Journal of Parallel and Distributed Computing 148 (2021) 69–82

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Graph-Waving architecture: Efficient execution of graph applications on GPUs

Ayse Yilmazer-Metin

Istanbul Technical University, Istanbul, Turkey

ARTICLE INFO

Article history:
Received 11 November 2019
Received in revised form 23 August 2020
Accepted 15 October 2020
Available online 27 October 2020

Keywords:
GPGPU
GPU microarchitecture
Graph application
Scalar waving
SIMD efficiency

ABSTRACT

Most existing graph frameworks for GPUs adopt a vertex-centric computing model where vertex to *thread* mapping is applied. When run with irregular graphs, we observe significant load imbalance within SIMD-groups using vertex to thread mapping. Uneven work distribution within SIMD-groups leads to low utilization of SIMD units and inefficient use of memory bandwidth. We introduce *Graph-Waving (GW)* architecture to improve support for many graph applications on GPUs. It uses vertex to *SIMD-group* mapping and Scalar-Waving as a mechanism for efficient execution. It also favors a narrow SIMD-group width with a clustered issue approach and reuse of instructions in the front-end. We thoroughly evaluate GW architecture using timing detailed GPGPU-sim simulator with several graph and non-graph benchmarks from a variety of benchmark suites. Our results show that GW architecture provides an average of 4.4x and a maximum of 10x speedup with graph applications, while it obtains 9% performance improvement with regular and 17% improvement with irregular benchmarks.

1. Introduction

Graphs are used by a great number of real-world applications from a wide range of domains. Many graph applications use quite large graphs which makes them good candidates for acceleration. GPUs provide immense power-performance efficiency when executing data parallel applications. Yet, accelerating graph algorithms on GPUs remains as a challenge.

GPU computing exploits data-parallelism and utilizes SIMD execution for efficiency. Threads are bundled into groups known as *warps* in CUDA terminology and *wave-fronts* in OpenCL terminology.[1],[2] *Warps* share a common PC and they execute the same instructions in synchronization on a set of pipelined *SIMD processing elements* (i.e. a SIMD unit). A common front-end is shared across the SIMD processing elements. This execution model works well for *GPU-friendly* applications having uniform control-flows and regular memory access patterns. However, many applications with arbitrary control-flows and irregular memory access patterns suffer from low utilization of SIMD processing elements and inefficient use of memory bandwidth [16]. Unfortunately, graph applications fall under this category due to their data-dependent (dynamic) characteristics [6,52].

Most of the existing graph frameworks for GPUs adopt a *vertex-centric* computing model [35] where programmers express the graph algorithms as a set of operations that are applied on vertexes.[3] Typically, *a vertex* is mapped to *a SIMD-thread*. When the amount of work associated with each vertex differs across the vertexes, then we observe uneven work distribution within a warp. That results in low-utilization of SIMD processing elements. To deal with the *intra*-warp work imbalance, prior studies [6,22] suggested using *vertex* to *warp* mapping. When vertex to warp mapping is used, the work performed on the edges of a vertex gets distributed over the threads within a warp.[4] Vertex to warp mapping aids reducing *intra*-warp work imbalance. However, without adopting the GPU micro-architecture, it cannot help much with low-utilization of SIMD processing elements.

In this work, we aim to improve the utilization of SIMD processing elements when running graph applications on GPUs. To do so, we leverage the idea of vertex to warp mapping and *scalarized* vertex-centric parallel graph computing on GPUs and introduce the *Graph-Waving* architecture to better support scalarized vertex-centric graph computing while maintaining the efficiency of SIMD execution for regular GPU applications.

When using vertex to warp mapping on existing GPU architectures, some computations on a vertex become redundant within a warp, resulting in inefficient use of SIMD processing elements. To eliminate this redundant computations, we adopt *Scalar-Execution (SE)* which we proposed in our prior work [56].

E-mail address: yilmazer.ayse@gmail.com.

[1] CUDA and OpenCL are two commonly used General Purpose GPU (GPGPU) computing frameworks.

[2] In this paper, we will be using term *warp* for bundled thread groups.

[3] Because of their higher synchronization requirements, *edge-centric* graph computing models are not preferred on GPUs.

[4] This model offers higher concurrency similar to what the *edge-centric* graph computing offers.

https://doi.org/10.1016/j.jpdc.2020.10.005
0743-7315/© 2020 Elsevier Inc. All rights reserved.

We call this model as *scalarized* vertex-centric parallel graph computing. In this model, programmers are allowed to annotate the redundant computation as *scalar operations* to fully utilize the *Scalar-Execution*.

When adopting the existing GPU architecture to scalarized vertex-centric graph computing, we employ *Scalar-Waving (SW)* that dynamically groups scalar operations possessing the same PC and executes them together as a *Scalar-wave* on SIMD pipelines for improved utilization of SIMD processing elements [56]. When customizing Scalar-waving to scalarized vertex-centric graph computing, we face several challenges. We observe that irregular applications favor a narrow SIMD-width and SIMD-units. Therefore, our design employs narrow SIMD-units but with a clustered issue logic to maintain the benefits of larger issue width with regular applications. Narrow SIMD units help with underutilization of SIMD processing elements when the vertexes have low degrees to fully utilize the lanes of SIMD-units. However, narrow SIMD-unit widths may lead to reduced front-end efficiency and increased pressure in the front-end (in the fetch and decode stage). This could significantly harm the performance of regular applications whose performance is tightly coupled with front-end efficiency. To address this problem, our proposed architecture leverages instruction re-use to eliminate redundant fetching and decoding of instructions across the warps and the iterations of loops. We call this new graph friendly architecture as *Graph-Waving* (GW) architecture.

The contributions of this work can be outlined as follows.

- With an analytic approach, we show that vertex-centric graph computing could benefit from *vertex to warp mapping* to reduce unbalanced work distribution within a warp and uncoalesced memory accesses, but at the same time, vertex to warp mapping introduces redundant computation within the warp.
- We suggest that we can eliminate this redundant computation with scalarized vertex-centric graph computing and *Scalar-Waving* architecture. Using scalarized vertex-centric graph computing, programmers are allowed to take advantage of *Scalar-Execution* and annotate the redundant computation as *scalar* to fully utilize the *Scalar-Waving* architecture. Scalar-Waving enables efficient use of hardware resources when performing scalar execution.
- We show that while existing modern GPUs favor wide SIMD-widths for front-end efficiency with highly regular applications, graph applications using scalarized vertex-centric graph computing could benefit more from narrow SIMD-widths. We argue that while having narrow SIMDs, we can still reserve the benefits of wider issue. As a novel design, we introduce our narrow SIMD-units with clustered issue logic.
- We also show that having narrow SIMD-groups could create pressure in the front-end and harm the performance. We propose to overcome this issue by leveraging the re-use of fetched and decoded instructions across the warps and iterations of loop bodies. We introduce our new GPU front-end design.
- We combine all these ideas and introduce *GW architecture* which orchestrates an architecture combining Scalar-Waving architecture, narrow SIMDs with clustered issue logic, and a front-end with decoded instruction reuse. We thoroughly evaluate the scalarized vertex-centric graph computing with GW architecture and demonstrate that our design significantly improves performance for graph applications while providing better or sustained performance for regular applications.

The remainder of this paper is organized as follows. First, in Section 2, we briefly describe GPU execution model and our baseline GPU micro-architecture. In Section 3, we explain the application of vertex-centric data parallel graph computing on GPUs and discuss its inefficiencies in Section 4. Next, we introduce scalarized vertex-centric parallel graph computing in Section 5 and we present Graph-Waving architecture combining three ideas in Section 6. Finally, in Section 8, we provide the details of our evaluation methodology and our benchmarks, present/discuss our simulation results. In the end, we review the related work in Section 9 and conclude in Section 10.

2. GPU execution model and baseline GPU microarchitecture

In this study, we baseline our work on a GPU model similar to NVIDIA TeslaC2050 GPUs that are based on NVIDIA's Fermi architecture [39].[5]

Our baseline GPU architecture is composed of an array of *multi-threaded SIMD processors*. Each multi-threaded SIMD processor accommodates a set of on-chip hardware execution contexts to execute a set of *thread blocks*. A thread block is divided into *warps*. Threads in a warp share a common PC and execute same instructions in lock-step until reaching to a *conditional branch*. When the threads in a warp do not take the same path from a conditional branch, then a *branch divergence* occurs. One way of handling branch divergence in SIMD execution is *serialized execution* of branches. Each path from a conditional branch is executed serially by enabling/disabling the threads depending on their branch outcome. During serialized execution, not all of the SIMD processing elements are utilized due to disabled threads following the alternative path. Each warp has an *active mask stack* to support conditional execution. Fig. 1 shows the high-level organization of a multi-threaded SIMD processor in our baseline GPU architecture.

Each multi-threaded SIMD processor has a back-end that consists of a set of fully-pipelined *SIMD-units* to execute warps in an interleaved manner. A common front-end is shared by SIMD-units and this front-end is responsible for supplying SIMD-units with instructions to execute. At each cycle, the *fetch unit* issues an instruction request for a selected warp. Once an instruction is fetched, it is decoded and checked for dependencies in *Scoreboarding* unit. Decoded instructions are stored in *Instruction Buffer* slots. There are dedicated instruction slots in instruction buffer for each warp. Scoreboard entries are organized to be accessed using the *warp-id (wid)*. At each cycle, two *schedulers* select and dispatch instructions for two warps. When an instruction is dispatched for execution, an *operand collector* unit is assigned for collecting input operands from a large multi-banked *register file*. In the back-end, there exist 32 lane *SP units* and *special function units (SFUs)*. *Load/store (LD/ST) units* handle accesses to memory through an L1 data cache. *Address coalescing* is performed to merge accesses from different SIMD-lanes. Shared memory handles *bank conflicts* by processing them in a serialized manner.

3. Typical vertex-centric parallel graph computing on GPUs

In this paper, we use a well-known graph algorithm called *Pagerank* as an example for our discussions.[6] This algorithm is based on the idea of link popularity where the incoming links

[5] We are aware that there have been many improvements in modern GPU architecture since NVIDIA's Fermi architecture, but the problems we focus on this paper continues to uphold to newer GPU architectures. These problems and the solutions we propose are not limited with NVIDIA's GPUs, they also applicable to GPUs from other vendors.

[6] Pagerank is developed and used by Google [43].

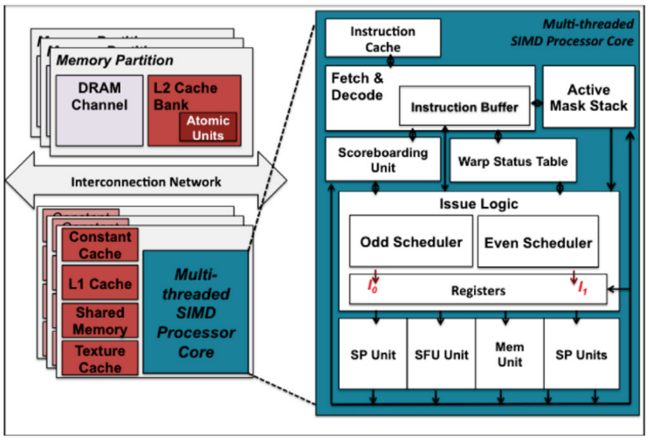


Fig. 1. High-level organization of our baseline GPU model and *multi-threaded SIMD processor*.

for a document determines the document's general importance. Pagerank is an iterative algorithm where a document's rank is calculated recursively by the ranks of other documents linked to it.

There are many possible implementation of Pagerank algorithm. We use the one based on Pannotia's [6] implementation. It is implemented with three CUDA kernels. It makes use of a widely used graph representation known as Compressed Sparse Row (CSR) format [46]. The input graph is represented with an array of *vertexes*, *edges*, and *ranks*. For each vertex, a pointer to list of outgoing edges is stored in array *vertexes*. Destinations of outgoing edges are stored in consecutive elements of array *edges*. The last element in array *vertexes* is a dummy vertex to denote the length of array *edges*. Initially, the rank for each vertex is set with a value of 1 divided by the total number of vertexes (1/*number_of_vertexes*()). The main computation part of the page rank algorithm includes a main loop to execute two kernels. These two kernels are called repeatedly until convergence happens or the loop count reaches to a predetermined number of iterations.

In Listing 1, we show the kernel that performs the calculation of new rank values for each vertex (implementation in CUDA). In this code, each vertex sends its current rank to all of its neighbors. To do so, each edge is visited in a for loop to push the updates. After receiving the update values sent from neighbors, each vertex sums them up to calculate its new rank. This implementation uses the vertex-centric parallel computation where vertex to a SIMD-thread mapping is applied.

```
1  __device void update(float *addr,  const float upval);
2  __global void ComputeRanks(int *vertexes, int *edges, float *
       cranks, float *nranks, int N) {
3   // get the global thread id to identify the assigned vertex
        id
4   int vid = blockIdx.x * blockDim.x + threadIdx.x;
5   if (vid < N) {
6    int start = vertexes[vid], end = vertexes[vid+1];
7    float upval = cranks[vid]/(end   start);
8    //navigate the neighbor list
9    for (int e = start; e < end; e++)
10    update(&nranks[edges[e]], upval);
11  }
12 }
```

Listing 1: CUDA implementation of ComputeRanks kernel in Pagerank algorithm

While we use *Pagerank* algorithm in this paper as a working example, the discussions that we carry out here apply to many other graph algorithms, too. We observe a common, fundamental computation pattern in these kernels: the computations on vertexes are performed while expanding edge list and gathering information from neighboring vertexes, such as pushing updates to neighboring vertexes as seen in *ComputeRanks* kernel. This pattern can be summarized with a pseudo code shown in Fig. 2. We will be using this fundamental computation pattern for our discussions as the representative of *ComputeRanks* kernel in Pagerank algorithm and many other similar graph kernels. In this paper, we refer to this computation pattern as *edge-list-expanding* computation pattern.[7]

4. Inefficiencies of vertex-centric graph parallel computing on GPUs

Typical implementation of vertex-centric parallel graph kernels with *edge-list-expanding* computation pattern suffers from serious performance penalties when the input graph is highly irregular, having many vertexes with divergent degrees (i.e., when the number of edges per vertex varies). We discuss two primary sources of these performance penalties using a running example shown in Fig. 2.

4.1. Intra-warp load imbalance and under-utilization of SIMD units

Intra-warp load imbalance will be observed when work is distributed unevenly across the threads of a warp. Unfortunately, uneven load distribution results in under-utilization of SIMD-units and may severely hurt the performance. In vertex-centric parallel graph computing, load imbalance could happen within warps when the main computation of a kernel has to be performed only for a subset of vertexes. It could also happen due to fact that some vertexes could have higher degrees (i.e., the number of edges) than others. For each vertex, edges are visited in iteration while performing some computation on each edge. The length of edge lists (i.e., the range of *e* in Line 9 of Listing 1) may vary across different vertexes. Thus, the threads in a warp with shorter edge lists have to wait for those SIMD-threads in the same warp with longer edge lists to finish executing the *for loop*.[8] This case is illustrated in Fig. 2(e).

Unfortunately, most real world graphs are known to be pretty irregular in terms of degrees. In Fig. 3(a), we show distribution of vertexes based on their degrees for such a graph called coAuthors [13] which is used to run Pagerank. As seen from this figure, most of the vertexes have a low degree (average degree is 17) while there are some vertexes with very high degrees (some have more than 300 degrees). With irregular graphs, most of the time it is highly possible that only a limited number of SIMD threads in a warp are active and only some of the SIMD processing elements are utilized. To understand the significance of load imbalance within the warps, we looked at the utilization of the SIMD units with a set of well known graph algorithms.[9] Fig. 4(a) shows a breakdown of execution cycles as a function of active SIMD-threads in warps to illustrate the utilization of the SIMD processing elements. Our evaluation results prove that graph applications use SIMD pipelines inefficiently due to load imbalance. We observe that SIMD units are not fully utilized for 45% or more of the execution cycles. Furthermore, for 30% or more of the execution cycles, only 8 or less of the SIMD processing elements happen to be active.

[7] Since concurrency is at edge-level in edge-centric graph computing, we do not observe this pattern in edge-centric computing.

[8] Remember that, in a typical vertex-centric parallel graph algorithm, kernels are structured such that each vertex is assigned to a SIMD-thread.

[9] The details of our benchmarks and evaluation methodology for this study are given in 8.1.

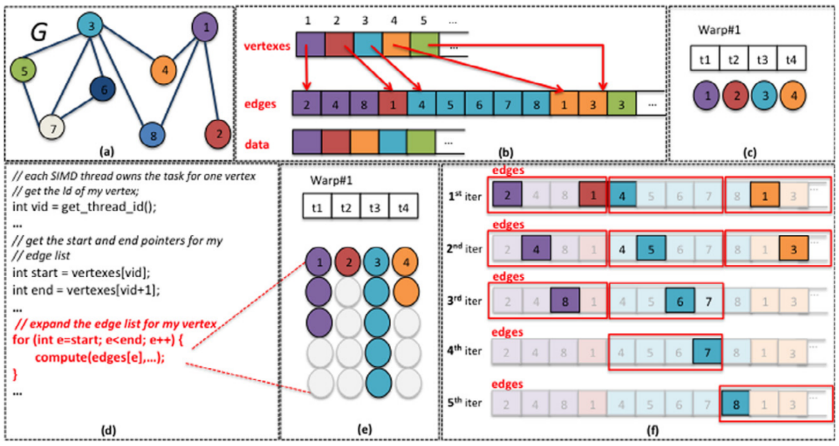

```
// each SIMD thread owns the task for one vertex
// get the Id of my vertex;
int vid = get_thread_id();
...
// get the start and end pointers for my
// edge list
int start = vertexes[vid];
int end = vertexes[vid+1];
...
// expand the edge list for my vertex
for (int e=start; e<end; e++) {
    compute(edges[e],...);
}
...
```

Fig. 2. Typical vertex-centric parallel graph computing on GPUs and illustration of its shortcomings. (a) An example input graph G, (b) data structures for graph G, (c) example mapping of vertexes to SIMD threads, (d) the fundamental computation pattern observed in many graph kernels, (e) SIMD lane utilization and (f) accesses to array *edges* while *expanding the edge list* to perform computations.

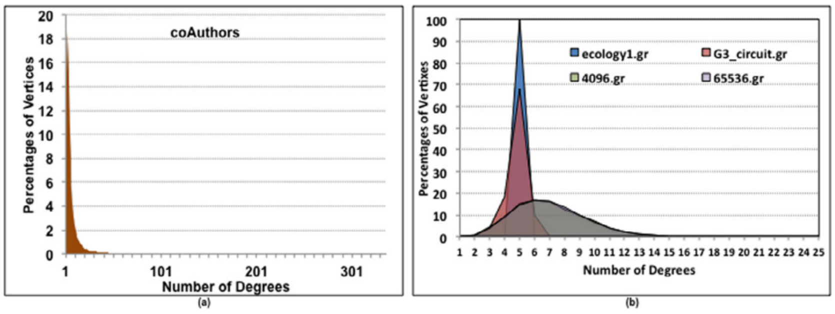


Fig. 3. Distribution of vertexes based on their degrees **(a)** A graph with diverse degrees. **(b)** Graphs with small degrees.

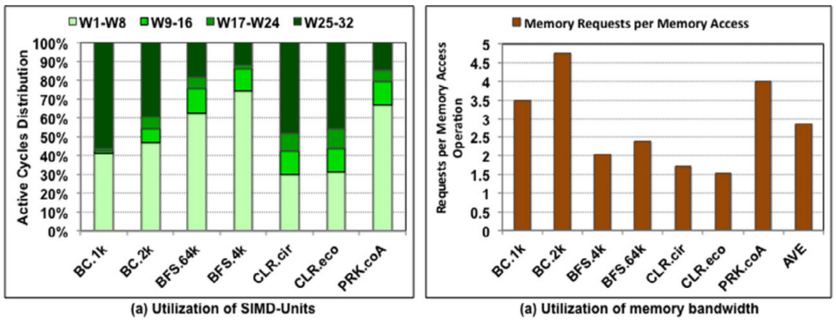
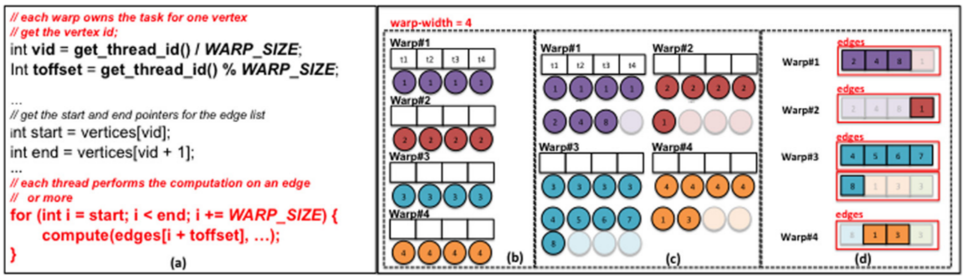


Fig. 4. **(a)** Utilization of SIMD-units and **(b)** memory bandwidth consumption during the execution of various graph benchmarks. *W1–W8* means 1 to 8 of SIMD-threads are active in a cycle, and a similar scheme applies for *W9–W16*, *W17–W24*, and *W25–W32*. Memory bandwidth utilization given as number of memory requests per memory access.

4.2. Non-coalesced memory accesses

In typical vertex-centric parallel graph computing, additional performance degradation comes from non-coalesced memory accesses and memory divergence. A memory operation generates multiple memory transactions, when the accesses of SIMD-threads in the same warp cannot be coalesced into a single request. *Memory divergence* arises when the threads of a warp encounter different memory access time due to data cache misses or memory bank conflicts as a result of uncoalesced memory accesses. For instance, the edge lists expanded by the threads of a warp could be dispersed to several blocks of memory. Fetching these data may take several memory transactions. When performing memory operations, the entire warp waits until all threads in the warp finish their memory accesses. As an example, in Fig. 2, we show the memory requests that are generated to access to the array *edges* by `Warp#1` during the iterations of the *for loop*.

To see if the amount of non-coalesced memory accesses is significant in graph applications, we looked at average number of memory requests per memory access in our graph benchmarks. Fig. 4(b) shows the amount of average memory requests per memory access for our graph benchmarks. Observing 2.85 memory requests per memory access on the average, we conclude that non-coalesced memory accesses are quite significant for graph applications. Indeed, non-coalesced memory accesses and memory divergence are common problems with most data intensive irregular applications like graph algorithms. Non-coalesced memory accesses increase the memory bandwidth pressure, may result in memory divergence and therefore result in performance degradation and higher power consumption.



```
// each warp owns the task for one vertex
// get the vertex id;
int vid = get_thread_id() / WARP_SIZE;
Int toffset = get_thread_id() % WARP_SIZE;

...
// get the start and end pointers for the edge list
int start = vertices[vid];
int end = vertices[vid + 1];
...
// each thread performs the computation on an edge
//   or more
for (int i = start; i < end; i += WARP_SIZE) {
      compute(edges[i + toffset], ...);
}
```

Fig. 5. Mapping a vertex to a warp. (a) Transformed code to apply vertex-to-warp mapping and distributing computation on edges to SIMD-threads. (b) Example mapping of vertexes to warps. (c) SIMD-lane utilization (d) and accesses to array *edges* while expanding edge list of a vertex.

5. Scalarized vertex-centric parallel graph computing

To overcome the shortcomings of typical vertex-centric parallel graph computing with *edge-list expanding* computation, we leverage *scalarized* vertex-centric parallel graph computing. Scalarized vertex-centric parallel graph computing combines two ideas: (1) mapping *a vertex to a warp* and then, (2) utilizing *Scalar-execution (SE)* [56]. Vertex-to-warp mapping aims to overcome intra-warp load imbalance but it causes redundant computations within the warps. Scalar execution eliminates redundant computation within a warp and it requires some minor modifications to baseline GPU microarchitecture.

Scalarized vertex-centric parallel graph computing includes these steps: (1) Programmer associates a vertex to a warp and constructs the algorithm such that the work on edges gets distributed over the SIMD-threads of a warp, and (2) annotates *Scalar* code when implementing the algorithm, (3) then, the compiler marks each scalar instruction based on programmer's annotation and also using static *scalarization analysis*, (4) at the execution time, hardware utilizes *Scalar-execution(SE)* [56] when the instructions to execute are scalar instructions.[10]

5.1. Vertex-to-warp mapping

Mapping a vertex to a warp has two main advantages. We can improve (1) SIMD utilization and (2) memory access coalescing. When a vertex is mapped to a warp, the work on edge list of a vertex gets distributed over the SIMD-threads within the same warp and therefore each SIMD-thread in a warp gets almost the same amount of work. This improves the utilization of the SIMD-units and more of the memory operations become coalescable. We illustrate these two advantages of vertex to warp mapping approach in Fig. 5 using our running example. The example uses the same input graph, *G*, shown in Fig. 2. The computation model using vertex to warp mapping is summarized in Fig. 5(a). With this model, the utilization of SIMD-units and the memory accesses in the *for loop* are illustrated in Fig. 5(b) and Fig. 5(c), respectively. As seen in Fig. 5(c), we might still observe idle SIMD processing elements with *vertex to warp mapping*, however, under utilization of SIMD-units might happen only at the last iteration of the *for loop*. With vertex to warp mapping, intra-warp load imbalance (due to diverse degree distribution of vertexes) becomes less significant.

5.2. Scalar execution (SE) to eliminate redundant computations

The redundant computations are not specific to vertex-centric parallel graph computing with vertex to warp mapping. Indeed, redundant computations exist in many GPU kernels. Former studies showed that the amount of redundant computations could be significantly high for some applications [9,56]. We show the amount of redundant computations present in various GPU benchmarks in Fig. 6. We can see that 30% or more of the executed instructions perform redundant computation in 12 (more than half) of the benchmarks. On the average, approximately 34% of the computations are redundant across all of the benchmarks. These scalar instructions are redundantly executed on SIMD pipelines if there is no scalar hardware [1] present.

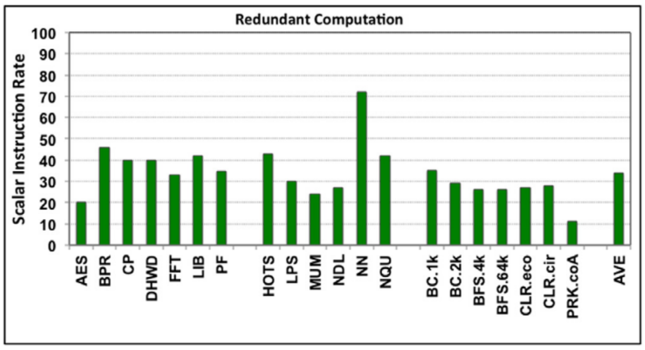


Fig. 6. Amount of redundant computation for various GPU benchmarks including our graph benchmark set.

We would like to utilize *Scalar-Execution* to eliminate the intra-warp redundant computations. SE requires static or dynamic identification of scalar instructions (described in Section 5.2.2).[11] Once we identify scalar instructions, we mark them as scalar with a flag in kernel binary. Then at the run-time, the *scalar flag* triggers *Scalar-Execution* on SIMD-units.

In this work, we employ static identification of scalar operations using programmer annotation and compiler analysis. We propose a few changes to Instruction Set Architecture (ISA) and a number of modifications to our baseline multi-threaded SIMD processor to support *SE*. Below we explain the techniques that we use for scalar identification and SE for scalarized vertex-centric parallel graph computing.

5.2.1. Programming for scalarized vertex-centric parallel graph computing

Scalarized vertex-centric parallel graph computing exposes *warps* (SIMD-groups) to programmers and warps become explicitly part of the programming model.[12] To expose warps in

[10] An instruction performing redundant computation within a warp is known as *scalar instruction* in the literature [9,56].

[11] In this work we do not consider dynamic identification of scalar instructions since it increases the hardware complexity and the power consumption. Extending this work with dynamic scalar identification can be considered as a future work.

[12] Programmers have already been using warp-aware programming for performance optimization even though warps are not explicitly exposed to the programmers.

the programming model, we introduce two new intrinsic — `get_warp_id()` and `get_simd_width()`. `get_warp_id` allows a SIMD-thread to query the ID of the warp that it belongs to and *get_simd_width* can be used to get the supported SIMD width by the GPU model being used.

```
__device void update(float *addr,  const float upval);
__global void ComputeRanks(int *vertexes, int *edges, float *
     cranks, float *nranks, int N) {

#pragma SCALAR
 /* get my vertex id, each warp owns the task for one vertex
      */
 int vid = get_warp_id();
 if (vid >= N) return;
 /* get the start and end pointers for the edge list for my
      vertex */
   int start = vertices[vid], end = vertices[vid+1];
   float upval = cranks[vid]/(end   start);
#pragma SCALAR_END
 /* get my thread local id within my group (warp) */
 int toffset = get_warp_local_id();
   /* expand the edge list for each vertex, if the simd_width
        is smaller
    * than the number of edges, then more than one iteration
        is required
    * by a warp */
   for (int e=start + toffset; e<end; e=e + get_simd_width())
    update(&nranks[edges[e]], upval);
}
```

Listing 2: CUDA implementation of ComputeRanks kernel with *scalarized* vertex-centric parallel graph computing approach.

Programmers can use `get_warp_id` to get the vertex-to-warp association. After having identified the associated vertex, each SIMD-thread works on a different edge until all edges of the vertex are processed. We provide modified code listing for *Pagerank*'s *ComputeRanks* kernel as shown in Listing 2 to illustrate the application of programming approach that we introduced with scalarized vertex-centric parallel graph computing.[13]

5.2.2. Identification of scalar operations

In this work, we use compiler annotations and compiler analysis. Programmers can annotate scalar regions in their code to have the compiler mark the instructions in that region as scalar instructions. Compiler can be easily extended to support scalar annotations and also marking scalar instructions with a bit flag in the binary. In Listing 2, we illustrate the use of compiler annotations to mark scalar regions. Our work also utilizes compiler analysis for extended identification of scalar operations. For this purpose, we use a compiler analysis technique based on divergence analysis [12]. We start with identifying an initial set of identical input operand values. These variables can be constants or the result of a value being broadcast to multiple instructions. Each variable that is identified as identical is tagged accordingly. Later, the analysis process traverses through the tags using data dependence analysis. An output variable is marked as identical if it is computed using only identical variables. Instructions possessing identical input operands generate identical results, and are marked as *scalar* instructions. After performing this analysis, we are left with a conservative set of identical variables and scalar instructions.

[13] When using scalarized vertex-centric parallel graph computing, if any data must be shared within a warp, the programmers can utilize the *shared memory*. Shared memory would be sufficient to provide efficient communication across the threads in a warp.

5.2.3. Architectural support for scalar execution on SIMD units

We can either use an undefined bit in ISA or extend the ISA to include an extra bit in each instruction's machine code to trigger Scalar-Execution. We also modify the *Instruction Buffer* so that each entry includes a single bit flag that will identify a scalar instruction at runtime.

6. Graph-waving architecture for scalarized vertex-centric parallel graph computing

The performance improvements of the SE and the scalarized vertex-centric parallel graph computing would be only limited without adapting existing GPU microarchitecture to it.[14] We propose modifications to baseline GPU architecture and introduce *Graph-Waving (GW)* architecture. GW combines three approaches: (1) it uses narrow SIMD-widths with clustered issue approach, (2) it employs decoded-instruction re-use, and (3) it extends and employs *Scalar-Waving (SW)* architecture [56]. In the remaining of this section, we explain modifications to ISA and changes to baseline GPU micro-architecture to implement Graph-waving SIMD architecture.

6.1. Narrow SIMDs with clustered issue

When using scalarized vertex-centric parallel graph computing, the utilization of the SIMD pipelines will be low, if the input graphs have large number of vertexes with low degrees. Fig. 3 shows the distributions of vertexes based on their degrees for five graphs that we use in our experiments. We see that most of the graphs have degrees less than 8 and this suggests that we employ narrower SIMD-widths for better utilization of SIMD-pipelines with graph applications.

Our proposed GW architecture supports 8-wide warps. We organize the available SIMD processing elements into 4 groups (i.e a SIMD-unit). Each group is 16-wide and executes a pair of an odd and an even numbered warp together. There is a dedicated scheduler for each of these four groups. Each scheduler tries to find a pair of odd–even warps with same PC to issue on 16-wide SIMD units. During the issue time we cluster odd and even warps with the same PC. However, when the control flow or memory divergence occurs, a warp in a pair is allowed to diverge and complete without waiting for other warp in the pair. As a result, a pair may go out of synchronization and a scheduler may not find a pair with same PC. In that case a ready odd or even warp will be selected to get scheduled alone. When that is the case, the idle part of the SIMD-units can be put into inactive mode for power savings. The four schedulers and 16-wide SIMD-units with aggregated issue logic can be seen in Fig. 7(a). At each cycle, 4 scalar/regular warp instructions get issued on SIMD-units.

6.2. Instruction re-use in front-end

While narrower SIMD-widths and scaled issue logic helps with under-utilization of SIMD-units under control flow divergence, the increased number of warps is likely to create pressure in the front-end. Pressure in the front-end could severely harm the performance of highly regular applications since their performance is tightly coupled with the front-end efficiency. To understand the impact of front-end pressure on performance, we performed a set of experiments with our benchmarks. In these experiments, we varied the SIMD-width and scaled the issue logic and SIMD-units in the back-end and looked how the performance changes. Fig. 8 shows the performance results from our experiments. Our results clearly show that the front-end pressure will cause a performance degradation for many of the benchmarks.

[14] The reader can refer to [56] for a more detailed study on performance improvements of SE.

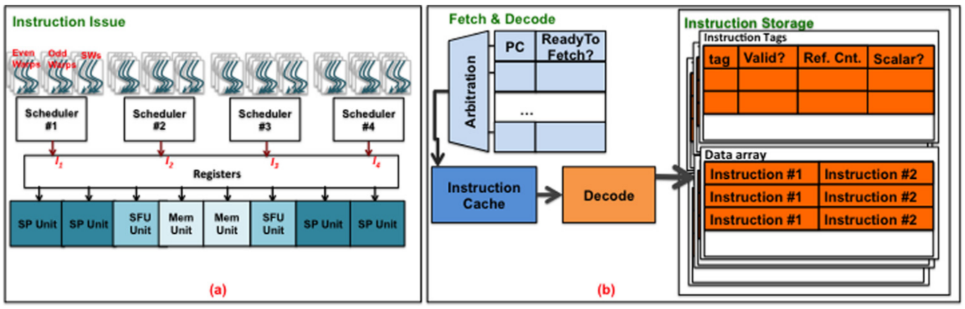


Fig. 7. Fetch, Decode, and Issue in GW architecture. (a) Organization of schedulers and SIMD-units in GW architecture; (b) Organization of Fetch & Decode Unit and Instruction Storage in the front-end.

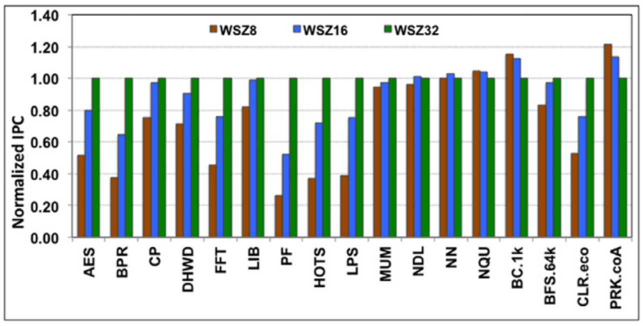


Fig. 8. Effect of front-end pressure on performance due to narrowed SIMD-width.

To overcome the pressure in the front-end, we exploit the redundancy in instruction fetch and decode and aim to reuse decoded instructions. In SIMD-execution, at any cycle it is highly likely to happen that many instructions across warps and across loop iterations are the same. As a result, most of the fetch and decode operations are redundant. We can take advantage of this redundancy in the front-end and re-use the already fetched and decoded instructions.[15] This requires modifications to fetch unit and re-organization of instruction buffer. In the new organization, instruction storage must be decoupled from warps. Fig. 7(b) illustrates the new *Instruction Storage* organization. Instruction Storage is organized in set associative way like a cache. Each entry in Instruction Storage includes a *tag* to identify the entry, a *Valid* bit, a *Reference Counter* and a *Scalar* bit indicating if the instruction is scalar. Each entry also includes details (opcode, operands, etc.) for two decoded instruction (*Instruction #1* and *Instruction #2* as shown in the figure). At a scheduling cycle, when a warp instruction is issued, the next PC is calculated for the issued warp. Then the Instruction Storage is checked to see if the next instruction already exists as fetched and decoded. If found, then the reference counter for this instruction is incremented and a pointer for this entry is inserted for the next PC pointer in the Warp Status Table (WST). If not found, then *ReadyToFetch* flag is set in WST to indicate that the warp needs instruction fetch. The Fetch Unit arbitrates over only the warps that have their *ReadyToFetch* flags are set. As far as there are available fetch buffers, a new instruction will be fetched at each cycle.

6.3. Efficient execution of scalar instructions

SE on SIMD pipelines utilizes (as explained in Section 5.2) only a single SIMD lane,[16] leaving all other lanes unused. These available lanes could be used if we can group a number of scalar instructions together into a batch and execute them in SIMD fashion. During the execution of a kernel, it is likely to have multiple warps ready to execute the same scalar instruction. Multiple warps executing the same scalar instruction (possessing the same PC) can be grouped into a *Scalar-wave* to execute together in SIMD fashion on the SIMD-pipelines. In our prior work [56], we introduced *Scalar-Waving (SW)* to dynamically form groups of scalar instructions to execute together for improved utilization of SIMD lanes.

It is possible to store the results from scalar operations as a single value —a *scalar value*, stored in one location for all threads in the warp, eliminating redundant storage of the same scalar value. We also describe how to store scalar values in a modified register file in Section 6.3.3.

To enable SW execution, the main modifications to our baseline multi-threaded SIMD processor consist of three main parts: (1) modification to the instruction buffer to differentiate scalar instructions, (2) adding a *Scalar-Wave Formation Unit (SWFU)* —a new unit that includes a *Scalar-Wave Status Table (SWST)* and manages the formation of scalar waves, and (3) modifications to the register file to provide efficient storage of scalar values. In Fig. 7, we see one bit *Scalar* flag indicating if the instruction is scalar. In Fig. 9, we show the newly added *SWF Unit*. In the following subsections we describe the formation and scheduling of Scalar-waves, as well as storage of scalar values in detail.

6.3.1. Scalar wave formation

When a SIMD instruction is decoded, it is inserted into buffers in Instruction Storage. If it is a scalar instruction, then the scalar flag (see 7(b)) in the buffer entry is set. Once register dependencies for the instruction are resolved, the *Scalar-Wave Status (SWST)* Table (shown in Fig. 9) is updated for it. The SWST is also checked to see if there is an existing *Scalar-Wave Entry* (SWE) with the same PC in SWST. If there is no match, but there is available room for a new entry, then an SWE is allocated for the scalar instruction. When an SWE exists with a matching PC, then there is no need to create a new SWE. Once an SWE is created or an existing SWE is found, then the corresponding bit in *Scalar-Wave Mask (SWM)* of SWE is set for associated warp. In the Instruction buffer, the *SW-Valid* flag is set to show that the instruction is associated with a Scalar-wave. A pending scoreboard entry is created to indicate dependencies associated with this instruction for the warp. If there is no matching entry found and a new entry cannot be allocated in SWST, then this scalar instruction waits until space becomes available in SWST.

6.3.2. Scheduling scalar-waves

At each cycle, each scheduler tries to find a ready warp pair to schedule next. If both warps in a pair are ready then they get scheduled together. If only odd or even part of any pair is

[15] Warps usually progress in synchronization and when they go out of synchronization the benefits of the *reuse* will be limited.
[16] SIMD lane is interchangeable used with SIMD processing element.

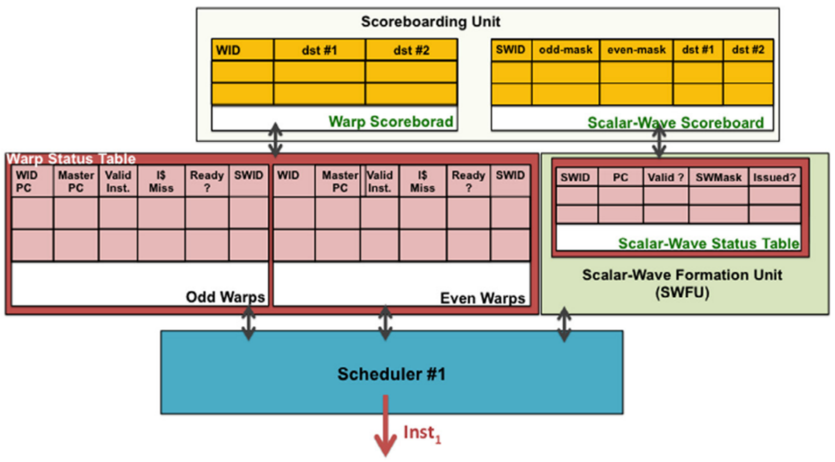


Fig. 9. *Warp Status Table*, *Scalar-Wave Formation Unit* including its *Scalar-Wave Status Table* and Scoreboards in GW architecture.

ready, pair gets scheduled to run only the active part. When the scheduler cannot find any ready warp pair to schedule, then the SWST is interrogated to find the oldest Scalar-Wave available to schedule next. When the selected Scalar-Wave is scheduled, the *Issued* flag in the SWST is set to 1. Scalar-Wave ID (SWID) and *Scalar-Wave Mask* are passed to scoreboarding unit so that a scoreboard entry is created in Scalar-Wave Scoreboard. When an issued Scalar-Wave finishes its execution, the entry for the Scalar-Wave in the SWST is released. The SWID is also passed to the scoreboard unit for clearing the register statuses in Scalar-Wave Scoreboard.

6.3.3. Scalar values and register storage for scalar-waves

While grouping scalar operations, the results from scalar operations can also be organized as a *group* in the register file. In our GW architecture, each SIMD-unit has 16-issue SIMD-units. Therefore, a register entry can provide scalar operand values for 16 warps (one scalar value per warp). GW architecture reserves a unique static id, *Scalar-Wave ID (SWID)*, for each Scalar-wave. Similarly, we tag the grouped scalar register entries with a tag to differentiate them from non-scalar register entries. When a scalar register entry is accessed with an SWID, it provides operand values for all warps of corresponding Scalar-wave.

7. Hardware cost and power consumption

GW architecture modifies the scoreboard, instruction buffer/storage, scheduler, issue unit and adds Scalar-wave Formation Unit in baseline GPU architecture. We modify the accesses and management of Instruction Storage/Buffer but we do not increase the available storage for instructions in the Instruction Buffer/Storage. We believe that GW architecture does not increase the area significantly. GW helps with dynamic power consumption with reduced execution time, instruction fetch and decode. However, we should also consider increased static power consumption due to newly added hardware in the GW architecture.

In this work, we specifically focused on performance aspect of GW architecture and we consider the detailed study of area estimation and power consumption for GW architecture as a future work.

8. Evaluation

8.1. Experimental setup

We model our proposed hardware schemes for GW architecture modifying version 3.x of GPGPU-Sim [2]. We build the simulator using gcc version *4.4*. GPGPU-Sim is capable of running CUDA [40] and OpenCL [29] applications. We used various non-graph CUDA benchmarks and graph benchmarks for our work. Our benchmarks are compiled using NVIDIA's tool *nvcc* version *4.2*. When compiling with *nvcc*, we used the option *—nvcc-arch=sm_20*. In the following subsections, we provide more details on our benchmarks and configuration parameters of our simulations.

8.1.1. Benchmarks

The non-graph benchmarks that we used for this study are collected from Rodinia [7], Parboil [25,47] Benchmark Suites, NVIDIA CUDA SDK [41] and the set of benchmarks used by Bakhoda et al. [2]. We used graph benchmarks from Pannotia [6] and Rodinia [7] Benchmark Suites. We classified non-graph benchmarks into two categories depending on their utilization of SIMD lanes. The benchmarks that fail to utilize all SIMD lanes during 25% or more of their execution time are classified as *IRREGULAR* benchmarks. The remaining benchmarks are classified as *REGULAR* benchmarks. We provide the list of non-graph benchmarks and graph benchmarks in Table 1.

8.1.2. Simulation configurations

We use GPGPU-Sim to model a baseline GPU similar to NVIDIA's Fermi GPU architecture [39]. We performed our experiments with the following simulation configuration parameters, modeling our baseline GPU architecture (B32) and a Graph-Waving architecture (GW8).

Our baseline GPU architecture (B32) and GW8 architecture include 14 multithreaded SIMD processors with a clock rate of 1.15 GHz. Each multithreaded SIMD processor has a 48 KB L1 cache with 128 Byte line-size and 6-way associativity and a 16 KB shared memory. A 786 KB L2 cache with 128 Byte line size and 8-way associativity is shared between the multi-threaded SIMD processors. There are 6 memory partitions and each partition includes an L2 bank and a DRAM channel. Memory system is modeled to run at 1.5 GHz clock rate. Multi-threaded SIMD processors and device memory are connected through a crossbar with 1.15 GHz clock rate. Each multi-threaded SIMD processor employs two schedulers in baseline B32 architecture and 4 schedulers in GW8 architecture. A round-robin model is used for the scheduling of 32-wide warps in baseline B32 architecture. GW8 architecture employs a modified version of round-robin scheduling as explained in Section 6.3.1. In GW architecture, each scheduler selects a pair of 8-wide warps or 16-wide Scalar-Waves ready to execute.

Table 1
List of *REGULAR* and *IRREGULAR* benchmarks evaluated and characterized in this paper.

REGULAR benchmarks	*IRREGULAR* benchmarks	*GRAPH* benchmarks	Input set
AES Encrypt. (AES) [2]	HotSpots (HOTS) [7]	Betweenness Cent. (BC.1k) [5,6]	1k-128k
Backpropagation (BPR) [41]	3D Laplace Solver (LPS) [7]	Betweenness Cent. (BC.2k)	2k-1M
Coulombic Potential (CP) [25]	MUMmerGPU (MUM) [2]	Breadth-First Search (BFS.4k) [7,11]	4096
1D Discrete Haar Wavelet Decomp. (DHWD) [41]	Needleman–Wunsch (NDL) [7]	Breadth-First Search (BFS.64k)	65 536
Fast Fourier Transform (FFT) [41]	NN Digit Recogn. (NN) [4]	Graph Coloring (CLR.eco) [6,11]	ecologi1
LIBOR Monte Carlo (LIB) [2]	N-Queens Solver (NUQ) [2]	Graph Coloring (CLR.cir)	G3_circuit
Path Finder (PF) [7]		Pagerank (PRK.coA) [6,43]	coAuthors

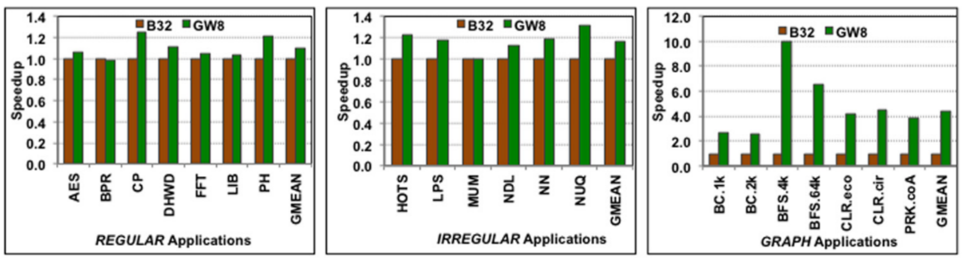


Fig. 10. Performance of GW8 and B32 architectures relative to B32 architecture for *REGULAR*, *IRREGULAR* and *GRAPH* applications.

8.2. Performance of graph-waving architecture

We start with presenting performance results of GW architecture. Fig. 10 shows the performance of GW8 architecture normalized to B32 architecture.

Using *REGULAR* benchmark set, we observe a 9% performance improvement on the average (geometric mean) with GW8 architecture. At the highest, CP and PATH obtain 25% and 21% speedups, respectively. DHWD obtains a smaller speedup (12%) and there is insignificant performance improvements for LIB, FFT. We see insignificant performance degradation for BPR.[17] We observe even higher performance improvements using *IRREGULAR* benchmarks with GW8 architecture, since irregular benchmarks can get more benefits from narrow SIMD-widths. The performance improvement of GW8 is 17% on the average for *IRREGULAR* benchmark set. NUQ, HOTS, NN, and LPS obtain speedups of 31%, 22%, 19% and 17%, respectively. We do not observe any performance improvement for MUM.

Graph applications are also characterized as irregular. Like irregular benchmarks, graph benchmarks also favor narrow SIMDs. Adding the benefits of scalarized graph computing model, GW8 architecture obtains good amount of performance improvements with *GRAPH* benchmarks. We observe around 4.41x performance improvement on the average with GW8 architecture. At most, we observe 10x and 6.5x speedups using BFS.4k and BFS.64k benchmarks, respectively.

8.3. In-depth analysis of graph-waving architecture performance

The performance of the GW architecture depends on several factors. In this section, we discuss these factors in detail.

[17] We will discuss the reasons for these performance results later in 8.3.

8.3.1. Performance benefits of scalarization and scalar-waving

Besides the number of scalar operations present in the code, there are other factors that affect the performance benefits of Scalar-Waving. At each cycle, a multi-threaded SIMD processor can issue an instruction or experience stalls due to downstream pipeline stages, memory dependencies or control dependencies. The number of issuable warps at a cycle depends on the occupancy of the SIMD processors and the dependencies that exist in the application. When scalar operations are issued as a wave, in the next few cycles the scheduler should be able to find warps to schedule (versus remaining idle). Otherwise, issuing the scalar operations as a group (wave) would not benefit performance since the processing units will be idle in later cycles. For our detailed performance analysis, we collected statistics to determine the number of *issuable warps* for each scheduling cycle. A warp is said to be issuable if the warp has a valid next instruction (no unresolved control dependency) and the next instruction has all its input operands ready (no data dependency).

The availability of issuable warps with scalar operations —*issuable scalars*, also affects the benefits of scalar waving. For this reason, we need to take a closer look at the number of issuable scalars during each cycle. In our detailed analysis study, we collected statistics to determine the number of issuable scalars for each scheduling cycle. A scalar instruction is considered to be issuable if the instruction is a valid scalar instruction and it has all its input operands ready (no data dependency).

Besides the number of *issuable warps* and *issuable scalars*, we also need to consider the number of scalars in each issued Scalar-Wave —the packing factor of scalar waves. In Fig. 12, we show the distribution of processor cycles where the number of *issuable warps* is 0 *(w0)*, 1 *(w1)*, 2–3 *(w2-3)*, 5–8 *(w5–8)*, 9–16 *(w9–16)*, or more than 16 *(>w16)* and in Fig. 11, we show the distribution of processor cycles, where the number of *issuable scalars* is: 0

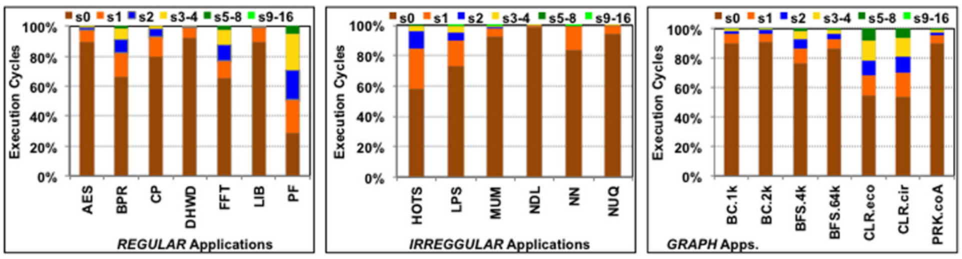


Fig. 11. Cycles breakdown based on number of *issuable scalars* for *REGULAR*, *IRREGULAR* and *GRAPH* applications.

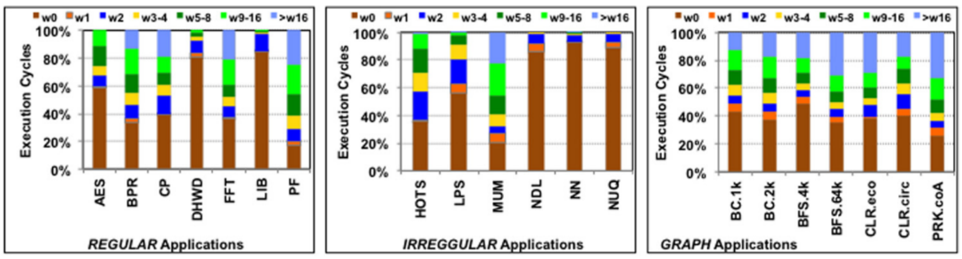


Fig. 12. Cycles breakdown based on number of *issuable warps* for *REGULAR*, *IRREGULAR* and *GRAPH* applications.

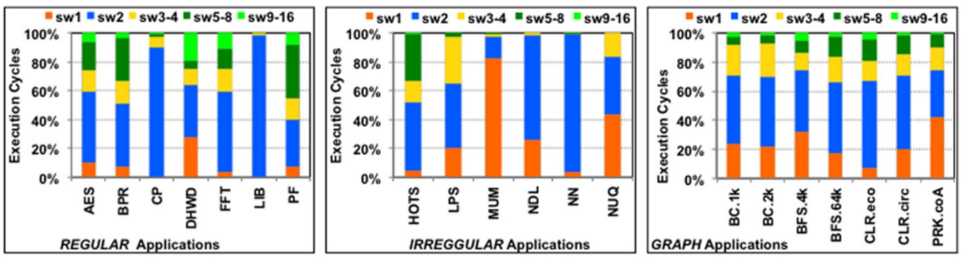


Fig. 13. Distribution of Scalar-waves based on their number of *scalars* for *REGULAR*, *IRREGULAR* and *GRAPH* applications.

(s0), 1 *(s1)*, 2-3 *(s2-3)*, 5-8 *(s5-8)*, or 9-16 *(s9-16)* for *REGULAR*, *IRREGULAR*, and *GRAPH* applications.

The number of instances where the scheduler finds two or more issuable —*multi-issuable*, warps is comparably higher in PF, FFT, CP, and BPR benchmarks from the *REGULAR* benchmark set. We also see higher amount of issuable scalars in PH, FFT, and BPR benchmarks from the *REGULAR* benchmark set.

Fig. 13 shows the distribution of Scalar-Waves, where the number of *scalars* is 1 *(sw1)*, 2-3 *(sw2-3)*, 5-8 *(sw5-8)*, or 9-16 *(sw9-16)* for regular, irregular, and graph benchmark sets.

In Fig. 13, we observe that packing rate of scalars in PF benchmark is significantly high. Higher availability of issuable warps and issuable scalars combined with higher scalar packing rate, the performance of PF improves the most from *REGULAR* benchmark set. For FFT and BPR benchmarks we also observe higher amount of scalar packing rate. However, FFT and BPR do not benefit form higher availability of issuable warps and issuable scalars as much as PF benchmark does. The amount of multi-issuable scalars is very small and the packing of scalar instructions to form a wave is not very effective for LIB. Most of the issued scalar-waves are formed with only one scalar. If we take a closer look at the LIB benchmark to better understand its behavior, we see that a significant number of executed scalars instructions due to a set of scalar global memory accesses occurring in a tight loop. The scalar instructions in this loop are data dependent on one another. This negatively impacts our ability to pack scalars in LIB.

In the *IRREGULAR* benchmarks, HOTS and LPS call for attention for their higher availability of issuable scalars and issuable warps. They also exhibit high scalar packing rate. These altogether affect the performances of these applications positively. Despite its high rates of issuable warps, MUM does not have high amount of issuable scalars and high scalar packing rate, and therefore it does not benefit much from Scalar-Waving.

For *GRAPH* applications, in general we observe higher rates of issuable warps, good scalar packing rate and good amount of issuable scalars for BFS.4k, CLR.eco, and CLR.cir benchmarks. We think that besides other factors, *GRAPH* benchmarks get good benefits of good occupancy rate and scalarization.

8.3.2. Instruction re-use in front-end

As part of this work, we also evaluated the effectiveness of our GW8 design for reusing the instructions and reducing the number of fetch and decode operations in the front-end.

In Fig. 14, we present the relative number of fetch and decode operations performed in the front-end. In general, GW8 architecture has high rate of instruction reuse in fetch-decode. Only exceptions are observed for benchmarks LIB, MUM, NUQ, and BFS.4k. Instruction reuse is not very effective for these benchmarks. We observe increased rate of instruction fetch and decode with the narrow SIMD-width. Despite the fact that it has good amount of scalar instruction to benefit from scalarization, the pressure in the front-end affects the performance of LIB due to higher rate of warps issued alone instead of issuing in pair.

8.3.3. Utilization of SIMD-units on graph-waving architecture

Fig. 15 shows a breakdown of execution cycles as a function of active SIMD-threads in warps to illustrate the utilization of the SIMD lanes. Please remember that, GW8 architecture has a SIMD width of 8 and 16 with aggregated issue. Therefore, we observe up to 16 lane utilization on GW8 architecture.

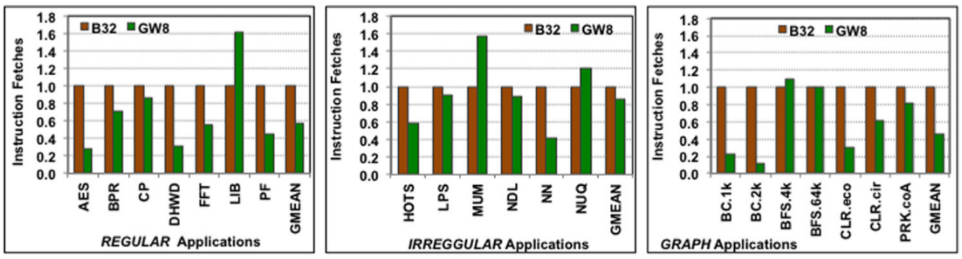


Fig. 14. Normalized number of instruction fetch&decode in front-end in GW8 and B32 architectures normalized to B32 architecture for *REGULAR*, *IRREGULAR* and *GRAPH* applications.

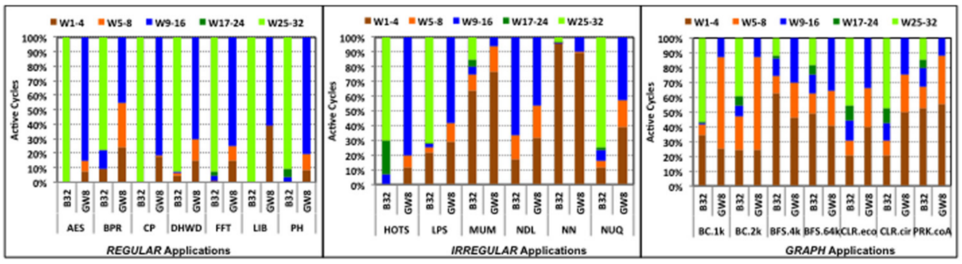


Fig. 15. Utilization of SIMD-units for *REGULAR*, *IRREGULAR* and *GRAPH* applications. *W1–W4* means 1 to 4 of SIMD-threads are active in a cycle, and a similar scheme applies for *W5–W8*, *W9–W16*, *W17–W24*, and *W25–W32*.

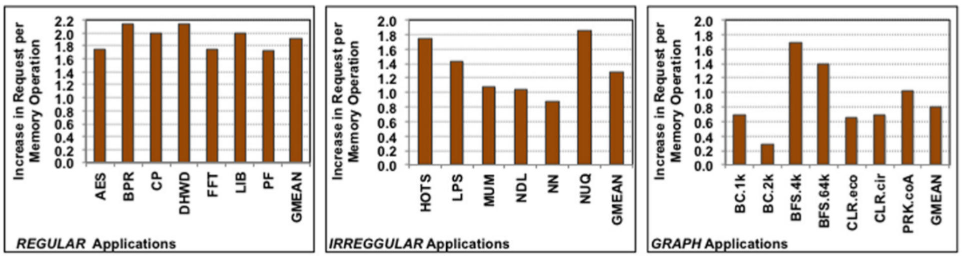


Fig. 16. Normalized number of memory transactions for GW8 architecture when running *REGULAR*, *IRREGULAR* and *GRAPH* applications.

GW8 architecture favors a narrow SIMD-width and employs a clustered issue to schedule an odd and even SIMD-group/warp at each scheduling cycle. However, when the scheduler cannot find a pair of odd and even warps to schedule together, then a single odd or even warp gets scheduled. When that happens, only the half of the SIMD-lanes gets utilized. In Fig. 15, we see decrease in SIMD utilization. Decreased utilization is mostly from scheduling of a single warp instead of a pair and from Scalar-wave execution with low packing rate. Thankfully, the benefits of scalar-waving overcome its limitations. However, issuing a single warp instead of a pair may harm the performance. Specifically, we notice that BPR and LIB have increased execution cycles with low SIMD utilization. As a future work, we would like to look into scheduling techniques for improved pair scheduling and increasing scalar packing rate.

8.3.4. *Memory accesses*

Narrowed issue width could lead to increased number of memory requests and that could create pressure in first level of caches. Therefore, in our detailed analysis, we looked at the average number of memory requests per memory operation, and L2 cache accesses and misses.

In Fig. 16, we show the number of memory requests in GW8 architecture normalized to B32 architecture when running regular, irregular, and graph applications. We observe that for most of the regular applications the number of memory requests increases. Even for a few benchmarks, we observe the amount of increase is more than twice. This is again because of single warp issue. Benchmarks LIB and DHWD show increased L2 cache accesses besides their increased number of memory request and their performance get effected from increased L2 accesses.

With irregular and graph benchmarks we observe mix results. For most of the regular and irregular benchmarks L2 cache access and miss rates do not change much as shown in Fig. 17. For most of the graph benchmarks (except BC.1k), we observe lowered L2 cache accesses and miss rate.

9. Related work

There are many studies on efficient implementations of graph algorithms on GPUs. However, none of them proposes architectural solutions to better support graph algorithms on GPUs. Harish and Narayanan [20] showed potential of using GPUs for acceleration of graph algorithms using large graphs. Many works [14,18,21,26,34,37,42,48,57] presented efficient implementation and performance evaluation of widely used graph algorithms for GPU and multi-GPU systems focusing on algorithmic changes for GPU architecture and use of scalable primitives. Che et al. [6] developed Pannotia Benchmark Suite for GPUs using OpenCL and characterized their performance on AMD GPUs. Xu et al. [53] also studied performance of graph applications and they performed detailed analysis using simulations. Similar to our findings, both studies showed that performance of graph algorithms are limited with their irregular data accesses, low utilization and data-dependent work distribution.

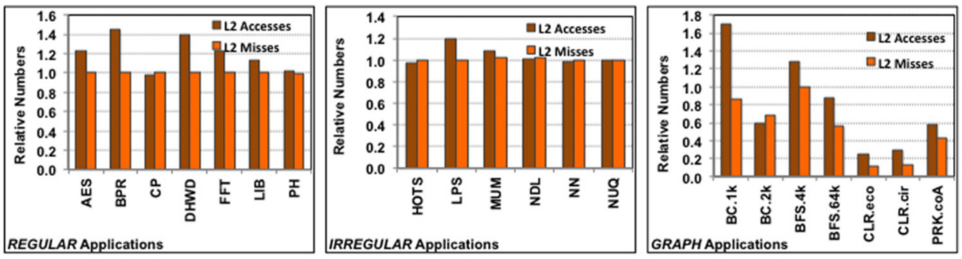


Fig. 17. Normalized L2 Cache accesses and misses for GW8 architecture when running *REGULAR*, *IRREGULAR* and *GRAPH* applications.

Hong et al. [23] proposed virtual warp-centric programming model for acceleration of graph algorithms on GPUs. Similar to our scalarized vertex-centric graph computing, virtual warp-centric graph computing maps a vertex to a warp. Khorasani et al. [27] proposed *Warp Segmentation* to further enhance the GPU device utilization by dynamically assigning the suitable number of SIMD-threads to a vertex. *Virtual warp mapping* and *Warp Segmentation* help with low utilization of SIMD units. However, these two studies are programming approaches at software level and do not eliminate redundant computation. They only work for applications where you can apply *Virtual Warp-Centric* computing or *Warp Segmentation*. On the other hand, our work is at the architectural level, combines techniques from software and hardware level. GW architecture targets a larger set of GPU applications.

There are two main approaches known as vertex-centric [28,33,36] and edge-centric [17,45] in graph processing. There are also frameworks [49,58] that combine both vertex-centric and edge-centric graph processing. Besides vertex-centric graph computing, edge-centric graph computing applications can benefit from our Graph-waving architecture. Such as, they can still benefit from the instruction re-use in the front-end, narrow SIMD-width, and scalar-waving. Even though there is no scalar annotation to target vertex specific calculations, the compiler can identify scalars for other calculations if there exist any. There are also other work that target optimizations using different data representations to improve memory accesses or SIMD-utilization on GPUs [3,24,28,51,54]. [19,31] and [55] present models for predicting the performance of SpMV on GPUs with different data storage types. Our analysis on graph algorithms using CSR agrees with the results presented from these studies.

In the context of scalar execution, researchers have proposed dynamic [10] and static analysis techniques [9,30] to identify scalar operations in GPU applications. Xiang et al. [50] proposed a new design for redundancy elimination in computation, energy efficiency and reliability enhancement. Chen and Kaeli explored a scalar–vector architecture in [8]. GScalar [32] work extends scalar execution to include divergent scalar instructions and also utilizes register compression for reduced power consumption. However, none of these prior works attempted to group scalar instructions for efficiency and none of them employs instruction re-use in front-end for elimination of redundant fetches and decodes. In our previous work [56], we proposed Scalar-waving architecture to eliminate redundant computation and vectorized storage of scalar results for efficiency. In this work, we extend and adopt the Scalar-waving, to use scalarized graph computing approach, integrated with narrow warps and instruction re-use in front-end.

There have been several proposals to address the low utilization of the SIMD lanes. In Dynamic Warp Formation (DWF) [16], Thread Block Compaction (TBC) [15], CAPRI [44], and Large Warp Micro-architecture (LWM) [38], researchers developed methods to dynamically form warps from warp splits under divergence. The general observation with thread regrouping is that the thread regrouping process impacts the number of coalesced memory accesses and creates shared memory bank conflicts. While each of the previous studies has positively impacted performance and addressed issues with SIMD efficiency and divergence, there has been no single approach that addresses these problems completely. None of the previous studies targeted the elimination of redundant computation, instruction reuse in the front in GPU computing.

10. Conclusions and future work

GPUs provide impressive speedups for *GPU-friendly* data parallel applications. On the other hand, graph and many other irregular applications suffer from low utilization of SIMD-units due to varied amount of computations across the vertexes. Low utilization of SIMD-units leads to significant performance degradation.

Our work introduces GW architecture to improve architectural support for irregular, data-dependent applications like graph applications. GW uses a narrow SIMD-width with a clustered issue approach, introduces extensions to the baseline GPU architecture to reuse instructions in the front-end and eliminates redundant computations using Scalar-waving. Combination of these three techniques helps with low utilization of SIMD-lanes, redundant computations, and inefficient use of SIMD front-end. With GW architecture, we can improve performance for many graph applications as much as 4.4x on the average, 17% and 9% for different regular and irregular GPU applications, respectively.

As a future work, scheduling optimizations can be studied to further improve the benefits of Graph-waving architecture. Extending GW architecture with dynamic identification of scalar instruction is another future work that we consider.

CRediT authorship contribution statement

Ayse Yilmazer-Metin: Conceptualization, Methodology, Software, Visualization, Investigation, Validation, Resources, Writing - review & editing, Writing - original draft, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: David Kaeli

References

[1] Advanced Micro Devices, Inc., GCN architecture, Sunnyvale, CA, URL https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, 2009, pp. 163–174.

[3] T. Ben-Nun, M. Sutton, S. Pai, K. Pingali, Groute: An asynchronous multi-GPU programming model for irregular computations, in: PPoPP '17, 2017.

[4] Billconan, Kavinguy, A neural network on GPU, URL http://www.codeproject.com/KB/graphics/GPUNN.aspx.
[5] U. Brandes, A faster algorithm for betweenness centrality, J. Math. Sociol. 25 (2001) 163–177.
[6] S. Che, B.M. Beckmann, S.K. Reinhardt, K. Skadron, Pannotia: Understanding irregular gpgpu graph applications, in: 2013 IEEE International Symposium on Workload Characterization, IISWC, 2013, pp. 185–195.
[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, 2009, pp. 44–54.
[8] Z. Chen, D. Kaeli, Balancing scalar and vector execution on GPU architectures, in: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2016, pp. 973–982.
[9] Z. Chen, D. Kaeli, N. Rubin, Characterizing scalar opportunities in GPGPU applications, in: 2013 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2013, pp. 225–234.
[10] S. Collange, D. Defour, Y. Zhang, Dynamic detection of uniform and affine vectors in GPGPU computations, in: Proceedings of the 2009 International Conference on Parallel Processing, Euro-Par'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 46–55.
[11] T.H. Cormen, C. Stein, R.L. Rivest, C.E. Leiserson, Introduction to Algorithms, second ed., McGraw-Hill Higher Education, 2001.
[12] B. Coutinho, D. Sampaio, F.M.Q. Pereira, W. Meira Jr., Divergence analysis and optimizations, in: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 320–329.
[13] DIMACS, Citation networks, URL https://www.cc.gatech.edu/dimacs10/archive/coauthor.shtml.
[14] B.O. Fagginger Auer, R.H. Bisseling, in: R. Keller, D. Kramer, J.-P. Weiss (Eds.), Facing the Multicore-Challenge II, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 108–119, Ch. A GPU algorithm for greedy graph matching.
[15] W.W.L. Fung, T.M. Aamodt, Thread block compaction for efficient SIMT control flow, in: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, HPCA '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 25–36.
[16] W.W.L. Fung, I. Sham, G. Yuan, T.M. Aamodt, Dynamic warp formation and scheduling for efficient GPU control flow, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, IEEE Computer Society, Washington, DC, USA, 2007, pp. 407–420.
[17] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 17–30.
[18] A.V.P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, M. Hall, Evaluating graph coloring on GPUs, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 297–298.
[19] P. Guo, L. Wang, P. Chen, A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs, IEEE Trans. Parallel Distrib. Syst. 25 (5) (2014) 1112–1123.
[20] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: Proceedings of the 14th International Conference on High Performance Computing, HiPC'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 197–208.
[21] K.A. Hawick, A. Leist, D.P. Playne, Parallel graph component labelling with GPUs and CUDA, Parallel Comput. 36 (12) (2010) 655–678.
[22] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA graph algorithms at maximum warp, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 267–276.
[23] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA graph algorithms at maximum warp, SIGPLAN Not. 46 (8) (2011) 267–276.
[24] C. Hong, A. Sukumaran-Rajam, J. Kim, P. Sadayappan, MultiGraph: Efficient graph processing on GPUs, in: 2017 26th International Conference on Parallel Architectures and Compilation Techniques, PACT, 2017, pp. 27–40.
[25] IMPACT Research Group, Parboil benchmark suite, URL http://impact.crhc.illinois.edu/parboil/parboil.aspx.
[26] O. Kalentev, A. Rai, S. Kemnitz, R.F. Schneider, Connected component labeling on a 2D grid using CUDA, J. Parallel Distrib. Comput. 71 (2011) 615–620.
[27] F. Khorasani, R. Gupta, L.N. Bhuyan, Scalable SIMD-efficient graph processing on GPUs, in: 2015 International Conference on Parallel Architecture and Compilation, PACT, 2015, pp. 39–50.
[28] F. Khorasani, K. Vora, R. Gupta, L. Bhuyan, CuSha: vertex-centric graph processing on GPUs, in: HPDC 2014 - Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, 2014.
[29] Khronos Group, Open Computing Language (OpenCL) 1.2 specification, URL http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.
[30] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, K. Asanovic, Convergence and scalarization for data-parallel architectures, in: Code Generation and Optimization, CGO, 2013 IEEE/ACM International Symposium on, 2013, pp. 1–11.
[31] K. Li, W. Yang, K. Li, Performance analysis and optimization for SpMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 196–205.
[32] Z. Liu, S. Gilani, M. Annavaram, N.S. Kim, G-Scalar: Cost-effective generalized scalar execution architecture for power-efficient GPUs, 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2017, pp. 601–612.
[33] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Graphlab: A new framework for parallel machine learning, in: P. Grünwald, P. Spirtes (Eds.), UAI, AUAI Press, 2010, pp. 340–349.
[34] L. Luo, M. Wong, W.-m. Hwu, An effective GPU implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference, DAC '10, ACM, New York, NY, USA, 2010, pp. 52–55.
[35] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146.
[36] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146.
[37] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, SIGPLAN Not. 47 (8) (2012) 117–128.
[38] V. Narasiman, M. Shebanow, C.J. Lee, R. Miftakhutdinov, O. Mutlu, Y.N. Patt, Improving GPU performance via large warps and two-level warp scheduling, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44 '11, ACM, New York, NY, USA, 2011, pp. 308–317.
[39] NVIDIA Corporation, NVIDIA's next generation CUDA compute architecture: Fermi, URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
[40] NVIDIA Corporation, Parallel programming and computing platform, CUDA, URL https://developer.nvidia.com/cuda-zone.
[41] NVIDIA Corporation, NVIDIA CUDA SDK code samples, URL http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.
[42] V.M. Oliveira, R. de Alencar Lotufo, A study on connected components labeling algorithms using GPUs, 2010.
[43] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank Citation Ranking: bringing Order to the Web, Technical Report 1999-66, Stanford InfoLab, 1999, Previous number = SIDL-WP-1999-0120.
[44] M. Rhu, M. Erez, CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures, in: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 61–71.
[45] A. Roy, I. Mihailovic, W. Zwaenepoel, X-Stream: Edge-centric graph processing using streaming partitions, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, ACM, New York, NY, USA, 2013, pp. 472–488.
[46] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
[47] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G.D. Liu, W. mei W. Hwu, Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing, University of Illinois at Urbana-Champaign, 2012.
[48] V. Vineet, P. Harish, S. Patidar, P.J. Narayanan, Fast minimum spanning tree for large graphs on the GPU, in: Proceedings of the Conference on High Performance Graphics 2009, HPG '09, ACM, New York, NY, USA, 2009, pp. 167–171.
[49] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J. Owens, Gunrock: a high-performance graph processing library on the GPU, ACM SIGPLAN Not. 51 (2016) 1–12.
[50] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L.R. Hsu, H. Zhou, Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 433–442.
[51] G. Xiao, K. Li, Y. Chen, W. He, A. Zomaya, T. Li, CASpMV: A customized and accelerative SpMV framework for the sunway taihulight, IEEE Trans. Parallel Distrib. Syst. (2019) 1–1.
[52] Q. Xu, H. Jeon, M. Annavaram, Graph processing on gpus: Where are the bottlenecks? in: 2014 IEEE International Symposium on Workload Characterization, IISWC, 2014, pp. 140–149.
[53] Q. Xu, H. Jeon, M. Annavaram, Graph processing on gpus: Where are the bottlenecks? in: 2014 IEEE International Symposium on Workload Characterization, IISWC, 2014, pp. 140–149.

[54] W. Yang, K.-L. Li, K. Li, A pipeline computing method of SpTV for three-order tensors on CPU and GPU, ACM Trans. Knowl. Discov. Data 13 (2019) 63:1–63:27.
[55] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned SpMV on GPUs and multicore CPUs, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.
[56] A. Yilmazer, Z. Chen, D. Kaeli, Scalar waving: Improving the efficiency of SIMD execution on GPUs, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 103–112.
[57] K. Yonehara, K. Aizawa, A line-based connected component labeling algorithm using GPUs, in: 2015 Third International Symposium on Computing and Networking, CANDAR, 2015, pp. 341–345.
[58] J. Zhou, C. Xu, X. Chen, C. Wang, X. Zhou, Mermaid: Integrating vertex-centric with edge-centric for real-world graph processing, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, 2017, pp. 780–783.

Dr. Ayse Yilmazer received her Ph.D. degree in January 2014 from Electrical and Computer Engineering Department of Northeastern University. She holds an M.S. degree in Computer Engineering from Electrical and Computer Engineering Department of University of Rhode Island and a B.S. degree in Computer Science and Engineering from Hacettepe University.

Her main research interests include GPUs and GPGPU computing, heterogeneous systems and heterogeneous computing, parallel computing and parallel architectures, synchronization, memory consistency, memory systems performance analysis, compiler analysis techniques and runtime optimizations. Dr. Yilmazer's research was recognized by IPDPS Best Paper Award in 2006. Currently, Dr. Yilmazer works at Istanbul Technical University as an Assistant Professor. Before joining to Istanbul Technical University, she worked at AMD Research for one year.