Efficient Execution of Graph Algorithms on CPU with SIMD Extensions

Ruohuang Zheng
Department of Computer Science
University of Rochester
Rochester, New York, USA
rzheng3@ur.rochester.edu

Sreepathi Pai
Department of Computer Science
University of Rochester
Rochester, New York, USA
sree@cs.rochester.edu

***Abstract*—Existing state-of-the-art CPU graph frameworks take advantage of multiple cores, but not the SIMD capability within each core. In this work, we retarget an existing GPU graph algorithm compiler to obtain the first graph framework that uses SIMD extensions on CPUs to efficiently execute graph algorithms. We evaluate this compiler on 10 benchmarks and 3 graphs on 3 different CPUs and also compare to the GPU. Evaluation results show that on a 8-core machine, enabling SIMD on a naive multi-core implementation achieves an additional 7.48x speedup, averaged across 10 benchmarks and 3 inputs. Applying our SIMD-targeted optimizations improves the plain SIMD implementation by 1.67x, outperforming a serial implementation by 12.46x. On average, the optimized multi-core SIMD version also outperforms the state-of-the-art graph framework, GraphIt, by 1.53x, averaged across 5 (common) benchmarks. SIMD execution on CPUs closes the gap between the CPU and GPU to 1.76x, but the CPU virtual memory performs better when graphs are much bigger than available physical memory.**

***Index Terms*—CPU SIMD, graph algorithms, Intel ISPC**

I. Introduction

Modern CPUs and GPUs are increasingly similar. While CPUs are not intrinsically SIMD, they commonly support SIMD execution through the use of SIMD extensions or short vector instructions. Intel's recently added AVX512 extensions enables CPUs to process 16 32-bit integers or floats in one instruction, which is half the size of an NVIDIA GPU warp.

Until recently, using these CPU SIMD extensions meant using SIMD intrinsics or relying on compiler autovectorization. With the insight, attributed to Tim Foley, that "auto-vectorization is not a programming model" [1], and that scalar programming models like CUDA were more suitable for generating SIMD code, there is now a third alternative for using the SIMD extensions on CPUs. This programming model, implemented by the Intel SPMD Compiler [2], and referred to as *ISPC* in this work, allows programmers to write mostly scalar code with some annotations that lead to the generation of multi-threaded SIMD code. This is both portable *and* highly controllable.

While state-of-the-art parallel CPU graph frameworks [3]–[18] have successfully increased efficiency of running on multiple cores, they do not make use of SIMD-style execution. The availability of ISPC allows the use of GPU programming model paradigms on CPUs. This prompts our work, where we take a state-of-the-art GPU compiler for graph algorithms – the IrGL compiler [19] – and retarget it to ISPC. Not only does this allow us to generate native SIMD versions of graph algorithms, it also allows us to study the effectiveness of the optimizations originally developed for GPUs and to directly compare CPUs and GPUs, as we run the same graph algorithms with the same optimizations applied on both the devices. Since GPUs now also support virtual memory, e.g. NVIDIA's Unified Virtual Memory (UVM), we can also compare the virtual memory systems of the CPU and the GPU. This paper makes the following contributions:

- We present the first framework for SIMD CPU algorithms based on a recent GPU compiler for graph algorithms. On average, compared to state-of-the-art CPU graph frameworks, Ligra, GraphIt, and Galois, the optimized multi-core SIMD implementation is 3.06x, 1.53x, 1.78x faster, respectively.
- We show that GPU-oriented optimizations also work for SIMD on CPUs delivering an additional 1.67x improvement over the naive SIMD implementation.
- We evaluate the effects of different SIMD variants (AVX, AVX2 and AVX512), finding that wider SIMD extensions may not always be better.
- Our work enables direct comparison of CPU SIMD and GPU graph algorithms which shows that the GPU only outperforms our fastest CPU by 1.52x after use of SIMD. Furthermore, for large graphs that do not fit in GPU memory, we find that the GPU's virtual memory subsystem can cause dramatic slowdown (more than 5000x) on our GPU making it totally unusable.

The rest of this paper is organized as follows. Section II provides necessary background and briefly discusses related work. Section III discusses the challenges and solutions of efficiently executing graph algorithms on CPUs in SIMD fashion. Section IV presents the performance evaluation results and compares CPUs and GPUs. And finally, Section V concludes the paper.

II. SIMD Graph Algorithms on CPUs

We describe the challenges in writing SIMD graph algorithms, and introduce the ISPC programming model, and provide a brief review of related work.

A. SIMD Graph Algorithms

Graph algorithms raise significant challenges for SIMD implementations. Manually writing SIMD code – either via assembly language or through the use of SIMD intrinsics – is mechanical, tedious, and ultimately results in code that lacks portability. If the SIMD instruction set changes then code must be rewritten. On the x86, which has gone through at least six different short vector extensions so far – MMX, SSE, SSE2, AVX, AVX2, and AVX512 – manually targeting SIMD can be particularly unproductive.

Auto-vectorization, which parallelizes scalar code and generates SIMD code does not suffer this portability problem. Although most modern compilers such as GCC and LLVM support autovectorization, the sparse data structures used by graph algorithms (e.g. compressed sparse row [CSR]) lead to heavy use of indirect memory accesses in the code. These indirect memory accesses and accompanying irregular loop structures hinder most auto-vectorization algorithms. Recent work has made considerable progress in tackling sparse data structures [20]–[22], but these remain unavailable in production compilers.

GPU SIMD implementations do not have deal with these problems since they use a mostly scalar, explicitly parallel programming model. When combined with the GPU's hardware support for handling control and memory divergence, this programming model neither requires use of SIMD intrinsics nor auto-vectorization support. CPU SIMD instruction sets have recently added similar hardware functionality. The Intel Haswell architecture's *AVX2* instruction set includes dedicated gather load instructions. The AVX512 instruction set, introduced in the Xeon Phi x200 and Skylake-X server CPUs, added eight opmask registers that allows individual SIMD lanes to be masked (i.e. predicated) for most instructions simplifying the handling of control divergence. It also added dedicated scatter store instructions. This leads to a new model, *implicit SPMD*, for SIMD algorithms.

B. Implicit SPMD

Traditionally, data-parallel programs that use multi-process or multi-threaded execution can be written in the single program multiple data (SPMD) style. In SPMD, parallel tasks are created to *independently* execute a single program, with data decomposition [23] used to distribute data to each parallel task. SIMD, on the other hand, operates at the instruction level and therefore each SIMD "task" belonging to the same instruction executes in lockstep. GPU programming models, like CUDA and OpenCL, showed that it is possible to write SIMD programs that execute in a SPMD fashion while ostensibly writing scalar code.

The Intel Implicit SPMD Program Compiler (ISPC) [2] is a compiler for a C-like language that implements a similar SPMD+SIMD programming model for CPUs. SPMD execution is achieved using multiple threads, while SIMD execution is obtained by using the SIMD extensions on CPUs. Since the CPU also supports purely scalar execution, unlike most GPUs, ISPC differs from GPU programming models and natively supports scalar execution. In contrast, efficient scalar execution of GPU programming models usually requires sophisticated analyses [24], [25]. The primary challenge for ISPC is to achieve SIMD execution from a primarily SPMD code, while maintaining the illusion that multiple SIMD lanes are executing independently.

TABLE I
MAP OF CUDA TO ISPC CONSTRUCTS

CUDA construct	ISPC construct	Executed on CPU by
CUDA Thread	Program Instance	SIMD Lane
Warp	ISPC Task	OS Thread
Thread Block	N/A	N/A

```
1  // example.ispc
2  task void reduction(uniform int * uniform array,
3          uniform int size, uniform int * uniform out) {
4      uniform int size_per_task = size / taskCount;
5      uniform int start = taskIndex * size_per_task;
6      int sum = 0; // Defaults to varying type
7      foreach (i = 0 ... size_per_task)
8          sum += array[start + i];
9      atomic_add_global(out, reduce_add(sum));
10 }
11 export void launch_reduction(uniform int num_task,
12     uniform int * uniform array, uniform int size,
13     uniform int * uniform out)
14 { launch[count] reduction(array, size, out); }

16 // example.cc
17 int main(int argc, char *argv[]) {
18     int sum = 0;
19     int *array = new int[4096];
20     // ... Init the array
21     launch_reduction(16, array, 4096, &sum);
22     cout << "Sum: " << sum << endl;
23 }
```

Listing 1. A simple ISPC program

ISPC programs are explicitly parallel. The basic building block is a *program instance* that is mapped to SIMD lanes. Multiple program instances form an *ISPC task* which is usually executed by an OS thread. Table I links these to their CUDA counterparts, note there is no ISPC equivalent for a CUDA thread block.

Listing 1 sketches a simple ISPC program that calculates the sum of an array of integers. It is spread across two source files: a main program written in C/C++ and an ISPC source file containing the computation kernels. The ISPC kernels are started by calls from the main program, similar to calling a regular function. To take advantage of multi-threading (in addition to SIMD), ISPC also supports launching multiple tasks through the *launch* statement. The *launch* statement invokes an underlying tasking system that creates multiple OS threads on to which tasks are mapped.

The ISPC language used for writing kernels is C-like, except that all variables are treated as vectors and implicitly declared as *varying* – meaning each program instance sees a different value for the variable. Thus, they behave like vectors and naturally, they will be handled by SIMD instructions. Scalar variables, which are shared by all program instances in the

same task, are declared as *uniform*.[1] In Listing 1, all arguments to the ISPC kernels are marked as *uniform*. The range of array indices that will be summed by each task is then calculated by splitting up the total work among all the tasks launched (`size_per_task`), and using that to identify the *start* of the task's block of work. The four built-in variables – programIndex (identifies program instance), programCount (SIMD width), taskIndex (task identifier), and taskCount (total tasks launched) – can be used by ISPC tasks to carve out work using data decomposition.

The variable `sum` is varying, and is local to each program instance. The *foreach* statement is used to indicate a parallel loop. In this case, the loop generates vector instructions that add each vector element of `sum` to a corresponding element of `array`. As all accesses to `array` are consecutive, a standard vector load can be used. However, in the general case, a gather instruction can be generated to load values from `array`. Once the *foreach* loop has completed, the ISPC library function *reduce_add* is used to reduce the contents of the vector `sum` into a scalar, which is then added to the scalar variable `out` using an ISPC library function *atomic_add_global* to perform the addition atomically across all executing tasks.

Although a ISPC-like programming model could have been built before, several recent hardware features have made a compiler viable. The primary causes of "divergence" in SIMD programs are memory accesses to non-consecutive locations and divergent branches. Hardware support for gathers and scatters deals with the former, whereas support for predication/masking allows dealing with the latter. Of course, ISPC can always generate purely scalar code where hardware support is missing.

C. Related Work

1) SIMD Graph Algorithms on CPUs: Vectorized garbage collection [26] used a vectorized breadth-first search (BFS)-based garbage collection algorithm on the Cray-2. A more recent proposal [27] describes a set of graph processing APIs that express SIMD parallelism on Intel Xeon Phi. Another proposal [28] explores manually vectorizing and optimizing BFS on Phi, achieving about 1.25x speedup over the non-SIMD implementation. In addition, AVX intrinsic accelerated set intersection operation [29] was proposed for a number of graph algorithms, including triangle counting, clique detection, and subgraph matching, achieving 3.6x, 12,7x, and 3.3x speedup, respectively. A variety of work also discuss vectorized set intersection operations [30]–[32].

2) Non-SIMD Graph Algorithms on CPUs: A large number of graph processing frameworks and systems have been proposed for CPUs [3]–[18], [33]–[38], which focus on thread-level parallelism, rather than SIMD. Notably, Ligra [3], Ligra+ [39], Julienne [40], and the most recent GBBS [41] is a line of state-of-the-art work that resembles one type of graph frameworks – C++ template libraries. Galois [42] is also a state-of-the-art graph library implemented in C++. In particular, it has a large collection of graph problems and a variety of algorithms for each problem. GraphIt [4], [43], [44] is another line of state-of-the-art work that resembles another type of graph frameworks – domain-specific language plus an optimizing compiler. It also allows each benchmark to be fine-tuned by "schedules" supplied by the programmer.

[1]For pointers, the pointer variable itself and the data it points to can be either uniform or varying.

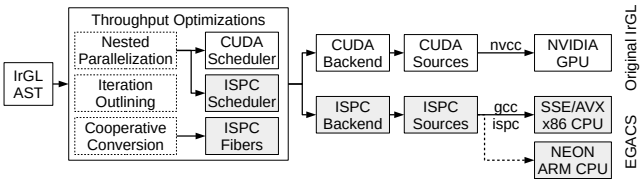


Fig. 1. IrGL and EGACS workflow

3) Graph Algorithms on GPUs and Accelerators: A large number of graph systems have been proposed for GPUs and accelerators [45]–[64]. They are available in libraries and domain-specific language plus compilers as well. In particular, our paper extends the IrGL compiler for irregular graph algorithms on GPUs [19] to CPUs.

Figure 1 highlights the contribution of this work, which we term `EGACS` (**E**fficient **G**raph **A**lgorithms on **C**PU **S**IMD). Shaded boxes represent the new components added to the IrGL compiler, and dashed boxes represent components that are already in IrGL, but modified to incorporate our new ISPC backend. Our work keeps the IrGL DSL frontend, but adds a new backend for Intel ISPC, allowing IrGL programs to be executed on CPUs using SIMD extensions. Graph benchmarks and APIs are also from GPU IrGL.

The use of the ISPC compiler allows our work to automatically support all targets that Intel ISPC supports, including SSE and AVX on x86, and NEON on both 32-bit and 64-bit ARM processors. However, in this work, we focus only on AVX extensions on x86 processors, and leave evaluation of ARM NEON to future work.

III. CPU SIMD Challenges and Solutions

In this section, we examine several performance issues in efficient utilization of SIMD units on CPUs, and show that some of them can be tackled by adapting optimizations first described for the GPU while others require new optimizations.

A. Task Launching

Many graph algorithms are iterative, re-executing a computation a large number of times until convergence. For BFS, the minimum number of iterations is the diameter of the graph, which for some graphs can be thousands of iterations.

In ISPC, each iteration requires tasks to be launched, equivalent to launching kernels on GPUs. While spawning CPU threads or assigning computation tasks to worker threads is not as expensive as launching GPU kernels, it is not free either. Tasks are an ISPC construct, and have different implementations that can be selected at compile time. On Linux, ISPC supports a standard pthread-based [65] task system with 3-D task launching, a simplified version called "pthread_fs" that only

TABLE II
AVERAGE TIME PER LAUNCH IN MICROSECONDS

Machine	pthread	pthread_fs	TBB	TBB_for	OpenMP	Cilk
Intel	62.56	72.14	4.74	5.68	8.85	5.91
AMD	151.21	69.62	32.87	14.47	18.35	8.13

TABLE III
EXECUTION TIME IN MILLISECONDS OF BFS-WL WITH AND WITHOUT OPTIMIZATION, COMPILED WITH DIFFERENT TASK SYSTEMS[a]

Config	pthread	pthread_fs	TBB	TBB_for	OpenMP	Cilk
Unopt	1215.44	1058.27	714.48	643.84	437.26	739.02
IO	261.05	276.00	263.06	260.45	259.06	264.71

[a]Using USA-Road graph. Data collected on Intel machine.

supports 1-D task launching, as well as task systems based on Cilk [66], OpenMP [67], and Intel Thread Building Blocks (TBB) [68].

To characterize the overheads of task launches in ISPC using different task systems, we execute a microbenchmark that only launches tasks that do nothing (i.e. "empty" tasks). Table II compares the time per launch averaged over 10000 continuous launches for all supported task systems. For this test, the number of tasks launched is equal to the total number of hardware threads on the machine. We observe that the pthread-based task system shows the highest launch time, while Cilk shows the lowest launch time.

Table III shows the results of repeating this experiment with a real benchmark, BFS-WL on USA-road graph, with and without the optimization called *Iteration Outlining* (*IO*, described later) that removes task launching overhead. These results show that in real programs, OpenMP has the lowest overhead but all systems still impose significant overheads. We then observe that optimization successfully removes that overhead, making the total execution time nearly the same regardless of task system. Because task launching is on the critical path, lowering its overhead also improves scalability – BFS-WL with USA-Road with 16 tasks on a 8-core SMT-enabled machine shows a 4.59x speedup only when the optimization is applied, compared to no speedup over a single-thread SIMD version.

To tackle both launch overhead and lack of scalability, we retarget the IrGL optimization called Iteration Outlining (IO) that reduces the number of task launches to one. Listing 2 illustrates how this optimization is applied to BFS-WL in ISPC. In the input IrGL code, the iterative loops are represented by a language construct called a Pipe [19]. The default translation of Pipe results in a C++ loop that launches tasks for each iteration. To reduce the number of task launches, however, we transform the iterative loop so that it is "outlined" to ISPC code. The transformed code launches the ISPC *bfs_loop* kernel, which in turn uses a loop to call the original *bfs* kernel. There is only one task launch in this case. To maintain the original semantics of task launches, we insert barriers after each original kernel invocation.

B. Improving SIMD Lane Utilization

Many graph algorithms contain nested loops, the outer of which iterates over all nodes of a graph, while the inner loop

```
1  int main() {
2      // ... Init
3      LEVEL = 1;
4      barrier_t bar(num_tasks);
5      launch_bfs(num_tasks, graph, LEVEL, pipe, &bar);

7      // W/O optimization, multiple launches required
8      // while (pipe.in_wl->nitems()) {
9      //     launch_bfs(num_tasks, graph, &LEVEL, pipe);
10     //     ... Worklist management code
11     //}
12 }
13 task void bfs_loop(...) {
14     while (pipe.in_wl->nitems()) {
15         bfs(graph, *LEVEL, &pipe);
16         barrier_wait(bar);
17         LEVEL++;
18         // ... Worklist management code
19     }
20     barrier_wait(bar);
21     // ... Cleanup
22 }
23 export void launch_bfs(...) {
24     launch[num_tasks]
25         bfs_loop(graph, LEVEL, pipe, bar);
26 }
```

Listing 2. BFS-WL launches only once with optimization

```
1  uniform int wl_end = in_wl->count;
2  for (int wli = tid; wli < wl_end; wli += nthreads) {
3      int node; // Outer
4      bool pop = pipe.in_wl->pop(wli, node);
5      int start = graph.getFirstEdge(node);
6      int end = graph.getFirstEdge(node + 1);
7      for (int edge = start; edge < end; edge++) {
8          int dst = graph.getAbsDst(edge); // Inner
9          if (graph.node_data[dst] == INF) {
10             graph.node_data[dst] = LEVEL; // IfPush
11             pipe.out_wl->push(dst);
12         }
13     }
14 }
```

Listing 3. Generated BFS-WL code snippet (modified for readability)

iterates over edges of a particular node (e.g. Listing 3). In ISPC, each outer loop iteration gets mapped to an ISPC program instance (i.e. a SIMD lane). All iterations of the inner loop are then executed by a single SIMD lane. So unlike regular algorithms like dense matrix multiply, the amount of SIMD parallelism available is dynamic and depends on the input graph. This can lead to SIMD lane underutilization since the number of inner loop iterations are usually different across SIMD lanes.

Table IV shows that SIMD lane utilization when executing the inner loop (Line 8 in Listing 3) is around 64% for USA-Road graph and 32% for RMAT22 graph.The USA-Road graph has low average degree and is relatively uniform, whereas RMAT22 has higher average degree and is highly skewed. After applying optimizations detailed later in this section, lane utilization increases to 82% and 84% for the inner loop for the two inputs respectively. We also observe that applying these optimizations reduces the number of dynamic instructions significantly (e.g. 18x for RMAT22) despite the additional code for scheduling. These primary goal of these optimizations is to increase the amount of work per program instance (Fibers) and address load imbalance (Nested Parallelism/NP).

TABLE IV
SIMD LANE UTILIZATION

Input	Config	Num. of Instrs	Lane Utilization %		
			Outer:L3	*Inner:L8*	*IfPush:L10*
USA-Road	Unopt	564 M	99	62	28
	NP+Fibers	547 M	100	82	35
RMAT22	Unopt	259 M	99	32	10
	NP+Fibers	145 M	100	84	14

1) Fibers: Unlike CUDA, ISPC has no scheduling construct corresponding to thread blocks or shared memory. Since graph algorithms have low arithmetic intensity, we found it advantageous to emulate thread blocks in ISPC to increase the amount of work per ISPC program instance. Thread block emulation is done through *fibers*, which emulate multiple ISPC tasks on the same OS thread. We implement this by inserting additional loops around the actual work loop. The number of loop iterations determines the number of fibers, and data for each virtual task is stored in local, per-task arrays. This is somewhat similar to thread-coarsening in GPU kernels [69].

Fibers enables emulation of CUDA shared memory and the CUDA thread block level *__syncthreads* barrier. Shared memory is emulated by placing variables before any of the fiber loops, thus all fibers have access to that location. The fiber loops need to be partitioned at each *__syncthreads* [70].

When fibers are enabled, each ISPC task now corresponds to a CUDA thread block, while the fibers correspond to CUDA warps. CUDA threads are iterations of the fiber loop, which we refer to as *virtual program instances*.

To choose the number of fibers per ISPC task, we must balance fixed scheduling overheads as well as resource consumption for fiber-specific state arrays. We opt to use a dynamic calculation:

$$NumFibersPerTask = \text{MIN}(MaxNumFibersPerTask, \frac{NumOfItemsInWL}{SIMDWidth \times NumOfTasks})$$

Here, $MaxNumFibersPerTask$ is the maximum number of possible fibers per task, set empirically to 256 to limit resource consumption while maximizing average speedup. $NumOfItemsInWL$ is the current number of worklist items. $SIMDWidth$ is 8 for AVX2 or 16 in AVX512. The $NumOfTasks$ is the number of launched ISPC tasks, usually the number of hardware threads. Each task computes $NumFibersPerTask$ before beginning the actual computation. This way, we only launch one ISPC task per hardware thread which then iterates over all the work.

2) Nested Parallelism: Nested parallelism (NP) redistributes inner loop iterations across ISPC program instances to reduce load imbalance which in turn improves SIMD lane utilization. Figure 2 illustrates how this redistribution works. Initially, nodes are assigned to virtual program instances as usual. Without nested parallelism, the SIMD lane utilization of this hypothetical 5-lane SIMD machine is quite poor. With nested parallelism, however, edges (i.e. inner loop iterations) of high-degree nodes (e.g., B) are distributed across all virtual program instances. Medium-degree node edges (C, D) are executed by real program instances. The edges of the remaining low-degree nodes (e.g., A, E) are packed using a prefix sum and redistributed across all program instances. Our compiler inserts the proper inspector–executor scheduler code to achieve this. This design closely follows the the CUDA backend, with the ISPC fiber, task, and fine-grained schedulers mirroring the CUDA thread block, warp, and fine-grained schedulers.

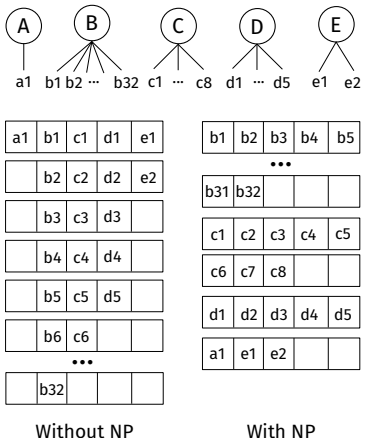


Fig. 2. Nested parallelism

C. Atomic Operations

Work-efficient graph algorithms use a worklist to track active nodes in the graph. Since this worklist is updated concurrently by program instances, the use of atomics is necessary.

Unlike GPUs, the CPU hardware does not yet provide native vector atomic instructions. However, ISPC provides built-in atomic functions that can act as vector atomic instructions. These atomic functions come in *local* and *global* variants. The local atomic functions only provide atomic guarantee within a single task. Since program instances are executed in lock-step fashion in a task, these atomic functions are implemented by a simple loop over all active program instances and do not require a hardware atomic.

In contrast, the global atomic functions provide atomicity guarantees across tasks and can be classified into three types depending on the source and destination operands. For the case of atomic operations on scalar memory locations with scalar input value, the translation to hardware is direct. For atomic vector-to-vector operations, a loop uses hardware scalar atomic instructions to update each memory location for each active program instance. The final case involves atomic updates from a vector value to a scalar location. These are implemented by first performing a vector reduction on the input vector within a task to obtain a scalar, and then issuing a scalar atomic operation to the scalar memory location. Since atomics are executed serially in hardware, reducing the number of atomics can improve performance.

To reduce the number of atomics executed, the Cooperative Conversion optimization transforms code so that multiple program instances aggregate their worklist pushes locally and

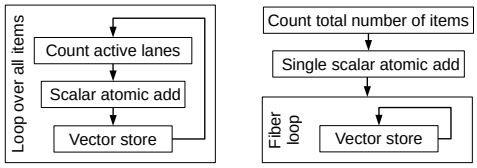


Fig. 3. Task-level (left) and fiber-level (right) cooperative conversion.

TABLE V
NUMBER OF ATOMIC WORKLIST PUSHES, REDUCTIONS IN PARENTHESES

Benchmark	Unopt	+NP+CC		+NP+CC+Fibers	
bfs-wl	5613534	4707262	(1.19x)	4707262	(1.19x)
bfs-cx	12414660	3610932	(3.44x)	17371	(714.68x)
bfs-tp	11323019	7823806	(1.45x)	6180474	(1.83x)
bfs-hb	12414660	3610932	(3.44x)	17371	(714.68x)
cc	1653040	1653040	(1.00x)	1653040	(1.00x)
sssp	10197285	8489255	(1.20x)	8489255	(1.20x)
mis	3074720	1600115	(1.92x)	1600115	(1.92x)
pr	58736245	43130914	(1.36x)	44168071	(1.33x)
mst	8240779	6927180	(1.19x)	4570299	(1.80x)

eventually perform one atomic for the aggegrated push. In the original IrGL GPU compiler, aggregation is performed at the warp and thread block levels. In the CPU compiler, we also aggregate atomics unconditionally at the task level across program instances. This is done by replacing an worklist push with an aggregate version as shown below:

```
1  void push_task(uniform WL * uniform wl, int item) {
2      uniform int cnt = popcnt(lanemask());
3      uniform int idx = atomic_add_global(wl->idx, cnt);
4      packed_store_active(&wl->content[idx], item);
5  }
```

First the ISPC built-in functions *popcnt(lanemask())* obtains the number of pushes by counting the number of active program instances. Then, the ISPC built-in atomic function *atomic_add_global* reserves space for all pushes using a task level atomic scalar *atomic_add_global*. Lastly, the ISPC built-in function *packed_store_active* takes values to be pushed from active program instances, packs them, and writes them to consecutive memory locations in the reserved space.

Since ISPC has no concept corresponding to a CUDA thread block, we instead implement cooperative conversion on fibers, illustrated in Figure 3. For some algorithms, total number of items that get pushed into the worklist can be calculated in advance. In such case, only one atomic add is required with the help of fibers. Unlike the GPU, since fibers still belong to the same ISPC task, lockstep execution is guaranteed permitting a more efficient implementation of fiber-level cooperative conversion.

Table V shows how the number of atomic worklist pushes is reduced after cooperative conversion. We always enable nested parallelism since it increases the number of active program instances increasing the effectiveness of cooperative conversion. Cooperative conversion significantly reduces the number of atomic pushes over unoptimized for all benchmarks where it is applicable. Fiber-level aggregation is only applicable in two benchmarks, bfs-cx and bfs-hb, where the number of atomic pushes is further reduced by 36.5x for BFS-CX (for a total of 125x).

TABLE VI
AVERAGE 32-BIT LOAD-TO-USE LATENCY IN NANOSECONDS ON L1/L2/L3 HITS

Instr. Type	Intel			AMD			Phi	
	L1	*L2*	*L3*	*L1*	*L2*	*L3*	*L1*	*L2*
AVX2	1.02	2.00	5.82	0.97	3.41	3.95	1.81	4.60
AVX512	0.59	0.96	3.10	N/A	N/A	N/A	0.98	1.99
Scalar1	2.35	8.43	39.5	1.79	5.34	24.96	3.64	15.30
Scalar8	0.30	1.06	5.19	0.26	0.86	3.12	1.06	1.96
Scalar16	0.31	0.63	2.80	0.24	0.51	1.56	1.51	1.62

D. Gather Loads and Scatter Stores

In early iterations of SIMD extensions, vectorized memory accesses to and from non-consecutive addresses were not supported. Gather instructions, which load from non-consecutive addresses, were first introduced in AVX2 while scatter instructions, which do the same for stores, were introduced in AVX512. These instructions take a 64-bit general-purpose register as base address and a vector register that provides offsets. AVX2 supports eight 32-bit offsets or four 64-bit offsets; AXV512 doubles this. Without these instructions, the ISPC compiler must generate scalar loops to perform gathers and scatters.

Gather and scatter operations are crucial for SIMD graph algorithms. Indirect memory accesses (i.e. `a[b[i]]`) are common in many graph formats, such as CSR, where they are used to locate the edges for each node. Although AVX2 and AVX512 provides specialized instructions to handle gather and scatter operations, we found performance issues compared to their scalar variants. We use a microbenchmark that loads random cache lines from a pre-allocated array using scalar or gathers. In the gather, each SIMD lane accesses a different cache line. We size the array to correspond to a particular level of cache and make random accesses. This makes it very likely that most accesses are satisfied from that particular cache level.

Table VI summarizes the results. Each row represents a vector or scalar load configuration. *Scalar1* through *Scalar32*, or simply *Scalarm*, consecutively load from *m* scalar addresses to different scalar registers. For *Scalar16*, this introduces register spills, and the latency shown here incorporates this, and is not load instruction latency alone. *AVX2* and *AVX512* are the average load-to-use latency of a single 32-bit word using AVX2 and AVX512 gather loads respectively. Multiplying by the SIMD width will yield the load-to-use latency of the entire instruction.

We observe that the average per-word latency for *AVX2* (1.02ns under L1 hit) is much higher than *Scalar8* (0.30ns under L1 hit), and this holds on both our Intel and AMD machines. The AVX2 gather cannot complete until *all* eight 32-bit words finish loading, while out-of-order execution allows the *Scalar8* load instructions to proceed as soon as the individual memory word is fetched.

Only on the Phi processor is the per-word latency of *AVX512* (0.98ns) lower than *Scalar16* (1.51ns) for L1 hits. The Phi is not as aggressive an out-of-order core as the Intel or AMD processors, and hence it is unable to hide the scalar load latency.

TABLE VII
MACHINES USED IN THE EVALUATION

Parameter	Intel	AMD	Phi
Model	Xeon Silver 4108	EPYC 7502P	Xeon Phi 7290
Clock	1.80 GHz	2.50 GHz	1.50 GHz
AVX	AVX512	AVX2	AVX512
Core/Thread	8/16	32/64	72/288
Cache	11 MB L3	128 MB L3	36 MB L2
DRAM	48 GB	256 GB	16 GB MCDRAM
SMT	2-way	2-way	4-way

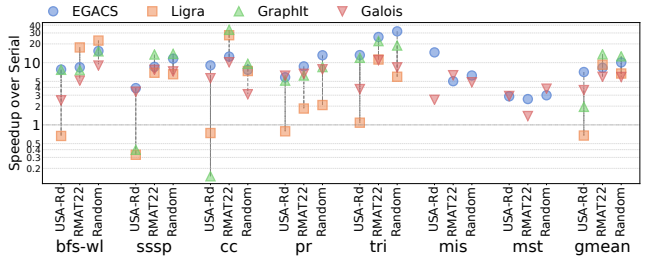
Gather loads are also a problem on GPUs, where they cause memory divergence. Current GPUs use very high degree of simultaneous multithreading (SMT) to hide load latencies and do not implement out-of-order execution. For example, most NVIDIA GPUs run 64 warps per streaming multiprocessor. CPUs do not implement such high degree of SMT, but we find that 2-way and 4-way SMT help alleviate some of this issue.

IV. Evaluation

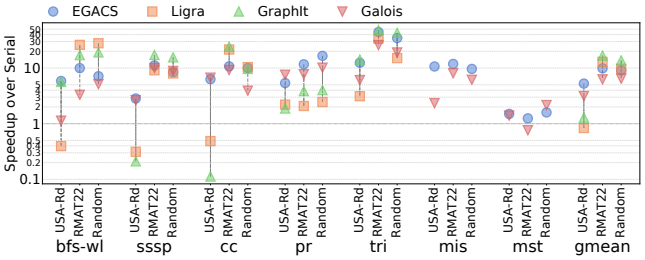
In this section, we evaluate the performance of our SIMD graph algorithms. We also compare our implementation against state-of-the-art proposals including Ligra, GraphIt and Galois. We mainly use two machines, one with an Intel Xeon Silver 4108, the other with an AMD EPYC 7502P. We also run our SIMD graph kernels on a machine with an Intel Xeon Phi 7290 processor. Table VII summarizes machine configurations.

Table VIII lists the kernels used in our evaluation. Note that for ISPC, we run multiple variants of BFS. When comparing against other proposals, we use BFS-WL. For SSSP-NF, we use the same input-specific `DELTA` across all proposals. To make sure our kernels execute correctly, we collect the outputs and check them against the reference output. We also modify Ligra and GraphIt to make sure they produce the same outputs. We also carefully place the timers in all kernels to make sure they measure the same type of useful work across different proposals. We run each kernel 5 times on Phi and GPU, 20 times on Intel and AMD; and then report the average execution time, excluding graph loading and output writing time. We use a pthread-based ISPC tasking system, derived from the stock ISPC pthread tasking system, adding the capability to pin tasks on specific hardware threads. Unless otherwise specified, on the Intel machine, we use 16-wide AVX512 target and launch 16 ISPC tasks; on AMD machine, we use 8-wide AVX2 target and launch 64 tasks.

All frameworks use 32-bit node and and edge indices with 64-bit pointers. We compile Ligra, GraphIt, and Galois with `GCC` using `-O3` which enables auto-vectorization by default. Disabling auto-vectorization using `-fno-tree-vectorize` had no noticeable performance impact. Ligra and GraphIt use Cilk as their underlying task system. For EGACS, we pin threads on physical CPUs. This is for studying scalability and SMT, not for performance. We found pinning alone speeds up EGACS by 2% on average. Pinning was not used for other systems. We use three input graphs: a uniform degree planar USA-Road (23M nodes, 46M edges), a scale-free graph RMAT22 (4M nodes, 33M edges), and uniformly random graph Random (8M nodes, 33M edges).



(a) Intel



(b) AMD

Fig. 4. Overall results comparing EGACS, Ligra, GraphIt, and Galois.

A. Comparison with Scalar Graph Frameworks

We compare the performance of EGACS to other CPU graph processing frameworks. We also evaluate the contributions of each optimization to overall performance.

1) Overall Performance: Figure 4 shows the speed up of EGACS (SIMD kernels with all optimizations enabled), Ligra, GraphIt, and Galois over the serial version. Execution times are in Appendix A. The serial version is derived from our ISPC code by marking all variables `uniform` and setting `task_count` and `program_count` to 1, and recompiling.

On the Intel machine, compared to other state-of-the-art proposals, in total 21 benchmark-input comparisons, our EGACS ranks the fastest in 11 benchmark-input configurations, second fastest in 9 configurations, and is slowest only for some inputs of MIS and MST. Using USA-Road graph, EGACS on average is 10.87x faster than Ligra and 3.77x faster than GraphIt (excluding MST and MIS), 1.95x faster than Galois. EGACS is also the fastest when using Random graph as input except in BFS and SSSP. Ligra performs very well on RMAT22 and Random graphs, for example, Ligra on BFS-WL with RMAT22 graph is 2.08x over EGACS.

On the AMD machine, similar trend is observed. In total 21 benchmark-input comparisons, EGACS ranks the fastest in 9 configurations, second fastest in 8 configurations. Averaged across all common benchmarks and inputs, EGACS is 2.18x faster than Ligra, 1.54 faster than GraphIt, and 1.55x faster than Galois.

Ligra and GraphIt implement direction-optimizing BFS [71] which is fundamentally faster than BFS-WL implemented in EGACS for these graphs (but not for the USA-Road graph). Our implementations lack algorithmic optimizations including not

TABLE VIII
BENCHMARKS AND ABSOLUTE EXECUTION TIME IN MILLISECONDS. FOR EACH INPUT GRAPH, WE INCLUDE SERIAL EXECUTION TIME AND THE BEST PARALLEL EXECUTION TIME AMONG ALL EVALUATED PROPOSALS. DATA COLLECTED ON INTEL MACHINE.

Benchmark	USA-Road (23M/57M)			RMAT22 (4M/33M)			Random (8M/33M)		
	Serial	*Best*		*Serial*	*Best*		*Serial*	*Best*	
BFS - Breadth-First Search	2032.21	261.94	EGACS	1038.53	59.16	Ligra	2695.86	119.19	Ligra
CC - Connected Components	5900.26	652.66	EGACS	6617.11	196.90	GraphIt	2035.48	212.69	GraphIt
TRI - Triangle Counting	2940.38	222.37	EGACS	19323.15	748.00	EGACS	8967.53	281.53	EGACS
SSSP - Single Source Shortest Path	3282.30	841.75	EGACS	6360.37	471.09	GraphIt	5188.37	375.85	GraphIt
MIS - Maximal Independent Set	5036.06	344.64	EGACS	2976.72	592.06	Galois	3893.84	626.32	EGACS
PR - Page Rank	37122.60	6001.35	Galois	24685.45	2833.61	EGACS	48577.26	3697.20	EGACS
MST - Minimum Spanning Tree	27301.77	9363.05	Galois	23433.13	9017.92	EGACS	60958.33	15967.40	Galois

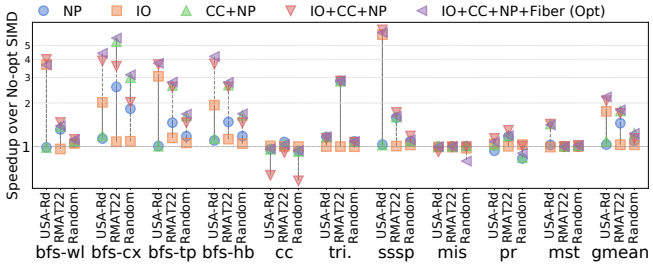


Fig. 5. Individual optimization. Intel machine.

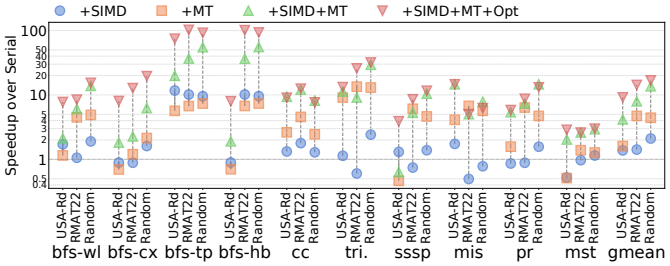


Fig. 6. Speedup due to SIMD and multi-tasking. Intel machine. `+MT`: Multi-tasking, `+SIMD`: SIMD execution, `+Opt`: Throughput Optimizations.

only direction optimization, but also bitvector representation, blocking or vertex scheduling, and asynchronous execution found in the other three graph systems. These algorithmic optimizations make them fundamentally faster than EGACS. In addition, PR and MST are affected by the extensive use of `cmpxchg` in EGACS. However, our focus in this work is on better using machine features, i.e. SIMD, rather than algorithmic improvements.

2) Effect of Throughput Optimizations: Figure 5 shows the effect of individual optimizations on the Intel machine. Task-level cooperative conversion (CC) is always applied in conjunction with nested parallelism (NP). Among our benchmarks, only BFS supports fiber-level CC.

On average, applying all optimizations yields the highest speedup over the unoptimized SIMD version. On the Intel machine, when all optimizations are applied, the average speedup for USA-Road graph is 2.10x, 1.77x for RMAT22, and 1.24x for Random. For each individual kernel and input, speedup ranges from 0.62x to 6.13x. Similar trend is observed on AMD and Phi [72].

It is known that individual optimizations can slow down performance on GPUs [73], and this seems to hold on the CPU as well. For example, `Fibers` works best for BFS-CX and BFS-HB, due to the significant atomic worklist push reduction. However, in BFS-WL and SSSP, the performance benefit is not enough to cancel out the overhead from executing additional `Fibers` code, slowing down the programs.

Some optimizations change the order of work affecting memory locality. For example, `IO+CC+NP` shows quite noticeable slowdown for USA-Road and Random graphs on the CC benchmark. In this case, the generated loop iterates over items in an order that happens to suffer from poor locality, leading to high cache and TLB miss rates. Interestingly, when `Fibers` is enabled on top of `IO+CC+NP` for CC, the slowdown goes away because `Fibers` changes the iteration order, bringing back memory locality.

B. Impact of SIMD

We evaluate the contribution of SIMD, Multi-tasking and throughput optimizations. Then, we evaluate the effects of SIMD width and AVX version.

1) SIMD vs. Multi-tasking: Speedup over serial is due to both SIMD execution on each task as well as running multiple independent tasks. We evaluate the individual contributions of these two sources.

Figure 6 summarizes the contributions of multiple tasks and the use of SIMD. Enabling SIMD alone is beneficial, resulting in 1.37x speedup for USA-Road , 1.45x for RMAT22, and 2.15x for Random over the serial version. Similarly, enabling multi-tasking alone is beneficial on average (1.47x for USA-Road, 4.56x for RMAT22, and 4.26x for Random). However, `+MT` slows down BFS-CX and SSSP with USA-Road since additional atomic operations and barriers required for multi-tasking affect scalability.

With both SIMD and multi-tasking enabled (`+SIMD+MT`), the number of atomics is reduced due to local task-level aggregation, and we see a significant performance boost (3.84x for USA-Road, 7.96x for RMAT22, 13.71x for Random) Adding throughput optimizations (`+MT+SIMD+Opt`) yields the best performance on average (8.06x for USA-Road, 14.08x for RMAT22, and 17.02x for Random).

2) SIMD Width: Figure 7 summarizes the effect of SIMD width. Solid lines represent speedup of a particular AVX target over AVX1-4; dotted lines represent number of dynamic instructions normalized to AVX1-4. To avoid contributions

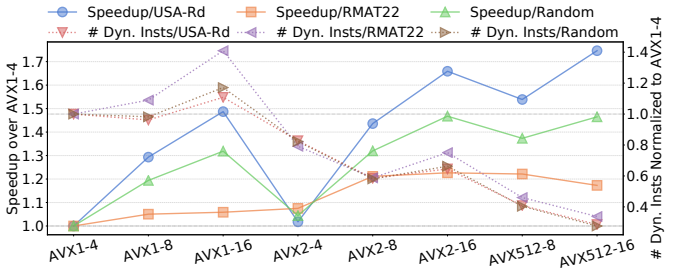


Fig. 7. Speedup (solid lines) and normalized number of dynamic instructions (dotted lines) vs. different AVX targets. Intel machine.

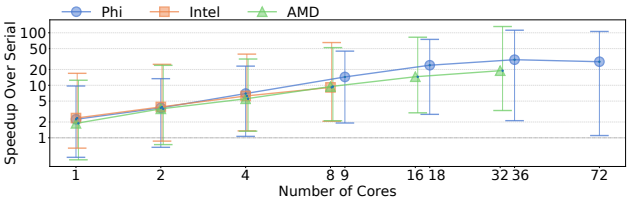


Fig. 8. Speedup vs. different number of cores. Upper and lower error bars represent the best and the worst scaling.

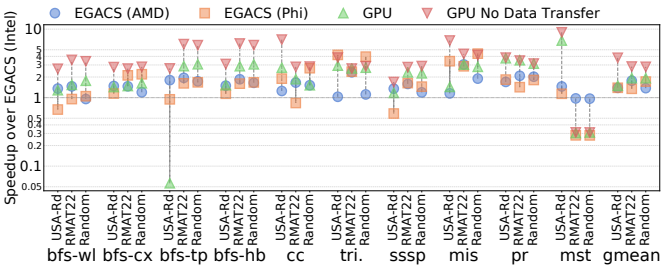


Fig. 9. CPU vs. GPU. GPU data collected on NVIDIA Quadro P5000.

from barriers, task launching, and compare-and-swap retries, we measure the number of dynamic instructions using a single-task SIMD version. For speedup, we still launch multiple ISPC tasks, *i.e.*16 tasks on Intel machine. Since all benchmarks show similar trend for a particular input, each line represent the geomean across all benchmarks for a particular input graph.

On average, wider SIMD is beneficial for USA-Road and Random. Compared to AVX512-8, AVX512-16 results in 1.15x and 1.07x speedup, respectively. However, AVX512-16 and AVX512-8 have roughly the same execution time for RMAT22. Since RMAT22 tends to have full SIMD lanes due to high average degree, a slow 16-wide gather load can prevent future dependent instruction from being issued.

For AVX2, AVX2-16 yields 1.17x and 1.61x speedup over AVX2-8 and AVX2-4, respectively, for USA-Road. For RMAT22, AVX2-16 and AVX2-8 have roughly the same execution time. Since AVX2 natively supports only 8-wide 32-bit integer operations, ISPC simulates 16-wide target by issuing two consecutive 8-wide vector instructions. Therefore, gather loads do not affect AVX2 as much since the two consecutive instructions are guaranteed to be independent and have higher instruction-level parallelism (ILP). However, we observe that AVX2-16 requires more dynamic instructions than AVX2-8, which reduces the benefit of higher ILP for RMAT22, leading to no speedup for AVX2-16 over AVX2-8.

AVX512-16 has fewer dynamic instructions than AVX512-8. For AVX2 and AVX1, the 8-wide version has the least number of instructions. Their 4-wide target under-utilizes the 8-wide SIMD units, while their 16-wide target requires additional instructions. Both result in higher number of instructions over the 8-wide version.

3) AVX Version: Figure 7 also summarizes the effect of different AVX versions. Newer versions of AVX have significant reductions in total dynamic instructions (including scalar) over older versions. For USA-Road, AVX512-16 has 1.41x fewer instructions than AVX2-16 which in turn executes 1.59x fewer instructions than AVX1-16. These reductions are due to the new instructions (such as gathers, scatters) and predication.

The reduction does not always translate to higher performance. For example, AVX512-16 is slower than AVX2-16 for Random and RMAT22 due to microarchitectural implementation. Our Intel processor has only one 16-wide SIMD unit composed of two 8-wide units. The AVX512-16 instructions use the whole unit, whereas the AVX2-16 use two 8-wide independent instructions. This means the AVX2-16 can benefit from both higher ILP as well as lower gather latency.

C. Scalability

Figure 8 shows speedup over serial version as the number of tasks (and hence cores used) is increased. Tasks are pinned to cores, and we do not use SMT, which will be studied later (Section IV-D1). Here we show the geomean across all three inputs since they all have similar scaling trend. As can be seen, the benchmarks scale pretty well, as they show nearly linear scaling for Intel, for AMD with 16 or fewer cores, and for Phi with 18 or fewer cores. Beyond that, the scaling stops for a number of benchmark and input combinations. However, on average, we still see a significant speedup with 32 cores over 16 cores (1.27x) on AMD. On Phi, the 36-core version on average still shows further speedup (1.28x) over 18-core version. However, when all 72 cores are enabled, we see a slowdown over the 36-core version (0.89x). Notably, use of SIMD contributes additional scaling, leading to maximum speed ups of 65.12x on Intel (8 cores), 131.55x for AMD (32 cores) and 111.85x for Phi (36c).

D. CPU vs. GPU

Our experimental setup allows us to compare CPUs against the GPU, running the same kernels generated by the same compiler for the CUDA backend. To ensure a fair comparison, we also include the time for data transfers to and from the GPU. We use a Quadro P5000, which has 20 32-wide streaming multiprocessors. Figure 9 shows the speedup over our EGACS from Intel machine.

On average, GPU is the fastest. The GPU speed up ranges from 0.06x to 6.81x for USA-Road, 0.30x to 3.51x for

TABLE IX
MEMORY FOOTPRINT SIZE IN MB AND SLOWDOWN WHEN AVAILABLE MEMORY EQUALS TO 75% AND 50% OF FOOTPRINT SIZE[a]

App	GPU			CPU		
	Mem	***75%***	***50%***	***Mem***	***75%***	***50%***
bfs-wl	2987	3556.74	DNF[b]	2653	59.46	187.53
cc	9321	2.93	5.95	7472	45.78	OOM[c]
tri	6899	2.20	5474.30	4112	799.50	2720.66
sssp	4928	2828.54	DNF[b]	4643	15.45	51.46
mis	5387	2.92	4.61	5402	39.30	95.13
pr	4436	DNF[b]	DNF[b]	4478	24.46	397.73
mst	13333	2.04	6.33	12602	6.35	OOM[c]

[a]Using OSM-EUR graph: 174M nodes, 696M edges, 3.89 GB
[b]Did not finish after 5 hours. The slowdown is more than 5000x.
[c]Killed due to out-of-memory. The system has only 5.9GB swap space.

RMAT22, and 0.31x to 3.13x for Random, compared to the Intel CPU. However, there are three exceptions where the Intel CPU is notably faster, BFS-TP (USA-Road) and MST (RMAT22 and Random). For BFS-WL(USA-Road) , BFS-CX(USA-Road), and SSSP (USA-Road), AMD is slightly faster than CUDA. Although the specifications of Phi look close to our GPU, this does not always translates to performance. On average, Phi is slower than the GPU. However, we found Phi faster in 8 out of 30 benchmark-input combinations with speedup ranging from 1.31x to 16.80x.

The `GPU No Data Transfer` results do not include the time to copy data to and from the GPU. Without this overhead, unnecessary on the CPU, the GPU is the fastest except for a few benchmarks. Phi is faster than GPU for TRI (USA-Road and Random), MIS (USA-Road and Random) by 1.10x, 1.38x, 1.14x, and 1.29x; Intel is faster than GPU for MST (RMAT22 and Random) by 3.22x and 3.23x, respectively.

1) Effect of Virtual Memory: Recent GPUs support virtual memory [74], lack of which has previously limited graph sizes, permitting us to evaluate the virtual memory capabilities of GPUs for graph algorithms .

Since all our frameworks use 32-bit addressing for graph data structures, we add a larger OSM-EUR graph (174M nodes, 696M edges, 3.89 GB) and limit the amount of physical memory available to a benchmark. This methodology also allows us to study the effect of multiple simulated physical memory sizes using the same input. On the CPU, we use *cgroups* to limit physical memory, while on the GPU we run a separate process to allocate GPU memory using *cudaMalloc* (whose allocations are non-pageable) making it unavailable to the benchmarks.

Table IX shows the memory footprint size of graph analysis on the CPU and GPU version without memory limitations. For our experiments, we limit the physical memory to 75% and 50% of this footprint size, and report normalized execution time. BFS-WL, SSSP, and PR ends up with dramatic slowdown on the GPU, making the system totally unusable, but only moderate slowdown on CPU. When limiting physical memory to 75% footprint size, CPU versions in general show higher slowdown than the GPU, except for BFS-WL, SSSP, and PR. When limiting physical memory to 50% footprint size, only

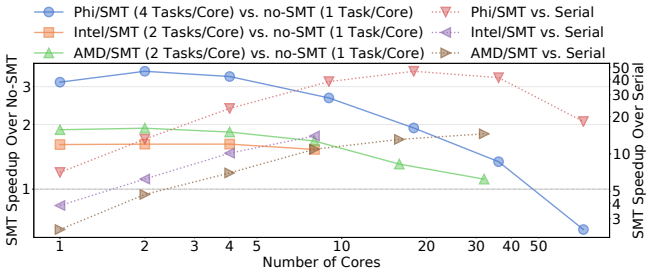


Fig. 10. Assigning multiple tasks on SMT cores.

MIS shows higher slowdown on the CPU than the GPU. Except for the two OOM errors in CC and MST, benchmarks on the CPU do not suffer from catastrophic slowdown.

2) Effect of SMT: We evaluate SMT by launching as many ISPC tasks as hardware threads and pinning each task to a dedicated hardware thread. For our "no-SMT" experiments where SMT is not used, we pin tasks to ensure only one task runs per core.

Figure 10 shows results from Intel, AMD, and Phi. Solid lines represent SMT speedups over no-SMT with the same number of cores enabled. Dotted lines represent speedups over serial version with and without SMT enabled, for a given number of cores. Since all three inputs show almost identical scaling trend, we show the geomean across all inputs.

On average, SMT benefits our benchmarks. For example, with 2 cores, enabling SMT gives us 1.61x, 1.93x and 3.54x speedup over no-SMT on Intel, AMD, and Phi, respectively. However, as the number of cores increase, the benefits decrease – speedups from SMT are only 1.52x and 1.07x when all cores are enabled for Intel (8 cores) and AMD (32 cores). Indeed, on Phi, when all 72 cores are used, we end up with *slowdown* (0.58x).Preliminary experimental results show that L3 latency on the AMD machine for MIS/USA-Road increases 2.30x from 16-thread to 32-thread configuration pointing to increased memory contention as the cause for poor scaling.

V. CONCLUSION

This work has presented the first compiler for CPU SIMD graph algorithms by extending an earlier GPU graph algorithm compiler to generate ISPC code. We have shown that the fastest SIMD implementations outperform the state-of-the-art non-SIMD CPU graph frameworks, Ligra, GraphIt, and Galois by 3.06x, 1.53x and 1.78x, respectively. Since SIMD can be added to these frameworks, our work is complementary to these proposals. The GPU optimizations have been adapted for CPUs, resulting in an additional 1.67x speedup over just SIMD. Comparing the CPU and GPU versions of the same algorithm, we found that even accounting for data transfers the GPU is 1.24x (Phi) to 1.76x (Intel) faster, though its virtual memory subsystem needs improvement.

APPENDIX A
RAW NUMBERS

Table X lists the absolute execution time in milliseconds comparing EGACS, Ligra, GraphIt, and Galois on both Intel

and AMD machines. We use AVX512 on the Intel machine and AVX2 on the AMD machine. Times in bold are the lowest execution time for a particular benchmark-input configuration.

APPENDIX B
ARTIFACT

A. Abstract

Our artifact provides sources for all evaluated benchmarks and the extended IrGL compiler that compiles the benchmarks, along with a set of scripts to run experiments and generate execution data and speedup graphs. We also include Ligra, GraphIt, and Galois sources that we used in writing the paper. The GPU version of our benchmarks are also included. Using these sources and scripts, users should be able to run our experiments on any supported machine and generate result figures and tables.

In addition, we include reference results and plot scripts. These are what we used in writing this paper. They can be used to generate the exact same plots and tables as in the paper.

B. Artifact Checklist

- **Algorithm:** AVX SIMD graph code generation.
- **Program:** BFS, SSSP, CC, TRI, MIS, PR, MST (sources included in the artifact).
- **Compilation:** GCC and ISPC (download script included in the artifact), CUDA (optional).
- **Transformations:** An extended IrGL compiler that targets ISPC, included in the artifact.
- **Binary:** Not included, but can be generated from the sources we provide.
- **Data set:** USA-Road, RMAT22, and Uniformly Random graphs.
- **Run-time environment:** GNU/Linux, x86.

TABLE X
EXECUTION TIMES IN MILLISECONDS

Algorithm	**bfs-wl**			**bfs-cx**			**bfs-tp**			**bfs-hb**		
Graph	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***
Serial	2032.21	1038.53	2695.86	2230.16	1765.41	3623.18	56324.60	22765.13	30310.23	2234.89	22795.74	30380.20
EGACS	**261.94**	123.18	175.10	**277.30**	**138.37**	**185.06**	**755.88**	**222.46**	**328.70**	**279.91**	**225.09**	**325.65**
Ligra	3063.62	**59.16**	**119.19**	NA	NA	NA	NA	NA	NA	NA	NA	NA
GraphIt	265.50	137.92	177.68	NA	NA	NA	NA	NA	NA	NA	NA	NA
Galois	817.95	201.95	300.25	NA	NA	NA	NA	NA	NA	NA	NA	NA

Algorithm	**cc**			**tri**			**sssp**			**mis**		
Graph	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***
Serial	5900.26	6617.11	2035.48	2940.38	19323.15	8967.53	3282.30	6360.37	5188.37	5036.06	2976.72	3893.84
EGACS	**652.66**	529.56	265.68	**222.37**	**748.00**	**281.53**	**841.75**	742.88	447.07	**344.64**	592.06	**626.32**
Ligra	7978.91	239.75	277.72	2712.88	1721.78	1510.77	9922.45	924.70	797.02	NA	NA	NA
GraphIt	39538.90	**196.90**	**212.69**	246.27	872.31	476.37	8270.20	**471.09**	**375.85**	NA	NA	NA
Galois	1047.95	651.10	653.30	781.05	1758.05	1067.15	960.55	828.40	716.80	1990.45	**473.00**	803.80

Algorithm	**pr**			**mst**		
Graph	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***
Serial	37122.60	24685.45	48577.26	27301.77	23433.13	60958.33
EGACS	6379.18	**2833.61**	**3697.20**	9511.18	**9017.92**	20403.02
Ligra	47077.08	13482.99	23320.65	NA	NA	NA
GraphIt	7225.77	3980.38	5730.18	NA	NA	NA
Galois	**6001.35**	3730.70	6212.45	**9363.05**	16880.45	**15967.40**

(a) Intel

Algorithm	**bfs-wl**			**bfs-cx**			**bfs-tp**			**bfs-hb**		
Graph	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***
Serial	1127.05	956.12	1820.60	1197.12	1243.99	2141.42	44102.08	18996.97	23515.74	1192.45	18905.33	23446.25
EGACS	**192.38**	95.86	255.18	**186.79**	**95.17**	**154.53**	**420.19**	**115.98**	**212.20**	**187.27**	**125.74**	**220.60**
Ligra	2848.24	**36.60**	**65.16**	NA	NA	NA	NA	NA	NA	NA	NA	NA
GraphIt	200.18	55.98	95.18	NA	NA	NA	NA	NA	NA	NA	NA	NA
Galois	995.00	289.10	353.40	NA	NA	NA	NA	NA	NA	NA	NA	NA

Algorithm	**cc**			**tri**			**sssp**			**mis**		
Graph	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***
Serial	3608.42	3454.41	1771.98	2703.32	14073.54	8764.69	1771.91	5945.43	4166.72	3185.45	2304.91	3197.87
EGACS	572.21	321.72	176.80	216.99	316.66	253.17	**624.80**	533.31	494.70	**298.63**	**194.40**	**331.10**
Ligra	7406.62	160.31	**170.27**	865.21	416.09	579.83	5673.05	651.90	520.43	NA	NA	NA
GraphIt	32124.93	**142.85**	183.77	**190.43**	**292.58**	**204.58**	8421.77	**346.13**	**267.45**	NA	NA	NA
Galois	**537.10**	373.25	452.25	444.05	543.15	459.85	667.05	591.35	468.55	1358.00	285.30	519.55

Algorithm	**pr**			**mst**		
Graph	***USA-Rd***	***RMAT22***	***Random***	***USA-Rd***	***RMAT22***	***Random***
Serial	20238.76	15909.12	30466.82	10977.92	11653.50	34160.67
EGACS	3769.64	**1361.45**	**1839.78**	**7316.97**	**9342.53**	21458.48
Ligra	9203.36	7640.12	12417.88	NA	NA	NA
GraphIt	10791.36	4127.19	7590.48	NA	NA	NA
Galois	**2671.40**	2051.20	2995.30	7848.40	15260.85	**15787.55**

(b) AMD

- **Hardware:** Multi-core CPU with AVX2 support. We used Xeon Silver 4108, EPYC 7502P, and Xeon Phi 7290. Optionally, a CUDA 10.0 capable GPU. We used NVIDIA Quadro P5000 GPU.
- **Output:** Execution time, speedup and scalability plots.
- **Disk space required:** 15 GB.
- **Time needed to prepare workflow:** 1 hour.
- **Time needed to complete main experiments:** 16 hours.
- **Time needed to complete all experiments:** 53 hours.

C. Description

1) How Delivered: Sources and inputs are available at `https://doi.org/10.5281/zenodo.4279811`

2) Hardware Dependencies: Multi-core CPU with AVX2 support. We used Xeon Silver 4108, EPYC 7502P, and Xeon Phi 7290. Optionally, a CUDA 10.0 capable GPU. We also used NVIDIA Quadro P5000 GPU.

3) Software Dependencies: GNU/Linux (we used Ubuntu 18.04), GCC (we used 7.5.0, or newer version should also work), Intel ISPC 1.12 (download script included), Intel Pin 3.11 (optional, download script included). Python 2.7 and additional packages: `toposort`, `cgen`, `pycparser`, `future`, `numpy`, `matplotlib`, and `seaborn`. Optional for Ligra/GraphIt/Galois: Cilk for GCC 7.5.0, OpenMP 4.5, CMake 3.10, and Boost 1.65. Optional for GPU benchmarks: CUDA 10.0. All other dependencies are included in the artifact.

4) Data Sets: USA-Road, RMAT22 and Random (r4-2e23) graphs.

D. Installation

```
$ # Install packages, assuming Ubuntu 18.04
$ sudo apt install build-essential python python-pip
$ sudo apt install cmake libboost-all-dev # Optional

$ # Download and extract artifact tarball
$ # And then set up root path
$ tar -xvf sources.tar.bz2
$ cd artifact
$ export ARTIFACT_ROOT=$(pwd)

$ # Downgrade Python pytools package,
$ # if your pytools > 2020.1
$ # Please make sure you are using Python 2.7
$ pip uninstall pytools
$ pip install pytools==2020.1

$ # Install Python 2.7 packages
$ pip install toposort cgen pycparser future
$ pip install numpy matplotlib seaborn # For plotting

$ # Download Intel ISPC
$ cd $ARTIFACT_ROOT/ispc
$ ./download

$ # [Optional] Download Intel Pin for Fig. 7
$ cd $ARTIFACT_ROOT/pin
$ ./download

$ # Build EGACS runtime library
$ cd $ARTIFACT_ROOT/ggc-ispc
$ make rt

$ # [Optional] Build CUDA runtime for Fig. 9
$ cd $ARTIFACT_ROOT/ggc-cuda
$ make rt
$ cd $ARTIFACT_ROOT/ggc-cuda-vm
$ make rt

$ # [Optional] Build GraphIt for Fig. 4 and Table X
$ cd $ARTIFACT_ROOT/graphit
$ ./build-artifact

$ # [Optional] Build Galois for Fig. 4 and Table X
$ cd $ARTIFACT_ROOT/galois
$ ./build-artifact
```

1) Inputs: Download `inputs.tar.xz` into

```
$ARTIFACT_ROOT/inputs
```

Next, extract the tarball

```
$ cd $ARTIFACT_ROOT/inputs
$ tar -xvf inputs.tar.xz
```

2) Detecting ISPC Target: We provide a script that automatically detects the best ISPC AVX target on current machine and sets up the Makefile.

```
$ cd $ARTIFACT_ROOT/ggc-ispc/skelapp-ispc
$ python detect_target.py
```

To specify a custom target other than the default one, please refer to "ISPC target" paragraph in "Experiment customization" section for details.

E. Experiment Workflow

The default experimental setup is intended for use on a single-socket multi-core 2-way SMT machine.

For a multi-socket machine, a CPU that has no SMT, or more than 2-way SMT, the script may have be modified to instruct EGACS on how to pin threads on the machine. Please refer to "Experiment customization" section for details.

After running the experiments, raw performance numbers, generated figures, and tables will be placed in the following directories, respectively.

```
$ARTIFACT_ROOT/ggc-ispc/eval/results
$ARTIFACT_ROOT/ggc-ispc/eval/plot/plots
$ARTIFACT_ROOT/ggc-ispc/eval/plot/tables
```

1) EGACS Results: Fig. 5 and 6.

```
$ cd $ARTIFACT_ROOT/ggc-ispc/eval
$ make -f Makefile.general all  # Run experiments
                                # about 16 hours
$ cd $ARTIFACT_ROOT/ggc-ispc/eval/plot
$ python plot_ispc.py           # Fig. 5
$ python plot_breakdown.py      # Fig. 6
```

2) Scalability Results: Partial Fig. 8 and 10. The original Fig. 8 and 10 consist of results from 3 different machines. The artifact scripts generate results for the machine on which EGACS is running. The default setup assumes a 8-core 16-thread single-socket Linux machine. Otherwise please refer to "Experiment customization" section on how to modify the script. (`Makefile.thread`).

```
$ cd $ARTIFACT_ROOT/ggc-ispc/eval
$ make -f Makefile.thread all   # 16 hours
$ cd $ARTIFACT_ROOT/ggc-ispc/eval/plot
$ python plot_thread_local.py   # partial Fig. 8
$ python plot_smt_local.py      # partial Fig. 10
```

3) AVX-Target Results: Fig. 7. Intel Pin required.

```
$ cd $ARTIFACT_ROOT/ggc-ispc/eval
$ make -f Makefile.target all           # 12 hours
$ make -f Makefile.target.icount all    # 4 hours
$ cd $ARTIFACT_ROOT/ggc-ispc/eval/plot
$ python plot_target_local.py           # Fig. 7
```

4) GPU Results: Fig. 9. CUDA 10.0 capable GPU required for running GPU benchmarks.

```
$ cd $ARTIFACT_ROOT/ggc-cuda/eval
$ make all                  # 1 hour
$ cd $ARTIFACT_ROOT/ggc-cuda-vm/eval
$ make all                  # 1 hour
$ cd $ARTIFACT_ROOT/ggc-ispc/eval/plot
$ python plot_gpu.py        # Fig. 9
```

5) Ligra/GraphIt/Galois Results: Fig. 4 and Table X. Cilk and OpenMP required.

```
$ cd $ARTIFACT_ROOT/ligra/eval
$ make perf_all             # 2 hours
$ cd $ARTIFACT_ROOT/graphit/eval
$ make perf_all             # 2 hours
$ cd $ARTIFACT_ROOT/galois/eval
$ make perf_all             # 1 hour
$ cd $ARTIFACT_ROOT/ggc-ispc/eval/plot
$ python plot_compare.py    # Fig. 4
$ python table_raw.py       # Table X
```

F. Evaluation and Expected Results

Reference results are placed in `ref` directory. Figures in the paper can be regenerated using the scripts we provide. Generated figures are placed in `ref/plots`. Tables are placed in `ref/tables`.

```
$ cd $ARTIFACT_ROOT/ref
$ python plot_compare.py        # Fig. 4 (Intel)
$ python plot_compare_amd.py    # Fig. 4 (AMD)
$ python table_raw.py           # Table X (Intel)
$ python table_raw_amd.py       # Table X (AMD)
$ python plot_ispc.py           # Fig. 5
$ python plot_breakdown.py      # Fig. 6
$ python plot_target2.py        # Fig. 7
$ python plot_thread2.py        # Fig. 8
$ python plot_gpu.py            # Fig. 9
$ python plot_smt2.py           # Fig. 10
$ python table_uvm.py           # Table IX
$ python table_vm.py            # Table IX
```

G. Experiment Customization

The experiments are fully customizable. For example, users can enable/disable the desired IrGL optimizations, set number of threads and pinning policy, and change ISPC AVX target.

1) IrGL Optimizations: Predefined optimization combinations are in `ggc-ispc/bmks/Makefile.ispc`. Users can simply add a Makefile target following the existing ones as example. Next, users can run those targets from `ggc-ispc/eval/Makefile.general` by simply appending the desired targets to `ALL_CONFIGS`.

2) Number of Threads and Pinning Policy: This can be changed in our evaluation Makefile by setting `TASK` variable in `ggc-ispc/eval/Makefile.general`. Default value is `0-0`, meaning our EGACS will determine number of threads automatically. There are two fields in this variable: the first field sets total number of worker threads. The second field sets the distance between two logical CPU IDs on which two consecutive worker threads are pinned. For example, `2-1` instructs EGACS to create 2 worker threads: thread #0 is pinned on CPU #0, thread #1 on CPU #1. `4-2` instructs EGACS to create 4 worker threads: thread #0 is pinned on CPU #0, thread #1 on CPU #2, thread #2 on CPU #1, and thread #3 on CPU #3. Correctly setting the value of this variable is important for scalability results (`Makefile.thread`).

3) ISPC Target: Intel ISPC compiler supports a wide range of targets. Possible targets include `avx512skx-i32x16` for Skylake-X CPUs, `avx512knl-i32x16` for Xeon Phi CPUs, and `avx2-i32x16` for generic AVX2 CPUs. To specify a custom target, there are two possible ways. First, editing

```
$ARTIFACT_ROOT/ggc-ispc/skelapp-ispc/SAMakefile
```

and adding the following line after line 28

```
TARGET = avx512skx-i32x16
```

Or, setting `CUSTOM_TARGET` variable when running evaluation commands. For example,

```
$ cd $ARTIFACT_ROOT/ggc-ispc/eval
$ CUSTOM_TARGET=avx2-i32x16 make -f Makefile.general
```

will run main EGACS experiments with AVX2x16 target.

REFERENCES

[1] M. Pharr, "The story of ispc: origins (part 1)," Apr 2018, (Accessed on 08/27/2020). [Online]. Available: https://pharr.org/matt/blog/2018/04/18/ispc-origins.html

[2] M. Pharr and W. R. Mark, "ispc: A spmd compiler for high-performance cpu programming," in *2012 Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–13.

[3] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[4] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: mining peta-scale graphs," *Knowledge and information systems*, vol. 27, no. 2, pp. 303–325, 2011.

[6] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[7] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, "A flexible open-source toolbox for scalable complex graph analysis," in *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM, 2012, pp. 930–941.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[9] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013, pp. 1–12.

[10] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg, *Practical graph analytics with apache giraph*. Springer, 2015, vol. 1.

[11] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *arXiv preprint arXiv:1204.6078*, 2012.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[14] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan, "Managing large graphs on multi-cores with graph awareness," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 41–52.

[15] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[16] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 211–222.

[17] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 349–362.

[18] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[19] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 1–19.

[20] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 521–532, 2015.

[21] T. Augustine, J. Sarma, L.-N. Pouchet, and G. Rodríguez, "Generating piecewise-regular code from irregular structures," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 625–639.

[22] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout, "Sparse computation data dependence simplification for efficient compiler-generated inspectors," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 594–609.

[23] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[24] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. Pereira, "Divergence analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 4, pp. 1–36, 2014.

[25] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanovic, "Exploring the design space of spmd divergence management on data-parallel architectures," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 101–113.

[26] A. W. Appel and A. Bendiksen, "Vectorized garbage collection," *The Journal of Supercomputing*, vol. 3, no. 3, pp. 151–160, 1989.

[27] L. Chen, X. Huo, B. Ren, S. Jain, and G. Agrawal, "Efficient and simplified parallel graph processing over cpu and mic," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 819–828.

[28] M. Paredes, G. Riley, and M. Luján, "Breadth first search vectorization on the intel xeon phi," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 1–10.

[29] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using simd instructions," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1587–1602.

[30] J. Zhang, Y. Lu, D. G. Spampinato, and F. Franchetti, "Fesia: A fast and simd-efficient set intersection approach on modern cpus," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1465–1476.

[31] H. Inoue, M. Ohara, and K. Taura, "Faster set intersection with simd instructions by reducing branch mispredictions," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 293–304, 2014.

[32] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using simd instructions." *ADMS@ VLDB*, vol. 1, p. 8, 2011.

[33] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 246–260, 2018.

[34] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1214–1225, Jul. 2015.

[35] L. Wang, L. Zhuang, J. Chen, H. Cui, F. Lv, Y. Liu, and X. Feng, "Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 276–289, 2018.

[36] C. Xu, K. Vora, and R. Gupta, "Pnp: Pruning and prediction for point-to-point iterative graph analytics," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 587–600.

[37] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 301–316.

[38] A. Mazloumi, X. Jiang, and R. Gupta, "Multilyra: Scalable distributed evaluation of batches of iterative graph queries," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 349–358.

[39] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference*. IEEE, 2015, pp. 403–412.

[40] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 293–304.

[41] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun, "The graph based benchmark suite (gbbs)," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2020, pp. 1–8.

[42] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 456–471.

[43] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 293–302.

[44] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with graphit," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 158–170.

[45] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 752–768.

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

[47] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 345–354.

[48] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.

[49] U. Cheramangalath, R. Nasre, and Y. Srikant, "Falcon: A graph manipulation language for heterogeneous systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–27, 2015.

[50] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–12.

[51] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," *Version 0.3. 0*, vol. 35, 2012.

[52] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[53] J. Soman, K. Kishore, and P. Narayanan, "A fast gpu algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[54] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 185–195.

[55] C. da Silva Sousa, A. Mariano, and A. Proença, "A generic and highly efficient parallel variant of boruvka's algorithm," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015, pp. 610–617.

[56] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 349–359.

[57] A. Polak, "Counting triangles in large graphs on gpu," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 740–746.

[58] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens, "Dynamic graphs on the GPU," in *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2020, May 2020, pp. 739–748. [Online]. Available: https://escholarship.org/uc/item/48j4k7np

[59] L. Wang and J. D. Owens, "Fast bfs-based triangle counting on gpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[60] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[61] K. Meng, J. Li, G. Tan, and N. Sun, "A pattern based algorithmic autotuner for graph processing on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 201–213.

[62] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[63] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, and R. Gupta, "Cumas: Data transfer aware multi-application scheduling for shared gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[64] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 39–50.

[65] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[66] C. E. Leiserson, "The cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[67] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[68] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[69] A. Magni, C. Dubach, and M. F. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–11.

[70] J. A. Stratton, S. S. Stone, and W. H. Wen-mei, "Mcuda: An efficient implementation of cuda kernels for multi-core cpus," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 16–30.

[71] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[72] R. Zheng, "Efficient execution of graph algorithms on cpu with simd extensions," Master's thesis, University of Rochester, Dec. 2020.

[73] T. Sorensen, S. Pai, and A. F. Donaldson, "One size doesn't fit all: Quantifying performance portability of graph applications on gpus," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 155–166.

[74] T. NVIDIA, "P100 gpu," *Pascal Architecture White Paper*, 2016.