Most graph algorithms are iterative and take the following prototypical form:

```
1  while (cond) {
2    kernel(...);
3    cond = check();   //termination condition
4  }
```

Usually, this loop is executed on the CPU and it performs a kernel launch on each iteration to process the current set of active nodes. The `cond` flag is computed on the GPU and transferred from the GPU to the CPU, usually via a `cudaMemcpy`[1], to tell the CPU whether to execute more iterations or not.

[1] We only focus on NVIDIA GPUs and CUDA in this paper

SSSP	Bellman–Ford		Near–Far	
Input	NY	USA	NY	USA
Iterations	813	5792	1893	49216
Avg. iteration time (µs)	88.17	9168.5	18.5	20
Time (unoptimized, ms)	123	53149	98.8	2536
Time (with DP, ms)	94	53600	92	2448
Time (with outlining, ms)	88.5	60954	41.7	1149

Table 1. Iteration outlining results for two variants of SSSP. The variant/input combinations with low average iteration times benefit (Bellman–Ford on NY and Near–Far on USA and NY). Legend: DP=Dynamic Parallelism

This implementation runs into a serious performance bottleneck. *On all the NVIDIA GPUs we tested, we cannot launch kernels fast enough to keep the GPU fully utilized.*

To investigate this problem, we wrote a GPU utilization microbenchmark that executes a kernel and a `cudaMemcpy` (of 4 bytes) for 10000 iterations. The kernel takes a fixed amount of time every iteration, and this time is varied between 10 µs and 3 ms. Figure 1 shows that if a kernel takes 10 µs or less per call, GPU utilization varies from 25% to 38% depending on the GPU. To achieve 95%–98% utilization, a kernel must execute for at least 1000 µs.

This is a serious problem for work-efficient graph algorithms because their kernels usually execute for less than 100 µs depending on the number of items in the worklist. Since the size of the worklist is dynamic, strategies from dense linear algebra, such as increasing the "block size" cannot be mimicked to increase the work per iteration.

CUDA has a new feature, *dynamic parallelism*, that allows launching kernels directly from the GPU without involving the host CPU. The entire loop above would be executed in a separate GPU kernel of its own without roundtrips to the CPU. However, while dynamic parallelism avoids the overheads of CPU roundtrips, the performance improvements are not commensurate with the level of underutilization observed.

Table 1 shows execution statistics for two SSSP algorithms: Bellman-Ford and Near-Far [18]. Bellman-Ford is a topology-driven algorithm, whereas Near-Far is a data-driven algorithm. This makes a big difference in their work-efficiency: for the USA road network, the time per iteration of Bellman-Ford is more than 9000 μseconds compared to 20 μseconds for Near-Far.

From Figure 1, Near–Far's GPU utilization will be about 50% with an expected gain of 2x at 100% utilization. This gain is realized with our scheme, *Iteration Outlining*, described later in Section 4.1, but not with Dynamic Parallelism. Note that while our optimization benefits Bellman-Ford on the small US road network for NY (average kernel time 88.17 µs), it does not benefit the full US road network since its time per iteration is more than 9000 μseconds and the throughput of kernel launches is not a bottleneck.

```
1  __device__ push(worklist, item) {
2    pos = atomicAdd(worklist.tail, 1);
3    ...
4    worklist.items[pos] = item
5  }
```

Listing 1. Worklist `push` implemented using fine-grain synchronization. Code to check overflow has been elided.

```
1  // Loop 1
2  for(i = 0; i < N; i++)
3     push(WL, a[i]);
4
5
6  // Loop 2
7  for(i = 0; i < N; i++)
8    if(cond(i))
9      push(WL, a[i]);
```

Listing 2. Examples of CUDA kernel loops that push elements of an array `a` onto the worklist without aggregation

Ironically, the throughput limitation on kernel launches is most harmful for *work-efficient* graph algorithms!

2.3 Fine-grained Synchronization Throughput

The next bottleneck arises from the need to maintain worklists of active nodes for data-driven algorithms.

Multiple threads may invoke a worklist's `push` method concurrently, so they must be synchronized properly. A possible implementation of a concurrent `push` method using fine-grained synchronization is shown in Listing 1. The use of CUDA's `atomicAdd` allows us to avoid coarse-grain locks, but it is the slowest kind of atomic on the GPU, executing at 1 per clock cycle [40]. If an item was added to the worklist for each edge traversed, performance on the NVIDIA Kepler would be capped at around 745M traversed edges per second (TEPS). This can be an order of magnitude lower than when atomics are not used. Worklist updates, therefore, form the next prominent bottleneck for work-efficient graph algorithms.

A general scheme to reduce the performance impact of atomics is to aggregate pushes to reduce the number of `atomicAdd`s executed. Consider the first loop in Listing 2, which is executed on a single thread. Assuming a function `agg_thread` that aggregates pushes within a thread is available, the number of atomics for this loop can be reduced to 1 by the following code:

```
1  start = agg_thread(N);
2  for(i = 0; i < N; i++)
3    do_push(start+i, a[i]);
```

Here, `agg_thread` uses a single atomic to reserve space in the worklist for N items, while `do_push` directly writes to the reserved location corresponding to the iteration, without performing any atomics.

Using `agg_thread` requires that the number of pushes can be determined before entering the loop. The second loop

in Listing 2 illustrates a case where this is not possible – the `push` is conditionally executed. While `agg_thread` cannot be used here, it is still possible to aggregate *across* threads, specifically those in the same CUDA warp. A CUDA warp consists of 32 threads that execute in lockstep and so the `agg_warp` function below can use warp-voting functions to determine which other threads in its warp will be executing `push`. Then one active thread in the warp executes the single atomic to reserve space for all the voters. Warp-aggregation reduce the number of atomics by 31 in the best case. After aggregation, the loop now looks like:

```
1  for(i = 0; i < N; i++)
2    if(cond(i)) {
3      start = agg_warp(1);
4      do_push(start, a[i]);
5    }
```

In fact, it is possible to aggregate pushes across all threads in the same thread block. Such an `agg_threadblock` has the potential to reduce the number of atomics to one per thread block for each push.

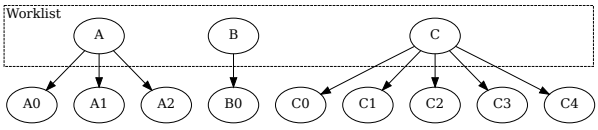
However, because it involves intra-thread-block communication, `agg_threadblock` must contain a CUDA barrier. Since CUDA barriers must be executed unconditionally by all threads in the thread block for correct execution, `agg_threadblock` *cannot* replace `agg_warp` in the example above. However, it may be able to replace `agg_thread` in the first loop if it can be determined that the `agg_thread` will be unconditionally executed by every thread.

Since CUDA kernels can deadlock if barriers are used incorrectly [9, 41], aggregation by hand is error-prone and usually not performed by programmers. Ideally, programmers would like to only use `push` in their code. The compiler would then analyze this code and perform the aggregation automatically. In Section 4.2, we describe the analyses and transformations required to achieve this *cooperative conversion*. In Section 3.4, we demonstrate how cooperative conversion leads to 3.25x improvement in performance for a loop similar to the first loop in Listing 2.

2.4 Graph Traversal Throughput

A common pattern in graph algorithms is a pair of nested loops (Figure 2) in which the outer loop iterates over a set of nodes, while the inner loop iterates over the edges connected to a given node. In nearly all benchmarks, the two nested loops are DOALL-style loops with no dependences between iterations.

To execute such loops on the GPU, loop iterations must be mapped to CUDA threads. Unless this is done carefully, the *throughput of loop iteration execution* becomes a bottleneck. If iterations of the outer loop are assigned to threads, all iterations of the inner loop for a given outer loop iteration are performed on the same thread. *This serialization can become a performance bottleneck when traversing graphs with highly-skewed degree distributions such as power-law graphs.*




```
for(node In worklist)
  for(edge In node.edges)
    ...
```

Figure 2. Graph traversal code. The outer `for` iterates over the worklist, while the inner `for` iterates over the edges of a node. Note that trip count for the inner `for` is different for each node.

Policy	BFS	SSSP-NF	Triangle
Serial	1.00	1.00	1.00
TB	0.25	0.33	0.46
Warp	0.86	1.42	1.52
Finegrained (FG)	0.64	0.72	0.87
TB+Warp	1.05	1.40	1.51
TB+FG	1.10	1.46	**1.55**
WP+FG	1.14	1.56	1.23
TB+Warp+FG	**1.15**	**1.60**	1.24

Table 2. Speedup relative to serial inner-loop execution for different combinations of scheduling policies on the RMAT22 input. Higher is better. Legend: TB=Thread Block. The FG policy dynamically assigns inner-loop iterations to consecutive threads (Section 4.3).

For regular programs, the two `for` loops can be coalesced by the compiler, but here the inner loop is non-affine, so a polyhedral compiler like PPCG [59] cannot coalesce these loops. Instead, dynamic scheduling policies must be used.

A number of dynamic scheduling policies have been studied in previous work on BFS implementations on GPUs [36] (which supersedes [28]). Iterations from the inner loop can be distributed to all threads in the thread block (Thread Block/TB), or only to warps (Warp) or may use a dynamic proportion of the threads (Finegrained/FG). Usually, the degree of the node (i.e. the trip count of the inner loop) is used to dynamically decide which scheduler is actually used for a particular outer loop iteration. High-degree nodes are usually handled by Thread Block, medium-degree by Warp and low-degree by Finegrained. Applying these policies to other applications, however, reveals that no single policy or fixed combination of policies is beneficial for all applications.

Table 2 shows the performance of BFS, SSSP Near–Far and PageRank. The results show that the best policy is different for different programs. Since there are 8 different combinations of scheduling policies that are possible, a compiler that can implement any of these combinations is required. In Section 4.3, we describe how our compiler can generate any combination of scheduling policies by using a common abstraction for scheduling policies.

```
1  Kernel BFS(graph, LEVEL) {
2    ForAll(wlidx In WL) {
3      n = WL.pop(wlidx)
4      edges = graph.nodes(n).edges
5      ForAll(e In edges) {
6        if(e.dst.level == INF) {
7          e.dst.level = LEVEL;
8          WL.push(e.dst.id);
9  } } } } }
10
11 {src.level = 0;
12 LEVEL=1;
13 Iterate BFS(graph, LEVEL) Initial [src] {
14   LEVEL++;
15  }
16 }
```

Listing 3. Level-by-level BFS kernel in IrGL. The worklist object (WL) is an implicit parameter.

2.5 Discussion

The three throughput limitations we have identified are fundamental to the performance of data-driven graph algorithms. The launch throughput limitation prevents the GPU from being fully utilized. Worklist-using algorithms are usually work-efficient, but the performance of atomics on GPUs is so low that programmers may opt to use topology-driven algorithms, which are not work-efficient. Parallel graph traversal with multiple scheduling strategies can improve performance across different algorithms.

These bottlenecks are orthogonal to each other, but there are significant second-order performance effects from combining them. Clearly, if launch throughput is not optimized, any improvement that lowers kernel execution time below $1\,\mathrm{ms}$ under-utilizes the GPU and does not result in improvements in overall performance. Aggregating atomics is important, but opportunities to do so are few in most programmer-written code. By optimizing graph traversal, we find more aggregation opportunities for cooperative conversion. This means that the combinatorial space of valid optimizations is quite large (> 64), making it infeasible for manual exploration. A compiler-based approach is required to generate these combinations automatically to select the best performing implementation.

3. IrGL Compiler Overview

This section describes IrGL, the intermediate-level notation used in this paper to express graph algorithms, and gives an overview of how IrGL programs are compiled to CUDA.

3.1 Graph Algorithms and Worklists

Since high-performance graph algorithms are organized around worklists, the performance of these algorithms can depend critically on how well worklists are created and iterated over. To expose this to the compiler for optimization, IrGL gives special treatment to worklists.

Listing 3 shows the IrGL code for data-driven BFS (initialization code is not shown). In the BFS kernel, the outer ForAll iterates over valid indices in the worklist `WL` which are used by `WL.pop` to obtain the actual nodes, and the inner ForAll iterates over the outgoing edges connected to a given worklist node `n`. As in other GPU notations, the ForAll construct tells the compiler that the iterations of the loop can be executed in parallel.

Invocations of the BFS kernel are performed by the Iterate construct (line 13). This construct implicitly passes a worklist to each kernel call; for the first call, a fresh worklist is created by the Initial construct, while successive calls are passed the worklist returned by the previous call. Kernel invocations terminate when the worklist returned by the kernel invocation is empty. Note that the dataflow of worklists between kernel calls is specified implicitly in IrGL through the Iterate construct.

3.2 IrGL Constructs

Table 3 shows a high-level summary of IrGL constructs. It is convenient to divide them into *kernel constructs*, which are used to write code that will be executed on the GPU, and *orchestration constructs*, which setup and manage worklists for the kernels. Coupled with a runtime library of parallel data structures such as graphs and multisets, these constructs allow a wide range of graph algorithms to be specified.

Orchestration constructs: IrGL provides a default worklist to every kernel. A kernel may `pop` work off the worklist, usually by iterating over it using ForAll, and it may `push` values onto another worklist to enqueue work. IrGL worklists are bulk-synchronous so work items pushed during an invocation cannot be popped in the same invocation.

Worklists are created and managed by Iterate and Pipe constructs. Iterate repeats invocations until a worklist is empty as shown in the BFS code in Listing 3. The Pipe statement orchestrates the invocation of a pipeline of kernels each of which produces a worklist for the next kernel. A Pipe Once executes once whereas a Pipe loops until the worklist is empty. Listing 4 illustrates the code for SSSP NearFar, a Δ-stepping based SSSP algorithm for the GPU [18]. The inner Pipe loops over the two Invoke calls, the first of which is implicitly and automatically looped due to the Respawn in the `sssp_nf` kernel.

We add Iterate and Pipe to IrGL for several reasons. First, most graph algorithms are data-driven and use worklists to keep track of work, so these constructs capture common patterns of worklist manipulation. Second, Iterate and Pipe significantly reduce the complexity of implementing the iteration outliner optimization (Section 4.1).

Since the GPU memory is distinct from CPU memory, the compiler and runtime transfer data between the CPU and GPU as needed. Redundant transfers are avoided by adapting the AMM runtime coherence scheme [46] and no manual memory transfers are needed.

Construct	Semantics
Kernel Constructs	
ForAll (*iterator*) { *stmts* }	Traverse *iterator* in parallel executing *stmts*
ReduceAndReturn (*bool-expr*)	Reduce values of *bool-expr* and return as kernel return value. The actual reduction, one of Any or All, is specified at kernel invocation.
Atomic (*lock-expr*) { *locked-stmts* } [Else { *failed-stmts* }]	Acquire *lock-expr* and execute *locked-stmts*. If an Else block provided, execute *failed-stmts* if *lock-expr* was not acquired. Implements divergence-free blocking locks [52].
Exclusive (*object, elements*) { *locked-stmts* } [Else { *failed-stmts* }]	Try once to acquire locks for *elements* in *object* and execute *locked-stmts* on succeeding. On failure execute *failed-stmts* if provided otherwise execute next statement. One thread is guaranteed to execute *locked-stmts* on conflicts.
SyncRunningThreads	Compiler-supported safe implementation of GPU-wide global barriers [64]
Retry *item* (*or* Respawn *item*)	Push *item* into a retry worklist and re-execute the kernel. Use of Retry indicates a runtime conflict and triggers conflict management in the runtime (e.g. serial execution).
Orchestration Constructs	
[Any | All (] Invoke *kernel*(*args*) [)]	Invoke *kernel*, passing current worklists if kernel uses them.
Iterate [While | Until Any | All] *kernel*(*args*) [Initial (*init-iter-expr*)]	Iteratively invoke *kernel* until termination condition is met or worklist is depleted. If invoked standalone, establish fresh worklists using *init-iter-expr* for initialization, else pass current worklists. Iterate can be emulated by a looped Invoke or Pipe and is provided for convenience.
Pipe [Once] { *stmts* }	Establish worklists to be used by *stmts*. Without Once, repeat *stmts* until worklists are empty. Nested Pipes will not establish worklists.

Table 3. Summary of IrGL Statements, [] indicate optional parts, | indicates options. See [45] for syntax in ASDL.

```
1  Kernel sssp_nf(...) {
2      ...
3      if(dst.distance <= delta)
4          Respawn(dst)    // near node
5      else
6          WL.push(dst)    // far node
7      ...
8  }
9
10 // establishes worklist shared by
11 // initialize, sssp_nf and remove_dups
12 Pipe Once {
13   // also places src into the worklist
14   Invoke initialize(graph, src)
15
16   // looping Pipe
17   Pipe {
18     // Invoke sssp_nf will automatically loop
19     Invoke sssp_nf(graph, curdelta);
20     Invoke remove_dups();
21     curdelta += DELTA;
22   } }
```

Listing 4. SSSP Near–Far using Pipe

Kernel constructs: These are used to write code that will execute on the GPU, and they are described briefly because they are not the focus of this paper. Parallelism is exposed through a top-level ForAll, which iterates over a worklist (*e.g.*, only nodes in the BFS frontier) or a data structure (*e.g.*, all triangles in a mesh). ForAll loops can be nested. The compiler manages the assignment of ForAll iterations to CUDA threads but this is not necessarily fixed at compile-time (Section 4.3). We provide high-level synchronization constructs Atomic, Exclusive and SyncRunningThreads because they are particularly hard to get right [2, 52].

The computational core of a kernel, such as the conditional update of a neighbor's label in BFS, is written in CUDA and is parsed in a limited fashion to obtain control and data flow information. Since our compiler has full discretion on scheduling the iterations of ForAll loops, some low-level features of CUDA such as shared memory are not supported in their full generality and may not be used when writing operator code. The only allowed use of GPU shared memory in operator code is as a software-managed cache (i.e. communication is not supported). Other CUDA features not supported in IrGL programs include referencing the indexing variables (e.g. `threadIdx`, `blockIdx`), the warp shuffle and vote functions. While the operator code cannot make use of these CUDA features, the underlying libraries and the generated code make extensive use of them.

3.3 IrGL AST Compiler

The overall structure of our compiler is given in Figure 3. The compiler operates on an AST of an IrGL program. Our compiler parses this code for well-formedness and for dataflow information, but is limited to C99 syntax. The compiler generates CUDA output, targeting NVIDIA GPUs from Kepler onwards. Preliminary front-ends exist to generate these ASTs from native C++ and Python programs. For lack of space, we defer discussing details of lowering the IrGL AST to CUDA to an associated technical report [45], since this lowering is not the focus of this paper.

	CUDA	OpenACC [43]	PPCG+PENCIL [59]+[3]	OpenMP 4.0 [44]	FALCON [14]	Whippletree [56]	IrGL
Approach	Language	Directive	Directive+IR	Directive	DSL	Runtime	IR
Constructs							
ForAll	N	Y	Y	Y	Y	N	Y
Atomic	primitives	primitives	N	Y (`critical`)	non-blocking	N	Y
Exclusive	N	disallowed	N	Y (`critical`)	Y (`single`)	N	Y
Worklists	N	N	N	N	Y	Y	Y
Optimizations							
Iteration Outlining	N	N	N	N	N	N	Y
Cooperative Conversion	N	N	N	N	N	manual	Y
Parallel Nested ForAll	N	N	only affine loops	Y	unsupported	dynamic tasks	Y
Graph Traversal Time	13ms	13ms	13ms	63ms	–	13ms	2ms

Table 4. Comparison of IrGL to other compiler-based GPU programming models. Execution times are for the graph traversal benchmark shown in Listing 5 on the RMAT22 graph.

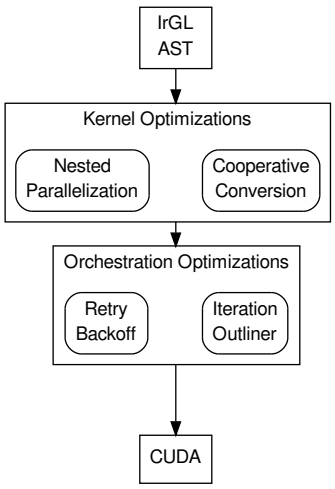


Figure 3. IrGL Compilation and Optimizations Overview.

```
1  for(n = 0; n < graph.nodes; n++) {
2    csr_edge_begin = graph.row_start[n];
3    csr_edge_end = graph.row_start[n+1];
4
5    for(e = csr_edge_begin; e < csr_edge_end; e++) {
6      pos = atomicAdd(worklist.tail, 1);
7      worklist.data[pos] = graph.edge_dst[e];
8    }
9  }
```

Listing 5. Graph Traversal Microbenchmark. The graph is stored in compressed sparse row (CSR) representation. All iterations of both `for` loops are independent but note that the inner `for` loop is not affine. Its iteration count is the degree of the node being traversed.

3.4 Comparison with other GPU Notations

Since IrGL programs are parallel programs, some kernel constructs obviously overlap with constructs in frameworks such as OpenMP or OpenACC. The key differences are in the orchestration constructs and in the throughput optimizations that our compiler performs.

Table 4 compares IrGL constructs with constructs in other GPU notations. Whippletree [56] is an ingenious C++ template-based runtime that performs dynamic task scheduling on the GPU and is a representative of high-performance runtimes. The FALCON compiler [14] is not publicly available. OpenMP supports execution of nested parallel loops but it was too slow and not used here.

Listing 5 shows a simple graph traversal microbenchmark that we implemented in all of the frameworks for which we had compilers available. Table 4 shows the running times on these frameworks: the throughput optimizations we describe in this paper enable the IrGL compiler to *automatically* improve performance by $6.5\times$ compared to existing frameworks. Like others, the IrGL program starts out at 13ms bottlenecked by atomics, it improves to 4ms after Cooperative Conversion (Section 4.2) and to 2ms after removing excessive serialization (Section 4.3).

4. IrGL Optimizations

This section describes the three key throughput optimizations: iteration outlining to overcome the kernel launch bottleneck (Section 4.1), cooperative conversion to overcome the atomic execution bottleneck (Section 4.2), and parallel execution of nested ForAll loops to overcome the graph traversal bottleneck (Section 4.3).

4.1 Iteration Outlining

The primary bottleneck for iterative graph algorithms is the limited rate of kernel launches. Any other optimization that speeds up a GPU kernel will either appear to have no effect (being masked by this limit) or will cause the program to run into this limit. Therefore, we must first tackle this bottleneck to improve the performance of graph algorithms.

Iteration outlining is the program optimization for addressing this problem. The basic idea involves moving the iterative loop from the CPU to the GPU. We first decide applicability of this optimization. Conveniently, the iterative nature of these algorithms is captured well by Pipe and It-

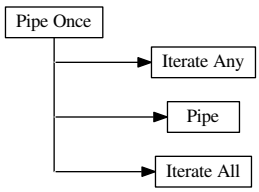


Figure 4. Example Outline Tree used to determine applicability and scope of iteration outlining

```
1  __global__ BFS(graph, level, worklist) { ... }
2  // CPU
3  while(worklist.nitems()) { ❶
4    BFS<<<...>>>(graph, worklist); ❷
5    level++;
6    ... // worklist management elided
7  }
```

Listing 6. CUDA code for BFS (Listing 3) without iteration outlining. ❶ is a `cudaMemcpy` and ❷ is a repeated kernel launch.

erate statements. Not all Pipe and Iterate statements can be outlined. Apart from invoking IrGL kernels, Iterate or Pipe can contain arbitrary code (e.g. Line 14 in Listing 3). If this arbitrary code involves calls to library code (e.g. I/O code) which cannot be executed on the GPU or reads/writes CPU memory via pointers, it cannot be outlined.

To maximize the opportunity for outlining, the IrGL compiler constructs a tree starting from each outermost Pipe and Iterate statement that it encounters. The nodes of the tree correspond to Pipe and Iterate nested within that root Pipe, and their depth in this tree corresponds to the depth at which they are nested in the original program. Figure 4 is an example tree resulting from this analysis.

A Pipe or Iterate node in this tree can be outlined if it meets two conditions: (i) it contains no arbitrary code that prevents outlining, and (ii) all of its children can be outlined. Assuming that all nodes in the tree of Figure 4 can be outlined, the compiler can outline the root Pipe Once resulting in one outline or it can outline the nested nodes separately resulting in up to three outlines. On the other hand, if say, the nested Pipe node could not be outlined, then only Iterate Any and Iterate All nodes can be outlined.

The choice of which node to outline is left to an autotuner. Indeed, the determination of which node should be outlined for best performance cannot be done statically without additional information. Once a node has been identified, the remaining transformations are straightforward.

Using the code of BFS in Listing 3 as an example for the optimization, we begin by noting that there is only one Iterate statement. Listing 6 and 7 show the CUDA code generated by the IrGL compiler without outlining and after outlining the Iterate statement respectively. The following steps implement the transformations required to outline a selected Pipe or Iterate:

```
1  __device__ BFS(graph, level, worklist) { ... } ❶
2
3  __global__ BFS_control(graph, glevel, worklist) ❷
4  {
5    level = *glevel;
6    while(worklist.nitems()) { ❸
7      BFS(graph, level, worklist); ❹
8      SyncRunningThreads(); ❺
9      level++
10     ...
11   }
12   *glevel = level;
13 }
14
15 // CPU
16 cudaMemcpy(gpu_level, level, ...);
17 BFS_control<<<...>>>(...); ❻
18 cudaMemcpy(level, gpu_level, ...);
```

Listing 7. CUDA code for BFS with iteration outlining. ❶ is the device kernel, ❷ is the outlined control kernel, ❸ is now an ordinary memory read, ❹ is now an ordinary function call, ❺ is required to prevent race conditions, ❻ is a single launch from the CPU.

- In the first step, the compiler "outlines" the selected Iterate or Pipe statement entirely to a separate *control* kernel on the GPU (Listing 7: ❷ and ❸). This allows it to eliminate memory copies of the condition variables since the control kernel and the iterated kernels are in the same address space.
- In the second step, the compiler generates GPU-callable functions for all kernels invoked from the Pipe construct (i.e. CUDA `__device__` functions) (Listing 7, ❶). The control kernel calls these device functions directly as ordinary functions (❹). To maintain launch semantics, each of these function calls is followed by a GPU-wide barrier (❺).
- In the final step, the compiler replaces the CPU-side loop with a call to the control kernel (❻) and code to copy local variables to the GPU and back.

After the outlining, all launches have been eliminated by converting them into function calls thus overcoming the launch throughput limitation and fully utilizing the GPU.

4.2 Cooperative Conversion

Our second compiler optimization, cooperative conversion, reduces the number of atomics by aggregating functions that use atomics. Recall that this involves replacing a function like `push` used in the code with its aggregate equivalent. This equivalent may perform aggregation at the thread or warp or thread-block level. To perform thread-block-level aggregation, the compiler must analyze the code to identify points in the code where barriers may be safely placed.

4.2.1 Aggregation

To build a general compiler aggregation optimization that works with many data structures, our compiler is agnostic

```
1  class WL {
2    // default single-thread method
3    void push(T item);
4
5    // the setup methods' return value is used
6    // as the ''start'' parameter to do_push
7    @thread(push)
8    int agg_thread_push(int items);
9
10   @warp(push)
11   int agg_warp_push(int items);
12
13   @threadblock(push)
14   int agg_tb_push(int items);
15
16   @task(push)
17   void do_push(int start, int taskid, T item);
18 };
```

Listing 8. Cooperative method annotations for WL.push()

to the actual aggregation mechanism used by functions like `agg_thread`.[2] Instead, functions that can be aggregated and their aggregate equivalents are identified by annotations in the data-structure library. Note that these annotations are completely transparent to the IrGL programmer.

Listing 8 illustrates the annotations for the `push()` method of a WL (worklist) class. The `WL.push()` method is identical to `push` method described earlier and uses fine-grained atomics. The `@thread`, `@warp` and `@threadblock` annotations identify the aggregation functions corresponding to the three levels of aggregation supported by our compiler: thread, warp and thread block. The `@task` annotation identifies `do_push` as the function that replaces the `push` after aggregation. This function is the same regardless of the level of aggregation.

All aggregation functions take an argument identifying the count of items aggregated and return a `start` value passed to the task function. The task function takes two additional arguments compared to the function it is replacing: the `start` value from the aggregation function and `taskid` indicating the index of the aggregated task. For example, if N was passed to the aggregation function, then `taskid` would take on values from 0 to $N - 1$.

Now, to perform thread-level aggregation, we only need to check that the loop has fixed trip count and that the loop cannot exit early. The corresponding `@thread` function is inserted into the code before the loop and the `@task` replaces the original function:

```
1  start = @thread((END - START)/STEP);
2  for(i = START; i < END; i+=STEP) {
3    // originally: wl.push(x)
4    @task(start, i - START, x);
5  }
```

[2] The aggregation mechanism is nearly always the prefix-scan primitive [53]. See [19, 36] for implementations.

Warp-level aggregation is simpler in that it can be performed unconditionally. The original function is replaced by an invocation to `@warp` and `@task` functions respectively.

At this point, we can consider "upgrading" a `@thread` aggregation function to its `@warp` or to `@threadblock` equivalent. The conversion to a `@warp` equivalent is unconditional, but conversion to `@threadblock` is not so straightforward.

Since a `@threadblock` function is assumed to contain barriers, it must be unconditionally executed by every thread in the thread block. Therefore, it must be placed at a point that is always executed by each thread. Since all threads in the scope of the barrier must pass simultaneously through such a point, we term these points *focal points*, an allusion to the optical focus through which all rays of light pass. The `@threadblock` aggregation function may then be placed at any such focal point, provided the placement preserves any data and control dependences of the original `@thread` or `@warp` function. We present a simple analysis in next section to identify such points.

4.2.2 Focal Point Analysis

Focal points are nodes in a program's control flow graph (CFG) that are guaranteed to be visited by every thread. It is always correct to place a barrier or a method containing a barrier at these points. For example, the dominators[3] of the $EXIT$ node in a CFG are all focal points. However, this set is usually too small to allow cooperative conversion.

The set of focal points can be enlarged by using the notion of a *uniform* branch. A branch is a CFG node with more than one outgoing edge. A uniform branch is one that is decided the same way by all threads in scope of the barrier that reach that branch; otherwise, the branch is said to be non-uniform[4]. If the branch is a If or a While statement, for example, all threads in scope of the barrier decide the condition the same way – they all take the true branch or all take the false branch. If it is a For loop, then all threads in scope have the same trip count.

Uniform branches may be identified by static analysis [5, 15, 31], annotation or by construction. In this work, we *construct* uniform branches when generating code. For example, uniform ForAll loops can be generated by changing the loop trip count and introducing guards. Uniform-by-construction allows us to sidestep some limitations of static analysis wherein the use of thread IDs in the initialization of our loops (e.g. the compiled outer ForAll loop) would prevent them from being recognized as uniform branches.

Now, a barrier must be control-dependent on a uniform branch to be placed correctly, and it can be placed on either side of such a branch. However, uniformity is a *local* property of a branch. In particular, it is possible in a nested If-Then-Else for the predicate of the outer conditional to be

[3] If every path from $ENTRY$ to a node N in the CFG passes through a node D, then D dominates N.

[4] It is helpful to note that CUDA and IrGL are SPMD programming models.

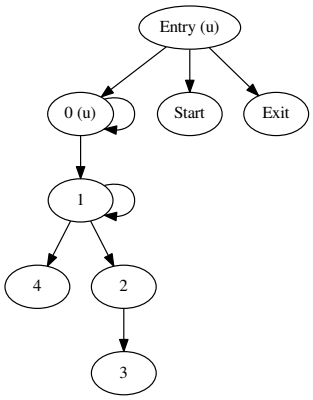
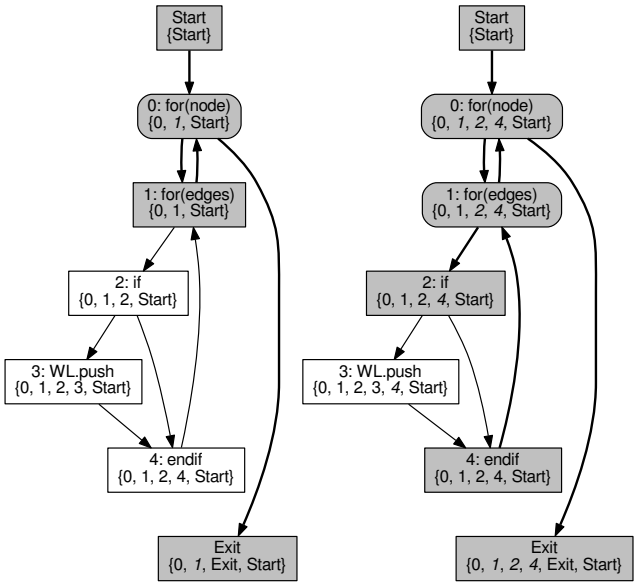


Figure 5. Control Dependence Graph (CDG) of the CFG in Figure 6(a). Nodes marked "(u)" are uniform branches.



(a) Before (b) After

Figure 6. Results of Focal Point Analysis before and after the inner loop is made uniform. Rounded boxes indicate uniform nodes. Highlighted nodes are focal points. Thread-block-level aggregation can be placed on the thickened edges. Differences from dominator analysis are italicized.

non-uniform and for the predicate of the inner conditional to be uniform; in that case, not every thread will execute the inner conditional, but threads that do will either all take the true branch or all take the false branch. Similarly, it is possible for a For loop to be uniform even if there is an early exit out of the loop that is taken by some threads but not others. Therefore, placing a barrier under a uniform branch is not sufficient; all branches (if any) leading up to the barrier must be uniform too. Thus, *a focal point in the CFG is a node that is iteratively control-dependent only on uniform branches, other than possibly itself.*

The set of focal points in a CFG can be determined by examining its Control Dependence Graph (CDG) [22, 47]. Figure 5 shows the control dependence graph (CDG) of the CFG in Figure 6(a). Given that node `0` is a uniform branch and that nodes `{1,2}` are non-uniform, it can be seen that the set of focal points is `{Start,0,1,Exit}`. Nodes `{2,3,4}` are not focal points since they iteratively depend on non-uniform node `1`. By enlarging the set of uniform branches to `{0,1}` corresponding to the CFG in Figure 6(b), the set of focal points is the set `{START,0,1,2,4,EXIT}`.

To illustrate both the analysis and transformations described so far, let us consider aggregating the `WL.push` in BFS of Listing 3 to its thread block version. There are two complications: the `WL.push` is definitely under a non-uniform `if` conditional and the iteration over edges is also non-uniform. As a result, initially, the only uniform branch is the outermost ForAll. Figure 6(a) shows the only focal points available. The `WL.agg_tb_push` cannot be placed on these focal points since it would violate data dependences. Therefore, we transform the inner ForAll to be uniform. This provides additional focal points within the inner ForAll loop (Figure 6(b)) that can now be leveraged to complete the transformation of `WL.push`. Listing 9 is the final result. Highlighted are: ❶ inner loop with a uniform loop trip count, ❷ guard to restrict execution to original loop iterations, ❸ capturing a valid push by the current thread, ❹ unconditional execution of `WL.agg_tb_push` by all threads in the thread block, ❺ the actual aggregated push executed conditionally.

Of these, ❶, the conversion of the inner loop to a uniform loop needs further explanation. To make the inner loop uniform, one correct but terribly inefficient method (that we do *not* use) is to set the trip count of the inner loop to the maximum trip count among all the threads in the same thread block. The determination of this value can take place before the inner ForAll loop. In our actual system, however, we replace the inner loop by the scheduling loops generated by the nested parallelism optimization described in the next section which are guaranteed to be uniform by construction.

4.3 Parallel Execution of Nested ForAll loops

Our compiler generates code to perform dynamic scheduling for parallel execution of the iterations of nested ForAll loops. Broadly, this is inspector–executor execution of irregular parallel loops. But as shown in Table 2, no single policy for executors is optimal for all programs or inputs. There is also no fixed combination of scheduling policies that is optimal. Therefore, to be truly useful, our compiler must be able to generate any combination of scheduling policies. We now describe how our compiler can generate *any* combination of schedulers.

```
1  for(node in worklist) {
2    for(e in node.edges) {  // uniform ❶
3      bool valid_push = false;
4      int to_push;
5      int start;
6
7      if(valid_iteration()) { ❷
8        if(e.dst.level == UNVISITED) {
9          e.dst.level = LEVEL;
10
11         valid_push = true;  ❸
12         to_push = e.dst;
13       } }
14     start = WL.agg_tb_push(valid_push); ❹
15     if(valid_push)
16       wl.do_push(start, 0, to_push);    ❺
17 } }
```

Listing 9. Cooperative conversion of the BFS operator code to use the thread block aggregate for `WL.push`

Policy	Iterations Executed By	Pref. Loop Size
Serial	Single thread	$>= 0$
Threadblock	All threads in thread block	$>= TBSIZE$
Warp [28, 36]	All threads in a warp	$>= 32$
Finegrained [36]	Consecutive threads in thread block	$>= 0$

Table 5. Scheduling policies for distributing inner loop iterations to threads. $TBSIZE$ is size of the CUDA threadblock (usually 256).

4.3.1 Background on Scheduling Policies

Table 5 describes the basic scheduling policies known to our compiler. Serial, Threadblock and Warp are straightforward policies, where a fixed number of threads are assigned to execute iterations of the inner loop in parallel. The Serial policy serializes the inner loop but executes the outer loop in parallel. The Threadblock policy parallelizes the inner loop but serializes the outer loop. The Warp policy assigns outer loop iterations to CUDA warps. Each warp parallelizes inner loop iterations across its 32 threads while serializing the outer loop iterations assigned to it. However, warps execute in parallel, so outer loop iterations assigned to different warps execute in parallel.

Assigning a fixed number of threads to execute inner loop iterations (as in Warp and Threadblock) leads to underutilization of threads for irregular (or low trip count) inner loops. The Finegrained policy, therefore, assigns consecutive inner loop iterations to consecutive threads of a thread block. After all inner loop iterations have been assigned, the scheduler switches to assigning iterations from a different parallel outer loop iteration to the unused threads. Thus, it can reduce underutilization with parallel execution of both inner and outer loops.

4.3.2 Combining Schedulers

While each of the policies described in the previous section can be used individually, it is better to combine them and assign each policy a partition of inner loops. This partitioning can take a scheduler's preferred loop size (Table 5) into account. In a Threadblock+Warp+Finegrained combination, for example, all inner loops with trip count of TBSIZE or more are executed by Threadblock. Then, loops of size > 32 are executed by the Warp policy with the Finegrained policy finally executing any remaining inner loop iterations. Of course, if the Finegrained policy was absent in this combination, the Warp policy would execute all iterations not assigned to Threadblock ignoring its preferred loop size.

To generate code for a given combination of scheduling policies, our compiler begins by arranging the user-selected basic policies in descending order of their preferred loop sizes. Schedulers for each scheduling policy are then emitted in this order. Listing 10 shows the generic template for combining schedulers. A scheduler is omitted if the policy it implements was not selected as part of the combination.

In Listing 10, *Scheduler initialization* code contains, for example, the calculation for assignment of inner loop iterations to threads for the Finegrained scheduler. Next, if the inner loop iterations contain (read-only) references to outer loop variables, the compiler computes the closure and generates code (*Closure-saving code*) to save these data values to CUDA shared memory. If our compiler is unable to parse operator code, annotations for read/write sets can be supplied to assist the closure computation. Schedulers for each selected policy follow, separated by a CUDA `__syncthreads()`, since they read and write from the same locations in CUDA shared memory.

Each individual scheduler (Listing 11) is a uniform and regular loop that distributes inner loop iterations to threads. First, the threads allocated to a scheduler instance identify which outer loop iteration to execute. For Threadblock and Warp, this is one outer loop iteration from the partitions assigned to them. Identification uses communication between the threads. Finegrained, on the other hand, reads from the calculated assignment.

Note that both the outer ForAll loop and the `while(true)` loop of the scheduler are uniform. Replacing the non-uniform inner irregular ForAll loop with these uniform scheduler loops allows promotion of `@thread`- and `@warp`-level to `@threadblock`-level aggregation. This aggregate function is executed before any scheduler and reserves space in the worklist for *all* inner loop iterations. The resulting `start` values are essentially references to outer loop variables and are treated similarly.

An IrGL programmer is therefore able to use our compiler to quickly and automatically evaluate all combinations of traversal strategies for a graph algorithm. Performance can

```
1  for(wlidx = tid; ...) { // uniform
2    node = WL.pop(wlidx);
3
4    Scheduler initialization code
5    Closure-saving code
6    __syncthreads();
7
8    Threadblock scheduler code
9    __syncthreads();
10
11   Warp scheduler code
12   __syncthreads();
13
14   Finegrained scheduler code
15   __syncthreads();
16 }
```

Listing 10. Generic CUDA template for inner loop parallelization using a combination of schedulers

```
1  while(true) {
2    identify node(s) to work on      // Inspector
3    __syncthreads();   // not needed for Warp
4    if( no work was available )
5      break;
6
7    // Executor
8    restore inner loop closure for selected nodes
9
10   execute inner loop in parallel
11   __syncthreads();
12 }
```

Listing 11. Generic CUDA scheduler template

benefit in two ways: i) by a faster traversal strategy, and ii) by exposing more opportunities for cooperative conversion.

4.4 Other Minor Optimizations

Our compiler also extends some well-known optimizations to IrGL constructs. In particular, our compiler can unroll Pipe constructs. This often triggers the NVIDIA CUDA compiler to perform inlining in the control kernels used in iteration outlining.

Our runtime library's implementations of data structures such as Graphs and Worklists contain variants that use the CUDA texture unit. A programmer can explore these data structure variants simply through a compiler switch.

For programs that use mutual exclusion constructs and worklists, we also provide the *Retry Backoff* optimization described next.

Retry Backoff The Retry statement populates a retry worklist, often as a result of failed Atomic or Exclusive. This worklist therefore exhibits *conflict locality*, items close to each other in this worklist have a higher probability of conflicts. To reduce conflicts, therefore, we assign consecutive items in the retry worklist to the same thread. By distributing items in the worklist in this blocked fashion to threads, we can reduce the number of retries needed. Once the number of workitems in the retry list falls below a certain number, they are processed sequentially. Currently, Retry Backoff is only applied to DMR which significantly benefits from it.

Road networks
NY (264K, 730K), FLA (1.07M, 5.4M), CAL (1.8M, 4M), USA (23M, 57M)
RMAT-style [11]
rmat16 (66K, 504K), rmat20 (1M, 8M), rmat22 (4M, 33M)
Other graphs
2d-2e20/2D grid (1M, 4M), r4-2e23/Uniformly random (8M, 33M)
Meshes
25k, 250K, 1M and 5M point meshes

Table 6. Input classes and constituent graphs used in the evaluation. Numbers in parentheses indicate number of nodes and edges respectively.

5. Evaluation

We compared the performance of IrGL-generated code[5] that of publicly available, best-of-the-breed CUDA implementations of eight algorithms (Table 7). We also compared with Gunrock [62], a state-of-the-art GPU-based graph analytics framework.

We used a Kepler-based Tesla K40c[6] for the evaluation, using NVCC 7.5 to compile the generated CUDA code except for MST where we use NVCC 6.0 due to a bug in how the NVCC 7.5 compiler compiles Atomic. We also partially disable warp aggregation in SSSP because it also triggers a bug in all CUDA compilers we tested. We used three major classes of inputs: road networks, RMAT-style graphs, and grids and random graphs (Table 6). All programs were timed from start to finish, but the times exclude I/O and the initial and final copies of the graph to and from the GPU. Thus, the time to setup runtime objects such as worklists is included.

By choosing different optimizations, our compiler can generate many variants from the same source code. The choice of optimizations can be different for different input classes even for the same benchmark. To keep the comparison fair, for each benchmark, we use a *single* binary compiled with one set of optimizations for all inputs.

5.1 Overall Results

Five of the eight IrGL-generated programs – MIS, MST, PR (except road networks), SSSP, and TRI – are faster than the third-party implementations on average (Figure 7), with improvements ranging from 1.24x to 5.95x (median 1.4x). DMR performance matches that of the hand-written benchmark. Only IrGL BFS, CC and PR (road) are slower than the handwritten benchmarks.

Merrill et al. [36]'s BFS is highly optimized – it implements all the optimizations discussed in this paper by hand and benefits from the use of customized data structures. PR

[5] We plan to release these as part of LonestarGPU 3.0

[6] Results on a Maxwell-based Quadro M4000 are in the Appendix

Benchmark	Evaluated	Fastest CUDA code	Performance (ms)	
BFS – Breadth-First-Search	TOTEM [24], Medusa [69], Falcon [14], LonestarGPU 2.0 [1], Gunrock [61]	Merrill et al. [36] (bfs-hybrid)	102	19
CC – Connected Components	TOTEM, Gunrock	Soman et al. [55]	179	186
DMR – Delaunay Mesh Refinement	Falcon	LonestarGPU 2.0 [1]	r5M: 8448	
MIS – Maximal Independent Set [34]	NVIDIA CUSP [57]	Che et al. [12]	125	140
MST – Minimum Spanning Tree (Boruvka)	Gunrock, LonestarGPU 2.0	da Silva Sousa et al. [17]	2526	35303
PR – Page Rank	TOTEM	Elsen and Vaidyanathan [20]	1065	1400
SSSP – Single Source Shortest Path	LonestarGPU 2.0, Gunrock	Near–Far (Davidson et al. [18])	7297	491
TRI – Triangle Counting	-	Polak [49]	474	809

Table 7. Baselines for our performance comparison. Performance numbers indicate execution time in milliseconds for the largest road (USA) and RMAT (rmat22) graph respectively. Falcon source code was not made available so we have used timings published by the authors [13]. Gunrock MST is faster, but does not work on RMAT graphs. It is compared separately in Section 5.3.

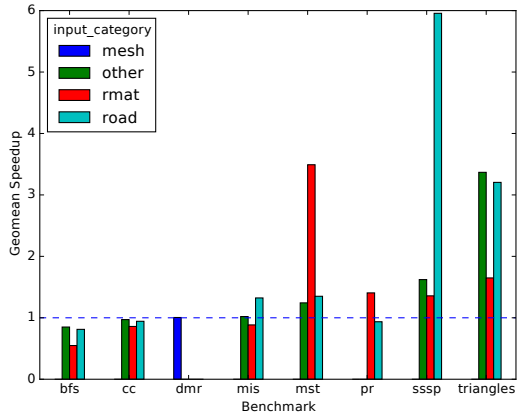


Figure 7. Speedup over hand-optimized code (geometric mean). Higher is better.

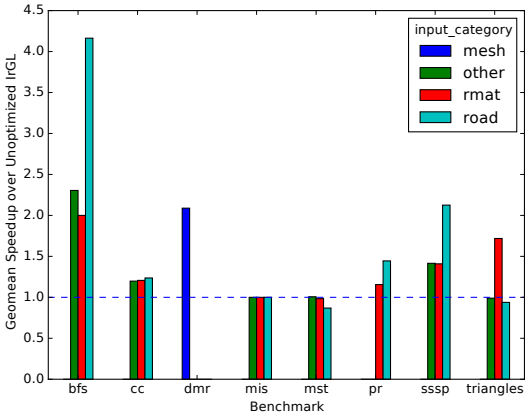


Figure 8. Speedup due to IrGL optimizations

is 7% slower on road networks because the baseline uses a parallel reduction on edges to accumulate PageRank values while we use `atomicAdd`. These are very expensive for road networks, conflicting $12\times$ more compared to RMAT graphs.

Most handwritten implementations do not implement all the optimizations we have identified, even if they are applicable. We ascribe this to the difficulty of finding the right combination of optimizations that will provide a speedup and the large programming effort involved for a systematic search. Our compiler allows programmers to quickly and automatically find the set of optimizations that will benefit their algorithms.

5.2 Benefit of Optimizations

Figure 8 shows that the majority of algorithms benefit from our optimizations. Our optimizations improve the performance of BFS by $4.16\times$ for road networks, with a minimum of $1.15\times$ for PR on RMAT-style graphs. The median improvement is $1.4\times$. For MST (road), our optimizations result in slowdowns while not affecting the other classes. We also find speedups due to the same optimizations vary significantly across input classes.

Figure 9 shows the contributions of each optimization in the benchmarks that had all three optimizations enabled. Connected Components does not benefit from most optimizations, but its slowdowns vary by class of graph. In SSSP, exploiting nested parallelism (np) significantly benefits RMAT-style graphs but negatively affects processing of road graphs which prefer Iteration Outlining (iter). Lest this be attributed solely to the high-diameter of road networks and the consequent high number of kernel calls, we point out that PageRank on RMAT-style graphs benefits from Iteration Outlining despite having a low number of kernel calls. It is the *rate* of kernel invocations that matters, not their number. Figure 9 shows that the optimizations interact in complex and unpredictable ways; having a compiler like IrGL is essential for exploring these interactions.

These results strongly motivate hybrid implementations – cases where it may be profitable to generate multiple versions of the same algorithm with different optimizations applied. Having a compiler like IrGL is essential for this.

5.3 Comparison to Gunrock

Gunrock [61] is a C++ template-based library for graph analytics. Gunrock programs operate on frontiers of nodes

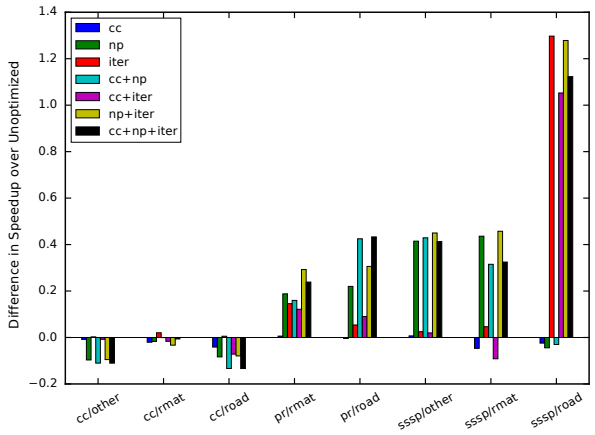


Figure 9. Difference in speedup for all combinations of optimizations. Legend: cc: cooperative conversion, np: nested parallelism, iter: iteration outlining

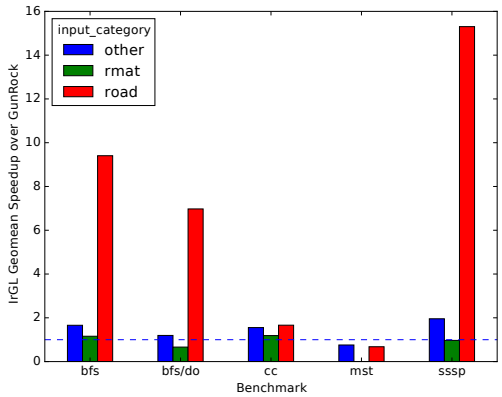


Figure 10. IrGL speedup over Gunrock. Higher is better.

or edges. A filtering operation removes inactive nodes or edges from this frontier after which user-defined functors are applied to frontier in parallel. This restricted programming model is sufficient for graph analytics, but makes it difficult to express programs like DMR.

Figure 10 compares the performance of programs Gunrock 0.3 to that of IrGL code. Four of our benchmarks are common, and Gunrock provides two implementations of BFS, work-efficient [36] and direction-optimizing [7] (`bfs/do`) . The latter is algorithmically better for RMAT-style graphs. The IrGL versions are faster than their Gunrock counterparts in all benchmarks except MST and BFS+DO (RMAT). We note that Gunrock MST does not handle unconnected graphs whereas IrGL MST does. Both Gunrock BFS and SSSP would benefit strongly from iteration outlining, but Gunrock does not support iteration outlining.

6. Related Work

Past work on characterization of performance of GPU graph algorithms has indicted the number of kernel calls [63, 65], atomics [10, 42, 63], workload imbalance in traversal [63] as well as other microarchitectural factors such as irregular data-dependent memory accesses. Our findings, from a preliminary study of systematically generated IrGL code, is more nuanced. For example, it is not the number of kernel calls but the *rate* of kernel invocation that matters. Many GPU algorithms (PageRank, MIS, CC) make few kernel calls, but at a high enough rate to benefit from iteration outlining. Similarly, we find *serialization* of nested parallel loops during traversal is the underlying performance bottleneck, not workload imbalance. Workload imbalance can be tackled through solutions like G-Streamline [67], but serialization requires a compiler-based inspector–executor approach. Finally, no characterization study so far has recognized how inspector–executor execution can increase the opportunities for reducing the number of atomics.

Several graph frameworks exist for the GPU [20, 24, 62, 69]. Most do not feature constructs for mutual exclusion, and irregular algorithms like DMR cannot be expressed in these frameworks. Since many of these frameworks are C++ template-based libraries, we consider it unlikely that they can comprehensively tackle the bottlenecks identified in this paper as effectively as our compiler.

Iteration outlining executes GPU kernels in a manner similar to persistent GPU kernel [26, 56]. Our paper is the first to propose a useful representation (the Pipe) and the transformations required to perform this automatically.

Several compilers target nested data parallelism on the GPU by compiling NESL to CUDA [8, 68], but the authors admit that the code produced by these compilers is not yet competitive with handwritten code. CUDA-NP [66] uses programmer-provided OpenMP-like pragmas on loops in CUDA code to generate code that exploits nested parallelism. However, only one policy is available and the schedule is fixed at compile time.

Cooperative conversion is functionally similar to *combining* [21, 27] where fine-grained synchronization operations are combined serially by a thread into a coarser-grained operation at runtime. CPU-like runtime combining cannot be implemented on the GPU – CUDA threads cannot spin safely [52]. Hence, cooperative conversion is performed by our compiler and also allows all threads to perform their requests in parallel.

Static analyses to identify uniform branches have been developed for scalarization of vector code [15], executing GPU code on CPUs [31] and for simplifying verification of GPU kernels [5]. In this work, we choose to construct uniform branches though we could use these analyses to determine if user-provided branches are uniform, potentially increasing the number of focal points.

Most CPU and distributed implementations of graph algorithms [25, 35, 39, 54] are not restricted by programming models like CUDA and so can use sophisticated runtimes to achieve high performance. Thus, other than Green-

Marl [29], a DSL compiler for graph analytics, and the Elixir system [50, 51], most are library-based frameworks. Green-Marl provides primitives like breadth-first traversal and depth-first traversal and emphasizes the generation of code to other runtimes such as Pregel [30]. The CPU optimizations used by Green-Marl do not overlap with the GPU optimizations presented in this work. The Elixir system synthesizes parallel algorithmic variants of graph algorithms from specifications higher-level than IrGL. It currently generates C++ code for the CPU, but could be retargeted to generate IrGL code.

7. Conclusion

This paper calls out for the first time three key throughput optimizations essential for GPU implementations of high-performance graph applications: iteration outlining, cooperative conversion, and exploitation of nested parallelism in graph traversals. We showed that these optimizations can be automated in a compiler that generates CUDA code from IrGL, a high-level programming model for graph applications. We implemented several recently proposed graph algorithms in IrGL and showed that IrGL-generated code outperforms handwritten CUDA code for these applications by up to $6\times$.

Acknowledgments

This research was supported by NSF grants 1218568, 1337281, 1406355, and 1618425, and by DARPA BRASS contract 750-16-2-0004. The Tesla K40 used for this research was donated by the NVIDIA Corporation.

Roshan Dathathri and Tal Ben-Nun caught subtle typographical errors in early versions of this paper. Discussions with Andrew Lenharth were helpful. The anonymous reviewers at OOPSLA suggested changes that significantly improved this paper.

A. Results on the Quadro M4000

We repeat the experiments from our evaluation on a NVIDIA Quadro M4000, a Maxwell-based GPU. Compared to the Kepler K40c, this card has less memory (8GB vs 12GB) and less memory bandwidth (192GB/s vs 288GB/s).

We are unable to run MST from the baseline because it is distributed as a binary-only blob that is not compatible with a Maxwell GPU. We use the same IrGL code as in the Kepler experiments, i.e. we do not tune for Maxwell. The NVIDIA CUDA compiler recompiles the generated code for Maxwell.

A.1 Overall Results

Our overall speedup results (Figure 11) do not change significantly from those on the K40.

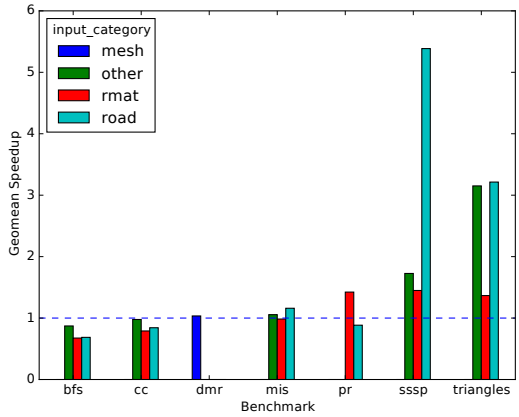


Figure 11. Speedup over hand-optimized code (geometric mean). Higher is better.

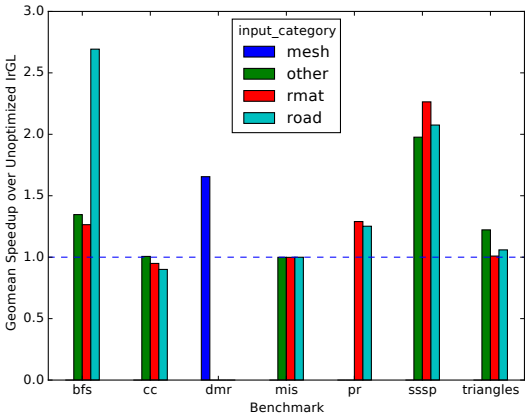


Figure 12. Speedup due to IrGL optimizations

A.2 Benefit of Optimizations

In general, Figure 12 shows lower speedups due to our optimizations. Our examination of the binary code generated by the NVIDIA CUDA compiler for Maxwell devices shows that it performs warp-level aggregation of atomics for Compute Capability 5.2 devices, a feature that we could not turn off. Thus, "unoptimized" IrGL benchmarks are faster on the Maxwell.

Figure 13, showing differences in speedups due to combinations of optimizations, highlights how the value of optimizations change, including relative ordering between them for a benchmark and input. Thus, a selection of optimization does not hold over across generations of GPUs.

A.3 Comparison to Gunrock

IrGL continues to outperform Gunrock for BFS, SSSP and CC as on the Kepler (Figure 14). Relative speedups are generally lower compared to the Kepler, except for IrGL SSSP on road networks.

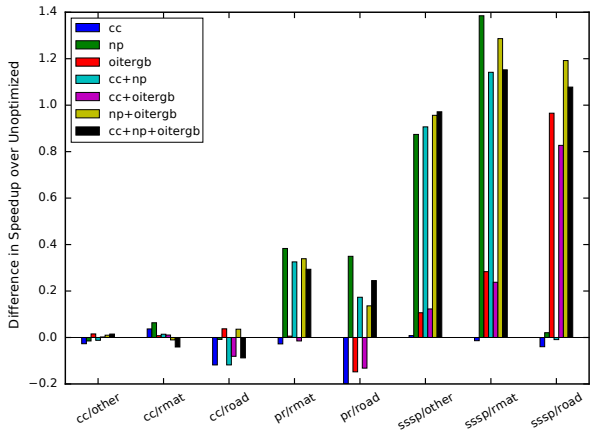


Figure 13. Difference in speedup for all combinations of optimizations. Legend: cc: cooperative conversion, np: nested parallelism, iter: iteration outlining

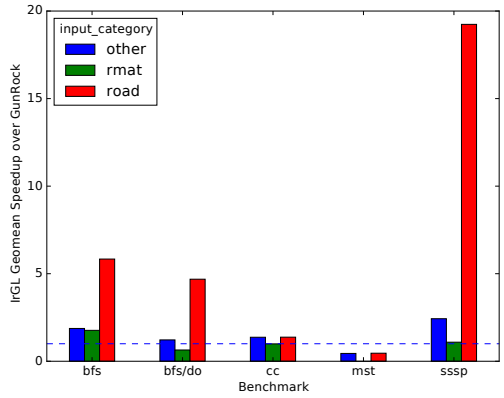


Figure 14. IrGL speedup over Gunrock. Higher is better.

References

[1] The LonestarGPU 2.0 benchmark suite, 2014. URL `http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu`.

[2] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 577–591. ACM, 2015. ISBN 978-1-4503-2835-7. . URL `http://doi.acm.org/10.1145/2694344.2694391`.

[3] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. In *WOLFHPC 2012 - 2nd Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, Salt Lake City, Utah, United States, Nov. 2012. URL `https://hal.inria.fr/hal-00786828`.

[4] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 138–149. IEEE Computer Society, 2015. ISBN 978-1-4673-9524-3. . URL `http://dx.doi.org/10.1109/PACT.2015.17`.

[5] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer. Engineering a static verification tool for GPU kernels. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2014. ISBN 978-3-319-08866-2. . URL `http://dx.doi.org/10.1007/978-3-319-08867-9_15`.

[6] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. ICS '08, 2008.

[7] S. Beamer, K. Asanovic, and D. A. Patterson. Direction-optimizing breadth-first search. In J. K. Hollingsworth, editor, *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 12. IEEE/ACM, 2012. ISBN 978-1-4673-0804-5. . URL `http://dx.doi.org/10.1109/SC.2012.50`.

[8] L. Bergstrom and J. H. Reppy. Nested data-parallelism on the GPU. In P. Thiemann and R. B. Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 247–258. ACM, 2012. ISBN 978-1-4503-1054-3. . URL `http://doi.acm.org/10.1145/2364527.2364563`.

[9] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10, 2015. . URL `http://doi.acm.org/10.1145/2743017`.

[10] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IISWC 2012, La Jolla, CA, USA, November 4-6, 2012*, IISWC 2012, 2012.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining*, pages 442–446.

[12] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2013, Portland, OR, USA, September 22-24, 2013*, pages 185–195. IEEE Com-

puter Society, 2013. ISBN 978-1-4799-0553-9. . URL http://dx.doi.org/10.1109/IISWC.2013.6704684.

[13] U. Cheramangalath, R. Nasre, and Y. Srikant. Falcon: A graph manipulation language for heterogeneous systems. Technical Report 2015-5, Indian Institute of Science, Department of Computer Science and Automation, 2015.

[14] U. Cheramangalath, R. Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. *TACO*, 12(4):54, 2016. . URL http://doi.acm.org/10.1145/2842618.

[15] S. Collange. Identifying scalar behavior in CUDA kernels. Technical report, Jan. 2011. URL https://hal.archives-ouvertes.fr/hal-00555134.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms, McGraw Hill, 2001.

[17] C. da Silva Sousa, A. Mariano, and A. Proença. A generic and highly efficient parallel variant of Borůvka's algorithm. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015.

[18] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE IPDPS*, 2014.

[19] I. J. Egielski, J. Huang, and E. Z. Zhang. Massive atomics for massive parallelism on GPUs. In D. Grove and S. Z. Guyer, editors, *International Symposium on Memory Management, ISMM '14, Edinburgh, United Kingdom, June 12, 2014*, pages 93–103. ACM, 2014. ISBN 978-1-4503-2921-7. . URL http://doi.acm.org/10.1145/2602988.2602993.

[20] E. Elsen and V. Vaidyanathan. Vertexapi2 – a vertex-program api for large graph computations on the gpu. 2014. URL www.royal-caliber.com/vertexapi2.pdf.

[21] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 257–266. ACM, 2012. . URL http://doi.acm.org/10.1145/2145816.2145849.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/24039.24041.

[23] Z. Fu, B. B. Thompson, and M. Personick. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In P. A. Boncz and J. Larriba-Pey, editors, *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, pages 2:1–2:6. CWI/ACM, 2014. ISBN 978-1-4503-2982-8. . URL http://doi.acm.org/10.1145/2621934.2621936.

[24] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT '12*. ACM, 2012. .

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30. USENIX Association, 2012. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[26] K. Gupta, J. A. Stuart, and J. D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing*, May 2012.

[27] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In F. M. auf der Heide and C. A. Phillips, editors, *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364. ACM, 2010. ISBN 978-1-4503-0079-7. . URL http://doi.acm.org/10.1145/1810479.1810540.

[28] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In C. Cascaval and P. Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 267–276. ACM, 2011. ISBN 978-1-4503-0119-0. . URL http://doi.acm.org/10.1145/1941553.1941590.

[29] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In T. Harris and M. L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 349–362. ACM, 2012. ISBN 978-1-4503-0759-8. . URL http://doi.acm.org/10.1145/2150976.2151013.

[30] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying scalable graph processing with a domain-specific language. In D. R. Kaeli and T. Moseley, editors, *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, page 208. ACM, 2014. ISBN 978-1-4503-2670-4. . URL http://doi.acm.org/10.1145/2544137.2544162.

[31] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In M. F. P. O'Boyle, editor, *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7210 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2012. ISBN 978-3-642-28651-3. . URL http://dx.doi.org/10.1007/978-3-642-28652-0_1.

[32] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.*, 44(4), Feb. 2009.

[33] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proc. Conf. Uncertainty in Artificial Intelligence*, UAI '10, July 2010.

[34] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. . URL http://dx.doi.org/10.1137/0215074.

[35] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010. ISBN 978-1-4503-0032-2. . URL http://doi.acm.org/10.1145/1807167.1807184.

[36] D. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU graph traversal. In *PPOPP 2012*. ACM, 2012. .

[37] R. Nasre, M. Burtscher, and K. Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *IPDPS 2013*, 2013.

[38] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on GPUs. In *PPoPP '13*, PPoPP '13, 2013.

[39] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471. ACM, 2013. ISBN 978-1-4503-2388-8. . URL http://doi.acm.org/10.1145/2517349.2522739.

[40] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110 (whitepaper).

[41] *The CUDA C Programming Guide 7.5*. NVIDIA, 2015.

[42] M. A. O'Neil and M. Burtscher. Microarchitectural performance characterization of irregular GPU kernels. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 130–139. IEEE Computer Society, 2014. ISBN 978-1-4799-6452-9. . URL http://dx.doi.org/10.1109/IISWC.2014.6983052.

[43] *The OpenACC API 2.0*. OpenACC.org, 2013.

[44] *The OpenMP API 4.0*. OpenMP Architecture Review Board, 2013.

[45] S. Pai and K. Pingali. Lowering IrGL to CUDA. *CoRR*, abs/1607.05707, 2016. URL http://arxiv.org/abs/1607.05707.

[46] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. PACT 2012. ACM, 2012. .

[47] K. Pingali and G. Bilardi. Optimal Control Dependence Computation and the Roman Chariots Problem. *ACM Trans. Program. Lang. Syst.*, 19(3):462–491, May 1997. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/256167.256217.

[48] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI 2011*, PLDI 2011. ACM, 2011. .

[49] A. Polak. Counting triangles in large graphs on GPU. *CoRR*, abs/1503.00576, 2015. URL http://arxiv.org/abs/1503.00576.

[50] D. Prountzos, R. Manevich, and K. Pingali. Elixir: a system for synthesizing concurrent graph programs. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 375–394. ACM, 2012. ISBN 978-1-4503-1561-6. . URL http://doi.acm.org/10.1145/2384616.2384644.

[51] D. Prountzos, R. Manevich, and K. Pingali. Synthesizing parallel graph programs via automated planning. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 533–544. ACM, 2015. ISBN 978-1-4503-3468-6. . URL http://doi.acm.org/10.1145/2737924.2737953.

[52] A. Ramamurthy. Towards scalar synchronization in SIMT architectures. Master's thesis, The University of British Columbia, 2011.

[53] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007*, 2007.

[54] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146. ACM, 2013. ISBN 978-1-4503-1922-5. . URL http://doi.acm.org/10.1145/2442516.2442530.

[55] J. Soman, K. Kothapalli, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*, pages 1–8. IEEE, 2010. . URL http://dx.doi.org/10.1109/IPDPSW.2010.5470817.

[56] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg. Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.*, 33 (6):228:1–228:11, 2014. . URL http://doi.acm.org/10.1145/2661229.2661250.

[57] L. O. Steven Dalton, Nathan Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. URL http://cusplibrary.github.io/. Version 0.5.0.

[58] A. Venkat, M. Hall, and M. Strout. Loop and data transformations for sparse matrix code. PLDI 2015, 2015.

[59] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013. ISSN 1544-3566. .

[60] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proceed-*

ings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics 2009. ACM, 2009. .

[61] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. PPoPP 2015. ACM, 2015. .

[62] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In R. Asenjo and T. Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, page 11. ACM, 2016. ISBN 978-1-4503-4092-2. . URL `http://doi.acm.org/10.1145/2851141.2851145`.

[63] Y. Wu, Y. Wang, Y. Pan, C. Yang, and J. D. Owens. Performance characterization of high-level programming models for GPU graph analytics. In *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, pages 66–75. IEEE, 2015. ISBN 978-1-5090-0088-3. . URL `http://dx.doi.org/10.1109/IISWC.2015.13`.

[64] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. IPDPS 2010. IEEE, 2010. .

[65] Q. Xu, H. Jeon, and M. Annavaram. Graph processing on GPUs: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 140–149. IEEE Computer Society, 2014. ISBN 978-1-4799-6452-9. . URL `http://dx.doi.org/10.1109/IISWC.2014.6983053`.

[66] Y. Yang and H. Zhou. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. PPoPP 2014. ACM, 2014. .

[67] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In R. Gupta and T. C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 369–380. ACM, 2011. ISBN 978-1-4503-0266-1. . URL `http://doi.acm.org/10.1145/1950365.1950408`.

[68] Y. Zhang and F. Mueller. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *41st International Conference on Parallel Processing, ICPP 2012, Pittsburgh, PA, USA, September 10-13, 2012*, pages 340–349. IEEE Computer Society, 2012. ISBN 978-1-4673-2508-0. . URL `http://dx.doi.org/10.1109/ICPP.2012.21`.

[69] J. Zhong and B. He. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 25(6), 2014. .