# EFFICIENT GRAPH ALGORITHM EXECUTION ON DATA-PARALLEL ARCHITECTURES

A Thesis
Presented to
The Academic Faculty

by

Nagesh Bangalore Lakshminarayana

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2014

# EFFICIENT GRAPH ALGORITHM EXECUTION ON DATA-PARALLEL ARCHITECTURES

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
*Georgia Institute of Technology*

Dr. Santosh Pande
School of Computer Science
*Georgia Institute of Technology*

Dr. Milos Prvulovic
School of Computer Science
*Georgia Institute of Technology*

Dr. Moinuddin K. Qureshi
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Dr. Richard W. Vuduc
School of Computational Science and Engineering
*Georgia Institute of Technology*

Date Approved: October 30, 2014

*Dedicated to my parents, siblings and wife*

# ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the people who have guided, helped and supported me during my time at Georgia Tech.

Firstly, I would like to thank my advisor Dr. Hyesoon Kim for her guidance, mentoring, support and encouragement over the years. From critiquing research articles to writing my own papers and dissertation, from identifying research problems to solving them, from evaluating ideas to presenting them, from applying for jobs to accepting an offer, she has provided valuable advice and guidance with the utmost patience. She has been a guiding light during the course of my graudate studies and continues to inspire me.

I would also like to thank Dr. Santosh Pande, Dr. Milos Prvulovic, Dr. Moin Qureshi, Dr. Richard Vuduc and Dr. Sudhakar Yalamanchili for serving on my dissertation proposal and defense committees and for their feedback and comments on improving the dissertation.

Many thanks to past and present HPArch members - Minjang Kim, Sunpyo Hong, Sungbae Kim, Jaekyu Lee, Pranith Kumar, Jaewoong Sim, Dilan Manatunga, Joo Hwan Lee, Jieun Lim, Hyojong Kim, Nimit Nigania, Jen-Cheng Huang, Prasun Gera - for their feedback on my research and for the discussions during the weekly meetings and paper readings. Special mention goes to Jaekyu Lee with whom I collaborated for some of my initial research and on the development of the simulator used for most of his and my research. I would like to thank Jaekyu Lee and Nimit Nigania for their help during my job search as well. Thanks to Sungbae Kim, Chayoung Lee, Jieun Lim, Joo Hwan Lee and Hyojong Kim for the many non-work related conversations that I thoroughly enjoyed and which often served as a much needed break during the long days in the lab. I would also like to thank Sushma Rao and Jinwoo Shin with whom I collaborated on some of my earlier work.

I would like to thank all my roommates and other friends I made during my stay in Atlanta; I have enjoyed good times with all of them. Special mention to my roommates at Metro Apartments and the rest of the gang from Metro and Icon Apartments for the fun-filled get togethers and parties. Thanks to Kaushik Ravichandran for our conversations, discussions and his help on numerous occasions. Thanks to Balaji Palaniswamy for our many trips to Alpharetta for Indian food. Thanks to my friends back in Wilson Garden, Bangalore and to my classmates from RVCE for the advice, conversations and help on several occasions.

I would also like to acknowledge the staff at the College of Computing and city of Atlanta itself; I have had a very postive experience in interacting with everyone and studying and living here.

Last but not the least, I would like to express my gratitude to my parents - Bharathi and Lakshminarayana, siblings - Geetha and Umesh, and wife - Pratibha, for their love, suppport, sacrifices and patience. They have spent many sleepless nights worrying about me and have rejoiced the completion of my PhD more than me. This thesis is dedicated to them.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

## *1.1 The Problem: Efficient execution of graph algorithms on data-parallel architectures*

Graph algorithms are an important component of many traditional, recent and emerging applications such as network simulation, social computing, bio-informatics, communication networks, network security, navigation systems, search engines, scheduling in transportation networks. With the ever increasing amount of data that is processed by applications (many of which rely on parallel graph algorithms), the need for faster and power efficient execution of graph algorithms is more today than it was before. Data-parallel architectures with their ability to process many data elements in parallel could be suitable for parallel processing of graphs. In this thesis we consider the modern GPU with its compute capabilities as the most prevalent data-parallel architecture and focus on it. GPUs are power and energy efficient and provide high throughput and bandwidth capabilities. Recently, there has been considerable work enabling and improving the performance of fundamental graph algorithms such as Breadth First Search, Minimum Spanning Tree, Single Source Shortest Path and others on GPUs [31, 76, 32, 70, 35, 34, 51, 4]. However, the irregular and data intensive nature of graph algorithms presents several obstacles to the GPU.

The GPU architecture is ideal for applications that are data parallel with regular memory accesses, regular control flow and uniform work distribution across threads. And such applications execute on the GPU with high execution and energy efficiency. By their very nature, graph algorithms have low execution efficiency on GPUs. They suffer from control flow divergence, non-uniform work distribution and a large number of data dependent memory accesses with divergence. Because of the large number of memory accesses that tend to be irregular and data-dependent, graph algorithms are often unable to hide memory access latency. Significant work has already been done on reducing the negative performance impact of control flow divergence [26, 50, 25, 22, 60, 8] in GPGPU applications and the same techniques can be used to reduce inefficiency caused due to branch divergence in graph applications. Though there have been some efforts to improve work distribution across threads and to improve the memory access latency tolerance of irregular applications including graph algorithms, sufficient scope still exists for innovation.

## *1.2 The Solution: Graph algorithm specific mechanisms*

The solution to the problem of inefficient execution of graph algorithms on GPGPUs is designing mechanisms that are aware of their characteristics. To this end, we propose a graph algorithm characteristic aware prefetching mechanism and explore the design of cache hierarchies for graph algorithms. We also examine the impact of implementation decisions on graph algorithm performance, power and energy consumption and the impact of software optimizations such as prefetching and loop unrolling on graph algorithm performance.

### 1.2.1 Graph algorithm aware prefetching

One approach to improving the tolerance to memory access latency is to prefetch data that will be accessed in the future. We propose a mechanism that detects and prefetches an access pattern that is common to several graph algorithm implementations. Another novelty of the proposed mechanism is that in addition to the prefetched data being stored in the data cache, it is also stored in spare registers that are available when a GPU kernel is executing. In GPUs, threads are allocated to processing units at thread block granularity, because of which even if sufficient registers to assign an entire new thread block are unavailable, a few additional registers can be allocated to each thread already assigned to the processing units. The proposed mechanism has the benefit that prefetched data does not get evicted from cache which would require memory requests to be issued again.

### 1.2.2 Exploring the design of cache hierarchies for graph algorithms

Though GPU caches are very small, the design of the cache hierarchy can also play an important role in improving tolerance of graph algorithms to memory access latency. Unlike the traditional streaming GPGPU applications, the graph kernels we examine have some locality in the accesses of individual threads and across threads as well. However, because of the small cache space per thread that is available on GPUs, cache blocks with locality get evicted from the cache before their last use and have to be fetched again from the lower levels of memory. Thus approaches which improve the effectiveness of caches would be helpful. We first evaluate how the cache inclusion property affects the performance of graph algorithms. Besides a non-inclusive cache hierarchy, we consider an exclusive cache hierarchy and other cache designs that selectively do exclusion including a new memory region based cache exclusion scheme. The new scheme identifies regions that are accessed by cores in a private fashion and does exclusion for only such regions. We then examine the impact of different cache bypass mechanisms on the performance of graph algorithms. Finally, we examine the impact of fine-grained cache hierarchies that dynamically decide how many cache sub-blocks to fetch. Though a fine-grained cache hierarchy does not improve the efficiency of caching, it makes the memory hierarchy more efficient by reducing the interconnect and memory bandwidth consumed and this could potentially improve performance.

### 1.2.3 Impact of implementation strategies

The strategy adopted by the programmer in implementing a graph algorithm can also impact the performance, power and energy consumption of the application. To understand the impact of implementation strategies we evaluate multiple implementations of the same graph algorithm for different inputs. We also consider the impact of software optimizations such as software prefetching and loop unrolling.

## *1.3 Thesis Statement*

The execution efficiency (in both time and power consumed) of graph algorithms on data-parallel architectures can be improved by designing mechanisms for the memory hierarchy that are aware of the distinguishing characteristics of graph algorithms.

## *1.4 Organization of this document*

Chapter 2 provides some background about implementation of graph algorithms on GPUs. Chapter 3 presents the prefetching mechanism, Chapter 4 presents the exploration of the cache hierarchy design and Chapter 5 presents the evaluation and discussion on different implementation strategies. Chapter 6 discusses related work and Chapter 7 concludes this document.

# CHAPTER II

# BACKGROUND

## *2.1 GPGPU terminology and architecture*

In this section we briefly introduce the GPU terminology used in this document and the GPU architecture we model. Our teminology and the modeled architecture are both based on NVIDIA's CUDA [56].

### 2.1.1 GPGPU Terminology

*Thread Warp:* A warp is a group of 32 threads that execute in lock-step. It is the granularity of execution in GPGPU computing i.e., instructions are executed for one warp at a time. [1]

*Thread Block:* A thread block is a group of threads that should be executed concurrently. Threads in a block can synchronize with each other via barriers and share data through scratch pad memory called Shared Memory. A thread block consists of one or more warps.

*Kernel:* A kernel is a function that runs on the GPU. It is executed by a grid of threads which is composed of one or more thread blocks.

### 2.1.2 GPGPU Architecture

A GPU (Figure 1) consists of several cores each with SIMD execution units, a software managed cache called shared memory, L1 instruction and data caches, memory-request queue (MRQ), and other units. Each core has an in-order instruction scheduler which executes instructions from warps in a round-robin fashion. No coherence

---

[1] The number of threads in a warp is implementation dependent; NVIDIA's CUDA architecture [56] uses 32 threads per warp.

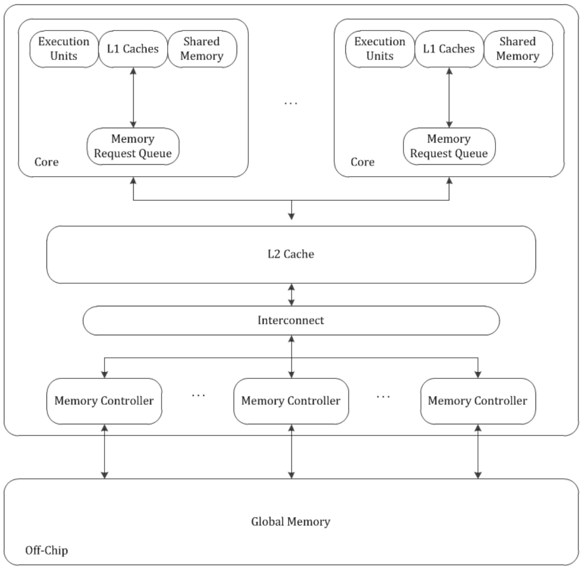
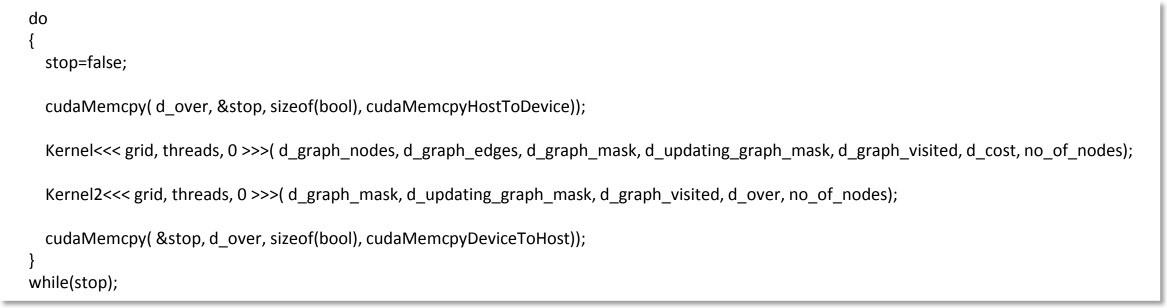


Figure 1: GPGPU Architecture

is maintained for the data in the private L1 caches. Misses in the L1 caches are forwarded to a shared, non-inclusive L2 cache, which in turn forwards each L2 miss to one of the multiple on-chip DRAM controllers. The L2 is highly banked to handle requests from multiple cores. Like the L2, the DRAM is also highly banked and each bank has a large request buffer to handle many requests; there are also multiple channels to improve DRAM throughput. The exact configuration of the architecture that is modeled is shown in Chapter 3.

## *2.2 Graph Algorithms on GPUs*



```
do
{
   stop=false;

   cudaMemcpy( d_over, &stop, sizeof(bool), cudaMemcpyHostToDevice));

   Kernel<<< grid, threads, 0 >>>( d_graph_nodes, d_graph_edges, d_graph_mask, d_updating_graph_mask, d_graph_visited, d_cost, no_of_nodes);

   Kernel2<<< grid, threads, 0 >>>( d_graph_mask, d_updating_graph_mask, d_graph_visited, d_over, no_of_nodes);

   cudaMemcpy( &stop, d_over, sizeof(bool), cudaMemcpyDeviceToHost));
}
while(stop);
```

Figure 2: Host-side code for BFS

Graph algorithm implementations on GPUs typically involve multiple iterations of a sequence of steps, with data dependency between each successive iteration and between the steps of an individual iteration. The data dependency between the successive steps of an iteration means that synchronization is required between successive steps and hence on GPUs each step is often implemented as an individual kernel (or multiple kernels). Figure 2 shows the host-side code for Breadth First Search (BFS) provided by Pawan et al. [31]. Pawan et al. implement BFS using two kernels - one kernel expands the BFS tree using a active vertex frontier (kernel) and another kernel (kernel2) builds the vertex frontier that will be used in the next iteration for expansion. The expansion continues until the vertex frontier for the next iteration is empty.

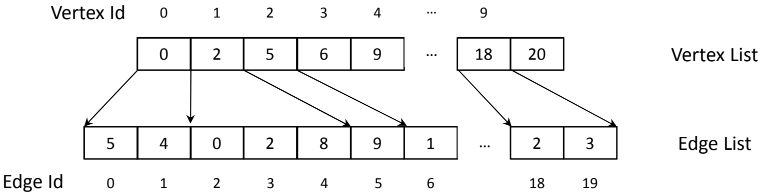


Figure 3: Example Graph in Compact Adjacency List representation

### 2.2.1 Graph Representation on GPUs

A common way for representing graphs in CUDA implementations of graph algorithms is the compact adjacency list representation. In this representation, a graph G(V, E) is essentially represented using two lists (arrays) - the Vertex list and the Edge list. The vertex list has one element for each vertex in the graph with the element for vertex $v$ being stored at index $v$ in the list. The edge list contains the destination vertex ids for all the edges in the graph. For a given vertex $v$, the endpoints of all edges from $v$ are stored contiguously in the edge list. The element for $v$ in the vertex list contains the index of the location in the edge list where the endpoint of the first edge from $v$ is stored. The number of edges originating from $v$ can be computed as

($vertex\_list[v] - vertex\_list[v+1]$). Compared to an adjacency matrix representation, an adjacency list representation consumes less space for sparse graphs and also makes neighbor traversal easier. Figure 3 shows the compact adjacency list representation for an example graph with 10 vertices and 20 edges. In the graph, vertex 0 has two outgoing edges to vertices 5 and 4, vertex 1 has 3 outgoing edges and so on.

## *2.3 Evaluated graph algorithms*

Recently, several graph algorithms have been implemented on the GPU. In this section we briefly discuss the algorithm implementations used in this thesis.

### 2.3.1 Breadth First Search (BFS)

Breadth First Search (BFS) algorithm initiates a search from a given source vertex. For each vertex that is reachable from the source vertex, BFS calculates the number of hops (edges) in the path from the source vertex to the given vertex. The BFS search progesses in levels; initially only the source vertex, which is at level-0, is active. In the processing of each level, yet to be reached neighbors of vertices active in the current level are marked to be active in the next level. In CUDA, the processing at each level is implemented as one or more kernels with the active vertices at each level being maintained using a vertex frontier. At each level, the vertex frontier for the next level is formed and the search terminates when the vertex frontier is empty. In this thesis we use BFS implementations from Harish et al [31], LonestarGPU [4], Luo et al [49] and SHOC [21, 34]. Some of these implementations use multiple kernels - one kernel for expanding the vertex frontier and another for building the frontier. For such implementations the simulator based evaluations use only the kernels that process the vertices in the fronter, while the evaluations on a real GPU card consider all application kernels. The same is true for rest of the algorithms described in this section.

### 2.3.2 Single Source Shortest Path

Single Source Shortest Path (SSSP) is similar to BFS, however, it caculates the shortest path for each vertex from a given source vertex instead of the number of hops from the source vertex. Each edge in the graph has an associated cost and SSSP calculates the minimum cost of all possible paths from the source to the vertex. Initially, the cost of each vertex from the source vertex is marked as infinite; the cost of the source vertex itself is marked as zero. In every iteration, vertices whose costs were updated in the previous iterations update the cost of their neighbors if the new cost is less than their previous cost. This continues until there is an iteration in which no costs are updated, meaning the costs of all vertices are at the minimum possible values. We use SSSP implementations from Harish et al [31] and LonestarGPU [4].

### 2.3.3 S-T Connectivity

S-T Connectivity checks whether two vertices (S and T) are connected, and if they are, provides one of the possible paths between S to T. The algorithm initiates BFS searches from both S and T vertices and terminates when the two searches have met or have completed. We implement S-T Connectivity based on the description provided by Harish et al [32].

### 2.3.4 Minimum Spanning Tree

The Minimum Spanning Tree (MST) algorithm computes the spanning tree (a subgraph that is a tree i.e. no cyles, and connects all vertices in the graph) whose cost is lower than or equal to the cost of all other spanning trees in the graph. The CUDA implementations [32, 4] we consider implement Boruvka's parallel Minimum Spanning Tree algorithm. The algorithm iteratively grows super vertices until there is only one super vertex. Initially each super vertex consists of one vertex in the graph. In each iteration, a given super vertex is merged with another super vertex which is connected to it by the minimum cost edge incident on the given super vertex. The minimum cost edge that connects the two super vertices is added to the growing MST. Any cycles that may been formed between the super vertices are removed before the next iteration. In our evaluations we use the kernel the finds the minimum cost edge that is incident on a super vertex.

### 2.3.5 Graph Coloring

Graph Coloring (CLR) assigns colors to the vertices of the graph such that no two adjacent vertices have the same color. Before CLR kernels are launched on the GPU, each vertex is assigned a random value on the host CPU. Then the CLR kernels are launched iteratively until all vertices are colored. In each iteration, the vertices that have not been colored so far examine the random numbers assigned to their uncolored neighbors and mark themselves with the current color if their random number is the highest among the uncolored vertices. We implement CLR based on prior work [17, 13].

### 2.3.6 Maximal Independent Set

Maximal Independent Set (MIS) attempts to identify the largest set of vertices in a graph such that no two are adjacent. Just like in CLR, before MIS kernels are launched on the GPU, each vertex is assigned a random value on the host CPU. Then the MIS kernels are launched iteratively until the MIS is found. Initially, all vertices are active and in each iteration, an active vertex checks whether it can be added to the current MIS. If so, the vertex is added to the MIS and all its neighbors are marked inactive. The algorithm terminates when there are no active vertices remaining. We implement MIS based on Luby's algorithm [46, 13].

### 2.3.7 Other algorithms

Besides these algorithms, few other graph algorithms such as the Traveling Salesman Problem (TSP), MaxFlow-MinCut, All Pairs Shortest Path (APSP) have CUDA implementations. These implementations either do not include loops (MaxFlow-MinCut), use a different graph representation than what is considered in this thesis (APSP, TSP using matrices) or can be implemented using one of the algrithm implementations considered above (APSP using SSSP).

# CHAPTER III

# SPARE REGISTER AWARE PREFETCHING

## *3.1 Introduction*

This chapter introduces a prefetching mechanism that exploits an access pattern found in several graph kernels. The mechanism makes use of spare registers that are often available during kernel execution in GPUs.

Unlike CPU cores which aggressively use all the available architecture/physical registers, GPU cores could have several unused physical registers. The physical register file in GPUs is designed to support a massive number of threads, but often applications do not utilize all available registers. Figure 4 shows the size of unused registers out of a 128KB register file for several different graph algorithm kernels (see Table 4 for kernels). [1] As we can see, a significant fraction of the register file is unused. Furthermore, in the case of irregular applications, the number of unused registers can vary at run-time because of the irregularity of applications. Hence, in this thesis, we propose to utilize the unused registers in GPUs in co-ordination with prefetching to attack problems due to memory accesses in graph algorithm.

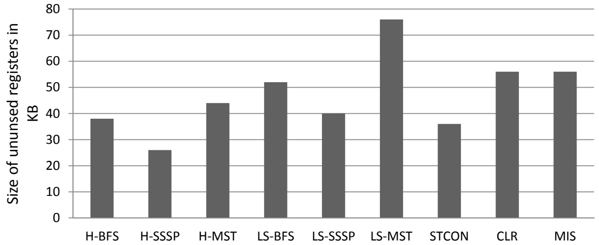


Figure 4: Unused registers out of 128KB register file

Prior prefetching mechanisms use caches to store prefetched data. Instead, the

---

[1] NVIDIA's Fermi architecture uses a 128KB register file per GPU core

proposed mechanism stores prefetched data in unallocated spare registers that are available in a GPU core. By doing so, our mechanism escapes from the problem of early eviction of prefetched data placed in the cache. Early eviction is a much more serious problem in GPU architectures than in CPUs due to the large number of threads and the relatively small cache size available per thread. Binding software prefetching also utilizes register files to store prefetched data but binding prefetch instructions could cause high overhead to check legitimacy of prefetched memory addresses. Furthermore, our mechanism is specialized to prefetch dependent loads that are the key memory access patterns in graph algorithms. In summary, this section makes the following contributions:

1. We show that irregular GPGPU applications tend to have large spare (unused) register files at run-time. Hence, we propose a prefetching mechanism to store prefetched data in spare registers to reduce prefetch overhead in caches such as early eviction.

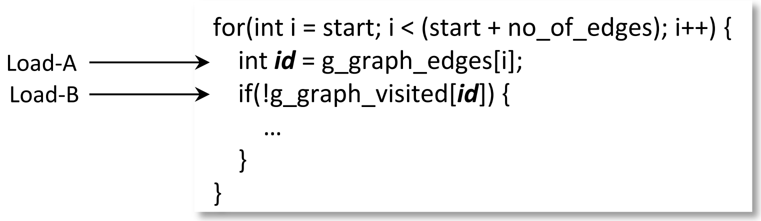
2. We propose a hardware based mechanism for detecting and prefetching load-pairs with one load dependent on the other to benefit graph algorithms on GPUs.

## *3.2 Background*

Section 2.2 provides a brief introduction to the representation and implementation of graph algorithms on GPUs. We first discuss why spare registers are often available on a GPU and then we explain the need for a new prefetching mechanism using a code sample taken from a graph kernel.

Unlike CPUs which utilize all available physical registers, a GPU could have several unused physical registers. There are two sources of spare registers in a GPU:

1. Due to resource constraints there may be unallocated registers in a GPU.



```
for(int i = start; i < (start + no_of_edges); i++) {
  int id = g_graph_edges[i];
  if(!g_graph_visited[id]) {
    ...
  }
}
```

Figure 5: Example BFS code

Threads are assigned to SMs at thread block granularity i.e., only if the resource requirement for all threads in a block can be satisfied will a block be assigned to a SM. It often happens that not enough resources are available to allocate an entire additional block, but a few additional registers can be assigned to each of the already assigned threads without reducing the occupancy of the kernel (see Table 4).

2. If a thread warp has terminated, then registers allocated to the warp can be used by other warps until the thread block containing the terminated warp has terminated and a new block is assigned to the SM.

### 3.2.1 Prefetching for graph algorithms

#### *3.2.1.1 Sample Code*

We refer to the code in Figure 5 which is taken from an implementation of Breadth First Search (BFS) [31] and shows the code for processing a vertex assigned to a thread. The neighbor list of the vertex starts at index *start* in array *g_graph_edges* and ends at index *(start + no_of_edges - 1)*. The thread accesses each neighbor of the vertex and some information about the neighbor; such processing can be considered to be typical of several graph algorithms on GPUs. Load-A loads from memory the vertex id of each neighbor and Load-B loads the flag which indicates whether a neighbor has already been visited by the search or not. Here, the index into the array *g_graph_visited* is loaded from array *g_graph_edges*, thus Load-B is dependent on

Load-A. The accesses to *g_graph_edges* are sequential/small strided across different loop iterations and thus have spatial locality. Depending on the input graph, accesses to *g_graph_visited* are unlikely to have any locality and a significant fraction of the execution time is spent on waiting for loads from *g_graph_visited* to complete.[2] Our proposal is in each loop iteration prefetch values from *g_graph_edges* and *g_graph_visited* that would be accessed in a subsequent iteration.

#### *3.2.1.2 Hardware Prefetching*

In CPUs, hardware prefetchers are trained with the last level cache (LLC) miss address stream. With sufficient training, the prefetchers can detect patterns or correlations in the miss address stream and can predict future miss addresses. The prefetcher designs used in CPUs cannot be directly applied to GPUs [44] - the prefetchers have to be thread (or warp) aware. In addition, for benchmarks with irregular and data dependent memory accesses, training using the LLC miss address stream will not help in detecting correlation in the memory accesses.

For the access pattern in Figure 5, a thread-aware hardware prefetcher that learns using the LLC miss address stream may detect that Load-A has strided accesses. If cache block addresses are used for training it is likely that no stride will be detected since Load-A is small strided and all accesses by a thread may be to the same cache block. Since Load-B does not have any strided behavior, a thread-aware prefetcher will be unable to prefetch the addresses accessed by Load-B. Also, Load-B is likely to have more cache misses than Load-A, and a thread-aware hardware prefetcher that does not prefetch Load-B may not show much benefit. Even complex prefetchers such as a Markov prefetcher [41] or a Content-Directed Data prefetcher (CDP) [19] would be unable to detect the relation between Load-A and Load-B. [3]

---

[2] multi-threading is not sufficient to hide memory access latency for the benchmarks considered

[3] A CDP looks for virtual addresses in the cache blocks and in this case the blocks only contain the index of the edge element to be accessed.

#### *3.2.1.3 Software Prefetching*

For the access pattern in Figure 5 software prefetching can be performed in several different ways:

**Prefetching Load-A only** Load-A can be prefetched into either the data cache or into a register. This would not result in significant performance benefit, since Load-B is the performance critical load.

**Prefetching Load-B into cache** Due to the dependency of Load-B on it, Load-A must be prefetched into a register to prefetch Load-B i.e., a binding prefetch must be issued for Load-A to prefetch Load-B. However, prefetching Load-B into the cache could increase memory accesses due to eviction of prefetched data and could hurt performance.

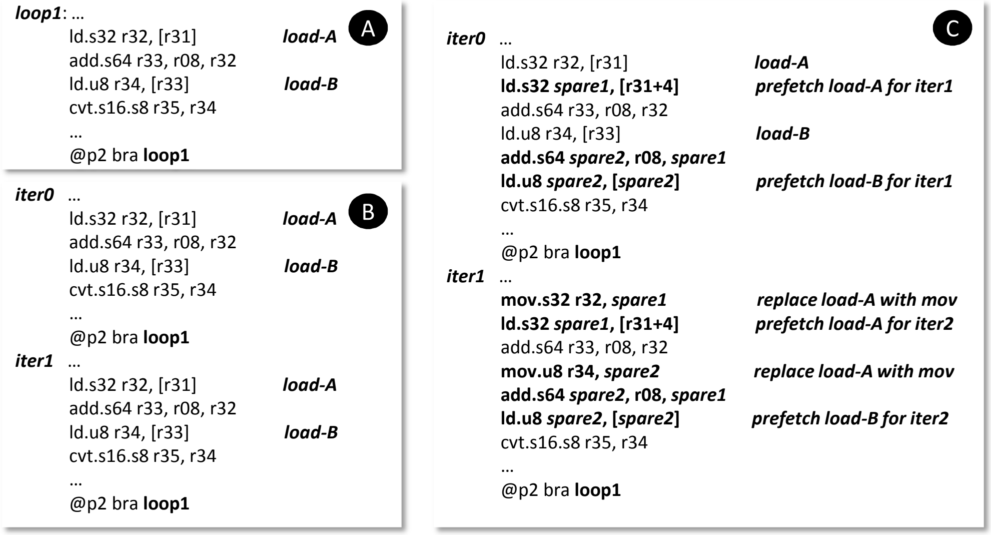
**Binding prefetches for both Load-A and Load-B** This approach is similar to the mechanism we propose (Section 3.3) with some subtle differences. Firstly, software prefetching is an all or nothing approach. For software prefetching to be performed, sufficient free registers must be available to do binding prefetching for all threads of all blocks that can be assigned to a SM at a given time, otherwise occupancy will be reduced. While software prefetching reduces occupancy if insufficient registers are available, our mechanism can be adapted to do prefetching for only a subset of the warps that will be allocated to a SM without reducing occupancy. However, we do not evaluate such an extension to our mechanism in this thesis. Also, software prefetching uses a fixed prefetch distance, while our mechanism can adapt the prefetching distance dynamically on a per loop invocation basis for each warp.

### 3.2.2 Loop Unrolling for graph algorithms

Loop unrolling with load hoisting could also be performed for the access pattern in Figure 5. However, the degree of unrolling should be determined statically. Furthermore, all threads in a kernel will have the same degree of unrolling because they all execute the same static code. Also, since loop unrolling replicates the entire loop body, it is likely to increase register usage and could reduce occupancy. As explained, the proposed mechanism can vary the prefetch distance dynamically and can be easily modified to prefetch for only a subset of the warps without reducing occupancy if the number of spare registers are not sufficient.

## *3.3 Prefetching into spare registers*

In this section the proposed mechanism is explained.



```
loop1: ...
       ld.s32 r32, [r31]             load-A
       add.s64 r33, r08, r32
       ld.u8 r34, [r33]              load-B
       cvt.s16.s8 r35, r34
       ...
       @p2 bra loop1
```

```
iter0  ...
       ld.s32 r32, [r31]             load-A
       add.s64 r33, r08, r32
       ld.u8 r34, [r33]              load-B
       cvt.s16.s8 r35, r34
       ...
       @p2 bra loop1
iter1  ...
       ld.s32 r32, [r31]             load-A
       add.s64 r33, r08, r32
       ld.u8 r34, [r33]              load-B
       cvt.s16.s8 r35, r34
       ...
       @p2 bra loop1
```

```
iter0  ...
       ld.s32 r32, [r31]             load-A
       ld.s32 spare1, [r31+4]        prefetch load-A for iter1
       add.s64 r33, r08, r32
       ld.u8 r34, [r33]              load-B
       add.s64 spare2, r08, spare1
       ld.u8 spare2, [spare2]        prefetch load-B for iter1
       cvt.s16.s8 r35, r34
       ...
       @p2 bra loop1
iter1  ...
       mov.s32 r32, spare1           replace load-A with mov
       ld.s32 spare1, [r31+4]        prefetch load-A for iter2
       add.s64 r33, r08, r32
       mov.u8 r34, spare2            replace load-A with mov
       add.s64 spare2, r08, spare1
       ld.u8 spare2, [spare2]        prefetch load-B for iter2
       cvt.s16.s8 r35, r34
       ...
       @p2 bra loop1
```

Figure 6: Example loop and instruction stream for loop without and with prefetching into spare registers

The proposed mechanism is based in hardware with some assitance from the programmer/compiler and is explained in Sections 3.3.2.1- 3.3.2.5. A mechanism that relies entirely on the compiler for detecting the target loads is explained in Section 3.3.2.6.

### 3.3.1 Overview

We first identify a target loop and the target loads within the loop. The target loads usually occur in pairs - an independent load with strided accesses (similar to Load-A in Figure 5) and a load dependent on the independent load (similar to Load-B in Figure 5). We refer to a pair of loads with one independent load and a dependent load as a load-pair. The independent load is called the head-load and the dependent load is called the tail-load.

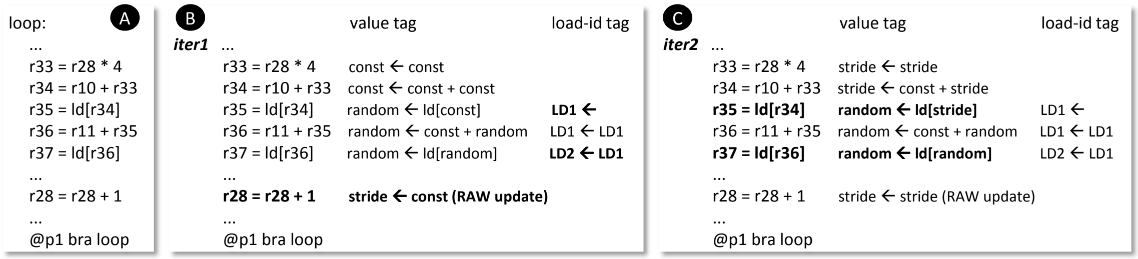
In each loop iteration we inject instructions to prefetch data that will be accessed by the target loads in subsequent loop iterations. The injected instructions prefetch data into available spare registers (as well as cache). And the load instructions whose addresses have been prefetched are translated into mov instructions that copy the prefetched data value into the destination register of the original load instruction. The mechanism is designed to ensure that the prefetch addresses are correct, therefore we can prefetch data into registers and use the prefetched data. The injection of the prefetching instructions can be explained with the help of the code in Figure 6A which is the PTX code for the example in Figure 5. This injection process could also be thought of as doing software pipelining in hardware - in each iteration we inject loads that are used by subsequent iterations. However, the loop does not have any prologue and epilogue.

The instruction stream under normal execution is shown in Figure 6B and the instruction stream with the proposed mechanism is shown in Figure 6C. In the original instruction stream both Load-A and Load-B are used immediately after being fetched. For the applications considered, load wait times are significant, especially for uncoalesced loads such as Load-A and Load-B and multi-threading is unable to hide memory access latency fully. However, for the instruction stream with prefetching, the wait time for Load-B would be greatly reduced since it is prefetched one iteration before it is used. Since Load-A is strided with a small stride, accesses generated by Load-A in successive iterations for a thread are more likely to be cache hits in either L1 or L2 cache than accesses generated by Load-B. Overall, the wait for Load-A may not be as significant as the wait for Load-B and reducing the wait time for

Load-B helps improve performance more. And using registers for storing prefetched data eliminates the problem of eviction that would plague mechanisms that insert prefetched data into the cache only.

### 3.3.2 Details

Before injecting instructions for prefetching data the mechanism has to identify the target loads for prefetching.



```
loop:
   ...
   r33 = r28 * 4
   r34 = r10 + r33
   r35 = ld[r34]
   r36 = r11 + r35
   r37 = ld[r36]
   ...
   r28 = r28 + 1
   ...
   @p1 bra loop
```

```
                          value tag                     load-id tag
iter1 ...
      r33 = r28 * 4       const ← const
      r34 = r10 + r33     const ← const + const
      r35 = ld[r34]       random ← ld[const]            LD1 ←
      r36 = r11 + r35     random ← const + random       LD1 ← LD1
      r37 = ld[r36]       random ← ld[random]           LD2 ← LD1
      ...
      r28 = r28 + 1       stride ← const (RAW update)
      ...
      @p1 bra loop
```

```
                          value tag                     load-id tag
iter2 ...
      r33 = r28 * 4       stride ← stride
      r34 = r10 + r33     stride ← const + stride
      r35 = ld[r34]       random ← ld[stride]           LD1 ←
      r36 = r11 + r35     random ← const + random       LD1 ← LD1
      r37 = ld[r36]       random ← ld[random]           LD2 ← LD1
      ...
      r28 = r28 + 1       stride ← stride (RAW update)
      ...
      @p1 bra loop
```

Figure 7: Detecting target loads and their properties

#### *3.3.2.1 Identifying target loads for prefetching*

We consider only loads from arrays that are tagged "const __restrict" [55] by the programmer. Such arrays are guaranteed to be read-only within the scope of the kernel. Thus the mechanism requires the programmer to tag read-only arrays and we can be confident that the addresses read by the target loads are not written to by other loads. We assume that such loads use a special load opcode. Note that this feature already exists in current CUDA releases.

The first step in identifying target loads is to detect loops. A loop is detected when a backward branch is encountered by a warp. The branch PC is the end of the loop and the branch target is the start address of the loop. To identify the target loads within a loop we used a tag-based hardware mechanism. Our approach uses two sets of tags - *value* tags to identify whether loads are strided or otherwise; *load-id* tags to detect dependencies between loads. There is one set of tags of each type per

SM (not per warp) with one tag for each architecture register; each tag is only 2-bits wide. Figure 7 shows an example loop and how the load properties are detected.

Once a loop is detected, the instruction decoder selects a warp that is beginning a new loop iteration for identifying the target loads (if any) in the loop. At the start of the loop iteration of the selected warp, the *value* tags of all registers are set to *constant*. The tags of the destination register ids for instructions that are commonly used in address generation are updated as shown in Table 1 (update rules for other instructions are not shown). At the end of the loop when the loop index variable is updated, a WAR update (register is read and written in the same instruction) with a constant increment is detected and the *value* tag of the register id holding the loop index variable is updated as *stride* (Figure 7B). When the next loop iteration is commenced, the updated tag values are used and we are able to detect whether a load is strided or not by the end of the third loop iteration (one iteration for detecting loop and two iterations for detecting load properties) as shown in Figure 7C.

The *load-id* tags are used to track dependences between loads. When a value is loaded into a register from memory, the register is assigned a *load-id* tag and the tag is propagated. If the source of a load has a valid *load-id* tag then the load is dependent on the load whose tag is in the source tag.

The tag-based mechanism has to ensure that the prefetched addresses will be always correct since we are prefetching data into registers and using it.

Table 1: Prefetching: Tag propagation for target load detection

| Dest Tag | Opcode | Op-0 RAW update? | Op-0 Tag | Op-1 Tag |
|---|---|---|---|---|
| random | ld | X | X | X |
| const | add/sub/mul/shl | 0 | const | const |
| stride | add/sub/mul/shl | 0 | stride | const |
| stride | add/sub/mul/shl | 1 | const/stride | const |

#### *3.3.2.2 Spare registers for prefetch destinations*

The two sources of spare registers were explained in Section 3.2 and in this work we only consider unallocated registers for holding prefetch data. Table 4 shows the number of registers required by each thread ($\mathrm{MaxReg_s}$) for several graph algorithm kernels and also the maximum number of registers ($\mathrm{MaxReg_d}$) that can be allocated to each thread without reducing the occupancy of the kernel. When allocating registers, we allocate $\mathrm{MaxReg_d}$ registers to each thread, instead of allocating only $\mathrm{MaxReg_s}$ registers.

#### *3.3.2.3 Prefetch distance*

In the proposed mechanism prefetch distance refers to how many iterations in advance are prefetches inserted. Either a fixed prefetch distance could be used for all loop invocations or the distance could be varied for each invocation.

**Fixed prefetch distance** If sufficient registers are available, the simplest form of our mechanism inserts prefetches with a distance of one. To prefetch with a distance of *pref_dist*, for each load we need *pref_dist* number of spare registers to hold the prefetched value(s).

**Variable prefetch distance** The prefetch distance could be varied per loop invocation depending on the average number of loop iterations per thread as shown in Algorithm 1.[4] A larger prefetch distance potentially reduces the wait time for the tail-load. Section 3.3.2.4 explains the calculation of the number of loop iterations of each thread.

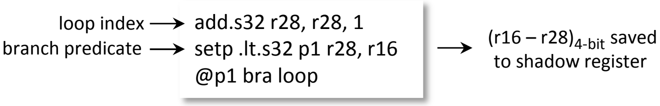
If the spare registers required by the calculated prefetch distance are not available, the prefetch distance is reduced until the required number of spare registers are available.

---

[4] we use the following values for the constants in Algorithm 1: low = 1, med = 2, high = 3, MID = 4, HIGH = 6

**Listing 1** Calculating the prefetch distance

```
if $avg_{iter\_count} < MID$ then
    pref_distance = low
else if $avg_{iter\_count} < HIGH$ then
    pref_distance = med
else
    pref_distance = high
```



```
add.s32 r28, r28, 1
setp .lt.s32 p1 r28, r16
@p1 bra loop
```

Figure 8: Determining iteration counts for each thread

#### *3.3.2.4 Determining number of loop iterations*

We use a shadow register of 4-bits per thread for each warp to hold the iteration count for each thread. In GPUs, conditional branches are implemented as predicated jump instructions. For the target loop, we detect the *setp* instruction that sets the predicate register on which the loop-ending branch is predicated. The setp instruction usually evaluates a "less than" condition with the current value of the loop index and the final value of the loop index as its operands. When the setp instruction is evaluated, we also evaluate a modified *sub* operation with the same operands as the setp instruction to obtain the pending iteration count for each active thread in the warp as shown in Figure 8. The average number of loop iterations per active thread can be calculated from these 4-bit shadow registers. The average iteration count is calculated only once while the shadow register is updated every iteration.

#### *3.3.2.5 Inserting Prefetches*

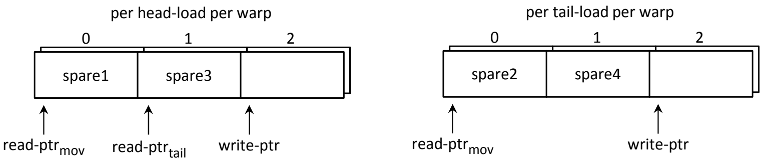


Figure 9: Tracking registers allocated for prefetches

Once the prefetch distance has been calculated by the hardware, instructions are injected into the pipeline to prefetch data into spare registers. We accomplish this using circular lists called *AllocatedReg* lists (Figure 9), which are of size three (the largest allowed prefetch distance) and hold the register ids allocated for holding prefetched data; each warp requires one list per load. The list for a head-load has three pointers: *write-ptr* which points to the next location to be used - the id of the next allocated spare register is updated in this location, $\textit{read-ptr}_{mov}$ which points to the next location to be read by the *mov* instrucutions replacing the *ld* instructions and $\textit{read-ptr}_{tail}$ which points to the next location to be read by the instructions inserted for prefetching the tail-loads. Each tail pointer also has a similar circular list but with only two pointers - *write-ptr* and $\textit{read-ptr}_{mov}$. After inserting prefetches for *pref_dist* number of iterations, the head and tail loads in subsequent iterations are converted into *mov* instructions. Figure 6 shows the insertion of head and tail prefetches and the conversion of loads into mov instructions. Spurious prefetch instructions (when threads have exited the loop) are avoided by using the shadow registers containing the iteration counts to control the generation of prefetch instructions and their active masks.

#### *3.3.2.6 Compiler-assisted mechanism*

The hardware-only mechanism requires three non-overlapping loop iterations (the iterations need not be from the same warp) to detect the target loads. Due to the time overhead of this mechanism we are unable to insert prefetches for a few iterations for the initial set of warps assigned to each SM. In addition, to eliminate spurious prefetch memory loads, the hardware-only mechanism needs to know the iteration counts of the threads before inserting prefetches, and determining iteration counts takes one iteration for each warp. Thus the hardware prefetcher misses several prefetching opportunities. Instead, if we take the aid of the compiler, we can insert prefetches from the first loop iteration of each warp. The compiler can not only identify the target loads and their properties but can also mark the instruction which tests the loop condition before the loop is entered. This instruction would be a *setp* instruction which could be used to determine loop iteration counts before entering the loop (similar to Section 3.3.2.4). Thus a compiler-assisted mechanism can insert prefetches starting from the first iteration itself.

Table 2: Prefetching: Hardware Cost

| Component | Size | Cost |
|---|---|---|
| Value tags | 2-bits * 63 | 126 bits |
| Load-id tags | 2-bits * 63 | 126 bits |
| $\text{MaxReg}_\text{s}$ | 6 bits | 6 bits |
| $\text{MaxReg}_\text{d}$ | 6 bits | 6 bits |
| Prefetch distance counter | 2 bits * 48 warps | 12 bytes |
| Shadow iteration count reg | 4 bits * 32 threads * 48 warps | 768 bytes |
| AllocatedReg list | | |
| Head-load/warp | 6 bits/entry * 3 + 2 bits/ptr * 3 | 3 bytes |
| Tail-load/warp | 6 bits/entry * 3 + 2 bits/ptr * 2 | 2 bytes + 6 bits |
| 2 load-pairs (all warps) | (11 bytes + 4 bits) * 48 warps | 552 bytes |
| Total | | 1365 bytes<br>(1.04% of reg file) |

### *3.3.2.7 Total Hardware Cost*

Table 3.3.2.6 shows the total hardware cost of the hardware-only mechanism with support for variable prefetch distances while assuming a maximum of 63 registers per thread and a maximum of 48 warps per SM in accordance with the Fermi configuration [55]. The compiler-assisted mechanism would not require the value tags and the load-id tags used for target load detection.

## *3.4 Methodology*

We emulate several graph applications with different inputs (Section 3.4.1) using GPUOcelot [23] and collect their traces which are simulated using MacSim [5], a

Table 3: Prefetching: Simulated GPGPU Architecture

| | |
|---|---|
| Num. of cores | 16 |
| Front End | Fetch width: 2 warp-instruction/cycle;<br>4KB I-cache; stall on branch; 5 cycle decode |
| Execution core | 2 warp-instructions/cycle, Frequency: 1.2 GHz;<br>16-wide SIMD, in-order scheduling, latencies are modeled according to the CUDA manual [56];<br>128 KB register file |
| On-chip caches | 16/48 KB software managed cache, 16 loads/2-cycle;<br>4-cycle 6-way 48KB L1 cache, 16 loads/2-cycle<br>or 4-cycle 4-way 16KB L1 cache, 16 loads/2-cycle;<br>4-cycle 8 KB constant cache, 16 loads/2-cycle;<br>4-cycle 8 KB texture cache, 16 loads/2-cycle;<br>100-cycle round trip access to 768 KB L2 cache; |
| DRAM | 2 KB page, 16 banks with page interleaving,<br>128 reqs/bank, 8 channels, 100.8 GB/s<br>400 cycle minimum round-trip time;<br>tCL=20, tRCD=28, tRP=12 (GDDR-5 [3]) |

cycle-level heterogeneous architecture simulator. The simulated architecture, shown in Table 3 is based on NVIDIA's Fermi [54].

### 3.4.1 Benchmarks and inputs

We evaluate kernels from several fundamental graph algorithm implementations; the evaluated kernels and their attributes such as register usage per thread and the maximum number of registers that can be allocated per thread without reducing occupancy are shown in Table 4.[5] H-BFS, H-SSSP and H-MST are taken from implementations provided by Harish, Vineet et al. [31, 76, 32] and LS-BFS, LS-SSSP and LS-MST are taken from LonestarGPU 0.9 [9]. We implement ST-CON [31] based on prior work. For each application we generate traces for multiple input graphs and since graph algorithms are iterative, each kernel is invoked multiple times during a single application run. Some characteristics of the different input graphs are shown in Table 10. Graph1M is obtained from the Parboil benchmark suite [73] and is a random graph

---

[5] We include only algorithms that have the access pattern described in Section 3.2.1.1

Table 4: Prefetching: Evaluated kernels

| Kernel | Reg/Thread - Actual, Max | Unused Reg space |
|---|---|---|
| *Kernel* from<br>Breadth First Search [31](H-BFS) | 15, 20 | 38KB |
| *DijkstraKernel1* from<br>Single Source Shortest Path [31] (H-SSSP) | 17, 20 | 26KB |
| *Find_Min_Edge_Weight* from<br>Minimum Spanning Tree [76](H-MST) | 21, 32 | 44KB |
| *drelax* from<br>Breadth First Search [9](LS-BFS) | 19, 32 | 52KB |
| *drelax* from<br>Single Source Shortest Path [9](LS-SSSP) | 22, 32 | 40KB |
| *boruvkaone* from<br>Minimum Spanning Tree [9](LS-MST) | 13, 32 | 76KB |
| *kernel* from<br>S-T Connectivity [31] (STCON) | 23, 32 | 36KB |

Table 5: Prefetching: Input graph characteristics

| Input | #Vertices | #Edges | Avg. Degree | BFS Depth |
|---|---|---|---|---|
| Graph1M | 1000000 | 5999970 | 5.99 | 12 |
| RMAT20 | 1048576 | 8259994 | 7.88 | 13 |
| R42e20 | 1048576 | 4194304 | 4 | 16 |

with an average vertex degree of 6, while RMAT20 and R4_2e20 are obtained from LonestarGPU 0.9 [9]. RMAT20 represents graphs found in the real world [11], it has few vertices with a large degree and a large number of vertices with a small degree; while R4_2e20 is a random graph with a average vertex degree of 4.

## *3.5 Results*

We evaluate different forms of the proposed prefetch mechanism - hardware only with a prefetch distance of 1 (SPREF1), compiler-assisted with a prefetch distance of 1 (C-SPREF1), hardware-only with a dynamic prefetch distance (SPREF) and compiler-assisted with a dynamic prefetch distance (C-SPREF). We compare our

Table 6: Prefetching: Evaluated hardware prefetchers

| Prefetcher | Description | Configuration |
|---|---|---|
| Stride | RPT Stride Prefetcher [36] | 1024-entry, 16 region bits |
| Stream | Stream Prefetcher [72, 69] | 512-entry |
| GHB | AC/DC GHB Prefetcher [39, 53] | 1024-entry GHB, 12-bit CZone, 128-entry Index Table |

mechanism against Stride, Stream and GHB hardware prefetchers; the configurations for these prefetchers are shown in Table 6. All prefetch mechanisms insert prefetched data into both L2 and L1 caches. Only the proposed mechanisms insert prefetched data into registers as well. Stride, Stream and GHB prefetchers are trained on L1 (and L2) access streams to improve their learning.

Since a kernel is invoked multiple times during the same application run (with a given input), for the sake of readability we show results for the different kernel invocations during an application run using a single value. We show the harmonic mean of the ratio of the IPC with the evaluated mechanism to the IPC of the baseline (i.e., no prefetching) across all kernel invocations for a given input. In the bar graphs, the harmonic mean value for a given input is labeled wth the name of the input graph. We also show the harmonic mean of the IPC ratios across all kernel invocations for all inputs in the bar graphs; these bars are labeled HMean in the graphs. Our evaluation considers only those kernel invocation instances that have about 1 million warp instructions or more. The IPC metric that is used is a weighted IPC metric calculated across all GPU cores. For the proposed mechanism, when calculating the IPC we ignore the additional instructions introduced by the hardware (but these additional instructions are simulated fully), thus the total instruction count for the proposed mechanism is the same as the baseline.

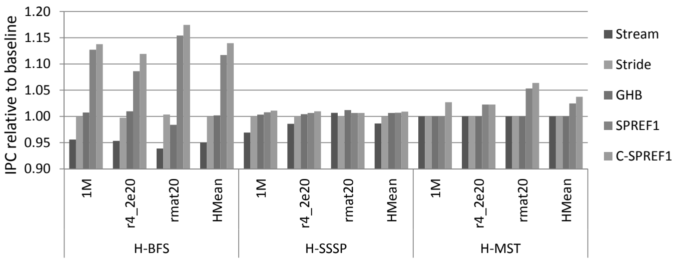


Figure 10: Prefetching: IPC improvements for H-BFS, H-SSSP, H-MST

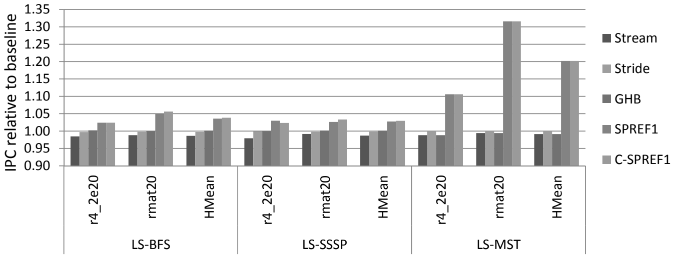


Figure 11: Prefetching: IPC improvements for LS-BFS, LS-SSSP, LS-MST

### 3.5.1 Performance improvement with different mechanisms

Figures 10, 11 and 12 show the performance improvement provided by Stride, Stream, GHB, SPREF1 and C-SPREF1 for the evaluated kernels with different inputs. While the previously proposed mechanisms either show no performance improvement or minor performance improvement for most of the evaluated kernels, SPREF1 and C-SPREF1 show significant performance improvement for different kernels with different inputs. The average performance improvement shown by SPREF1 and the other mechanisms across the evaluated kernels is shown in Table 7. On average, the improvement provided by SPREF1, C-SPREF1, SPREF and C-SPREF across all

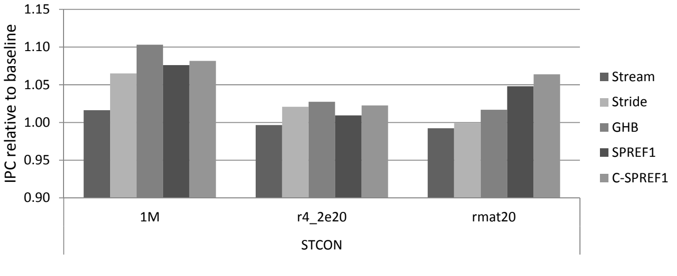


Figure 12: Prefetching: IPC improvements for STCON

Table 7: Prefetching: IPC improvements over baseline for evaluated mechanisms

| | H-BFS | H-SSSP | H-MST | LS-BFS | LS-SSSP | LS-MST | STCON | |
|---|---|---|---|---|---|---|---|---|
| | HM | HM | HM | HM | HM | HM | HM | HM (all) |
| Stream | 0.95 | 0.99 | 1.00 | 0.99 | 0.99 | 0.99 | 0.99 | 0.98 |
| Stride | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| GHB | 1.00 | 1.01 | 1.00 | 1.00 | 1.00 | 0.99 | 1.02 | 1.00 |
| **SPREF1** | 1.12 | 1.01 | 1.02 | 1.04 | 1.03 | 1.20 | 1.04 | 1.06 |
| **C-SPREF1** | 1.14 | 1.01 | 1.04 | 1.04 | 1.03 | 1.20 | 1.05 | 1.07 |
| **SPREF** | 1.12 | 1.01 | 1.05 | 1.04 | 1.03 | 1.23 | 1.04 | 1.07 |
| **C-SPREF** | 1.15 | 1.01 | 1.05 | 1.05 | 1.04 | 1.25 | 1.05 | 1.08 |
| SPREF1-C | 1.06 | 0.99 | 0.83 | 1.01 | 0.99 | 0.88 | 1.03 | 0.96 |
| Base-64KB | 1.02 | 1.04 | 1.02 | 1.07 | 1.10 | 1.03 | 1.00 | 1.04 |

kernel invocations of the evaluated kernels for all inputs are 6%, 7%, 7% and 8%.

### 3.5.2 Analysis of performance improvement

*3.5.2.1 Stride, Stream and GPH prefetchers*

Previously proposed mechanisms have difficulty in learning the addresses to be prefetched. For most kernel invocations, Stride prefetcher sends out very few prefetch requests because most loads have zero stride (head-loads) or random strides and the Stride prefetcher is unable to learn any patterns. On the otherhand, Stream and GHB are able to detect some patterns and send out significant prefetches for most kernel invocations. However, the fraction of useful prefetches is very small, especially for Stream, and hence we are unable to see much benefit from these mechanisms. Also for Stream and GHB prefetchers, most of the prefetches get evicted without being used.

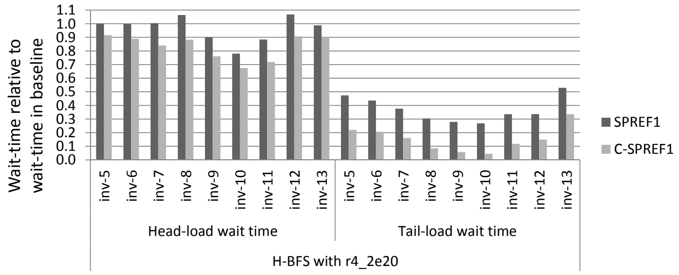


Figure 13: Prefetching: Wait times for H-BFS with r4_2e20 input

#### *3.5.2.2 SPREF1 and variants*

SPREF1 and related prefetchers mainly improve performance by reducing the wait time for load-pairs, especially the dependent load. Here, wait time refers to the duration for which the instruction dependent on the load waits for the load to complete. The wait times for tail-loads are typically longer than the wait times for the head-loads and multithreading hides some of the wait time of a load. Due to the reduction in wait times provided by SPREF1 (and variants) the number of core idle cycles are also reduced resulting in performance improvement. As an example, Figure 15 shows the wait times relative to the wait times for the baseline for the head-load and the tail-load for H-BFS with r4_2e20 input for different kernel invocations during the same application run. While the wait time for the head-load has reduced in some cases and increased in others, the wait time for the tail-load has reduced significantly and this provides performance benefit. Below we briefly analyze the performance improvement shown by the different kernels while analyzing the performance of H-BFS, especially H-BFS with r4_2e20 in detail.

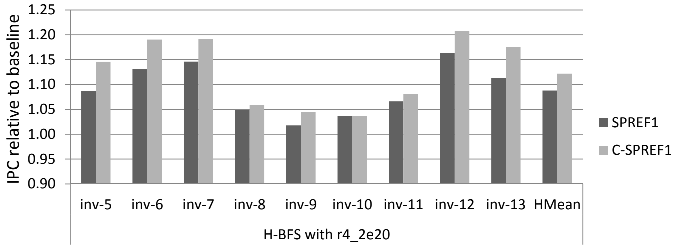


Figure 14: Prefetching: IPC improvements for H-BFS with r4_2e20 input

**H-BFS** H-BFS has one load-pair and across all inputs used, SPREF1 and C-SPREF1 show an average performance improvement of 12% and 14% for this kernel. In H-BFS and other kernels a notion of active/enabled vertex is used (except LS-BFS and LS-SSSP, in these kernels all vertices are active in every kernel invocation); in a kernel invocation, a vertex processes it neighbors only if the vertex is active/enabled. In many iterations (especially iterations at the start and at the end of the algorithm) often only a few vertices are active, these invocations tend to show low improvement since there are only a few opportunities for prefetching (this is applicable to other kernels as well).

Figure 14 shows the performance improvement provided by SPREF1 and C-SPREF1 for H-BFS with r4_2e20 input across different kernel invocations in the same application run. Since the vertices that are active during each step (invocation) of BFS are different, the performance improvement also varies from one invocation to another. In general the performance improvement increases with the number of active vertices and the number of neighbors per active vertices. However, when the number of active vertices is high, sometimes preformance improvement could be reduced due to contention in the memory hierarchy as in the case of invocations 9, 10 and 11 in Figure 14.

Figure 14 also shows that C-SPREF1 provides slightly higher performance improvement than SPREF1. Since C-SPREF1 does not have the overhead of detecting target loads and does not take an additional loop iteration to determine loop iteration counts before inserting prefetches, it has additional opportunities for inserting prefetches and thus can show more benefit than SPREF1.

**H-SSSP** This kernel has only head-loads and no tail-loads. One of the head-loads has a dependent memory access similar to a tail-load, however, the memory location is read and updated via an atomic instruction, hence the location cannot be prefetched into a register or the L1 cache and we are unable to improve performance significantly.[6] For this kernel we only prefetch the head-loads into registers.

**H-MST** In addition to the detected load-pair, this kernel has several other loads and one of them is dependent on the tail-load in the identified load-pair. Reducing the wait time of only the load pair without reducing the wait time of other loads does not improve performance much and thus we see small performance gains with

---

[6] atomic instructions update memory locations directly in the L2 cache

SPREF1 and C-SPREF1.

**LS-BFS and LS-SSSP** Similar to H-SSSP, these kernels have only a head-load and the dependent memory access is performed atomically, hence the only the head-load can be prefetched into registers (and cache) and this does not provide much benefit.

**LS-MST, STCON** All these kernels have a load pair and the kernels are similar to H-BFS. In each kernel invocation a set of active/enabled vertices traverse their neighbor list. LS-MST tends to show good benefit while STCON terminates quickly within a few iterations for the evaluated inputs and hence does not have many active vertices during each invocation and therefore shows slightly lower performance improvement.

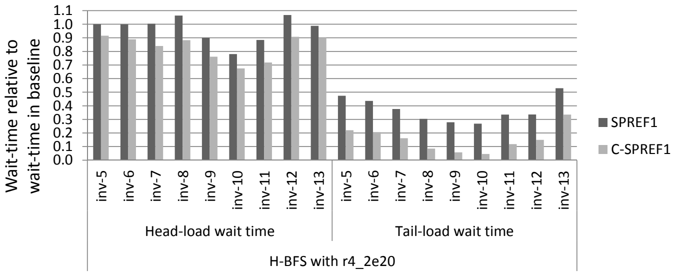


Figure 15: Prefetching: Wait times for Head and Tail loads for LS-BFS

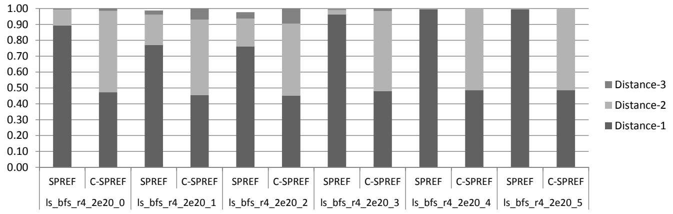


Figure 16: Prefetching: Fraction of different prefetch distances for LS-BFS

### 3.5.3 Compiler-assisted mechanism and dynamic prefetch distance

**Compiler-assisted mechanism** Though C-SPREF1 and C-SPREF, which are compiler assisted mechanisms perform slightly better than the hardware-only mechanisms

(SPREF1 and SPREF) the difference between them is not very significant (Table 7). For the hardware-only mechanisms, a warp must execute at least one iteration before the iteration counts of the warp's threads can be determined. Due to this overhead, SPREF1 and SPREF often have fewer opportunities for prefetching than their compiler-assisted counterparts. For the same reason, SPREF is likely to execute with a smaller dynamic prefetch distance than C-SPREF.

**Dynamic prefetch distance** Although using a dynamic prefetch distance does not improve performance by much on average, there are instances which show improvement as shown by the increase in the maximum IPC improvement for H-BFS and LS-MST (SPREF vs SPREF1 and C-SPREF vs C-SPREF1).

### 3.5.4 Benefit from prefetching into registers

We analyze the performance improvement due to prefetching into spare registers in addition to the cache. The row labeled SPREF1-C shows the performance improvement due to a mechanism that is the same as the proposed hardware mechanism, but does not insert the tail prefetches into a spare register but inserts them only into the cache. The head-loads would have to be inserted into spare registers since they are needed to calculate the prefetch address for the tail-loads. For kernels that do not have tail-loads, the head-loads are prefetched but inserted into the cache only. The performance of SPREF1-C in comparison to SPREF1 is varied. For STCON the performance of SPREF1-C is similar to SPREF1. But for kernels such as H-BFS, H-MST and LS-MST there is severe performance degradation for several kernel instances.

As mentioned, inserting prefetched data into cache only has a downside - prefetched data could get evicted from L1 requiring additional memory requests to the lower levels of the memory hierarchy, while the L1 data cache will have more requests to contend with (both prefetch and demand requests) when compared to SPREF1.

### 3.5.5 Evaluation with a larger cache

We increase the cache size for the baseline to 64KB for all kernels to see whether a baseline with a larger cache can provide the same performance as SPREF1 on a GPU with a normal sized cache (48KB/16KB for Fermi architecture depending on the kernel). For all kernels, combining the data cache and the consumed spare registers, SPREF1 does not consume more than 64KB of space. The row labeled Base-64KB in Table 7 shows the IPC improvement obtained by increasing the cache size to 64KB. While there is an improvement in performance for most kernels when compared to the baseline, the improvement is considerable for LS-BFS, and LS-SSSP. Compared to SPREF1, Base-64KB performs better only for H-SSSP, LS-BFS and LS-SSSP. As mentioned above, in case of these kernels, there are only head-loads and hence SPREF1 cannot improve their performance significantly. In case of LS-SSSP and LS-BFS, SPREF1 (and baseline) uses only a 16KB data cache, because these kernels use 48KB of the 64KB on-chip memory as Shared Memory. This results in a 64KB data cache providing significant additional cache space over what is used by SPREF1 and thus helps obtain a much higher performance. However, on average, our proposed mechanism shows better performance than a GPU with a larger cache by providing effective prefetching mechanisms.

We also evaluate SPREF1 on a machine with a unified on-chip storage of 192KB (same on-chip storage as Fermi). In such a machine the register file, Shared Memory and data cache all share the same physical storage. After allocating registers and Shared Memory as per the kernel reqiurements, the rest of the on-chip storage is allocated as data cache. In case of SPREF1 we allocate registers plus spare registers for prefetching and Shared Memory as per kernel requirements before allocating the remaining space as data cache. For such a machine configuration, SPREF1 performs better than no prefetching by 6% on average while performing slightly worse for H-MST and H-SSSP. Thus, though SPREF1 duplicates some data it makes effective use of on-chip storage.

## *3.6 Summary*

In this mechanism, we take advantage of unallocated registers in GPUs to design a prefetching mechanism for graph algorithms on GPUs which tend to be memory intensive with data dependent memory accesses. Using a tag-based hardware mechanism, we identify target load-pairs within a loop for prefetching. Once the target loads are identified, the mechanism injects instructions to prefetch data accessed in subsequent loop iterations into unallocated registers. The loads whose addresses have been prefetched are converted into mov instructions to use the prefetched data. The mechanism has the benefit that even if the prefetched data gets evicted from cache, the data would still be available in the registers. Also, our mechanism can vary the prefetch distance on a per loop invocation basis and we also propose a software-hardware cooperative mechanism besides the hardware only mechanism. The hardware-only mechanism provides an average benefit of 7% and a maximum benefit of 50% across several graph kernels with different inputs. The numbers of the software-hardware cooperative prefetching mechanism are 8% (average) and 51% (maximum). Though in this thesis the prefetching mechanism has been applied to graph algorithms, the idea of prefetching into unused registers can be applied to other algorithms as well.

# CHAPTER IV

# EXPLORING THE CACHE HIERARCY DESIGN FOR GRAPH ALGORITHMS

## *4.1 Introduction*

The design of the cache hierarchy for GPGPU applications has been much less explored than it has been for CPU applications. Early GPUs did not include any hardware managed caches since GPGPU applications were considered to be streaming data without any locality in accesses. As more applications were ported onto the GPU, hardware caches appeared as some of the new applications could benefit from the presence of a cache. These caches have tended to be small, with the cache space available per thread being of the order of few bytes in comparison to KBs of cache space available for a CPU thread. The GPU caches are designed for capturing any short-term temporal and spatial locality in accesses. Due to the large number of thread executing concurrently, a cache for capturing long-term locality would have to be very large. Despite being small, caches in current GPUs have been shown to be beneficial for certain GPGPU applications. In this chapter we explore several different aspects of cache hierarchy design for graph algorithms on GPUs.

The first design aspect we explore is the inclusion property of the cache hierarchy; current GPUs include a two-level, non-inclusive cache hierarchy. Compared to inclusive/non-inclusive caches, an exclusive cache hierarchy with the same sized caches provides effectively more cache space. The downside of an exclusive cache is the increase in the on-chip traffic. Alternatives such as Flexclusion [68] have been proposed to switch between non-inclusion and exclusion depending on whether exclusion provides benefit. In addition to exploring these two alternatives to a non-inclusive cache, we examine memory region-based exclusion for graph algortithms. Memory region-based exclusion has the potential for increasing performance relative to non-inclusion and reducing on-chip traffic compared to exclusion and Flexclusion.

The second aspect we explore is cache bypassing. As we see in Section 4.3, graph algorithms include accesses with locality and there are accesses without locality as well. Often, the cache sizes are too small to exploit the existing locality. In such situtations, cache bypassing can be a viable technique for improving caching efficiency. In addition to exploring the effectiveness of several different cache bypass mechanism that were previously proposed for CPUs we also examine a bypass extension to the memory region based exclusion mentioned above.

The third aspect we explore is a dynamic granularity cache hierarchy. Traditional cache hierarchies operate at a cache block granularity - caches store full cache blocks and on a cache miss a full cache block is requested from the next level of the memory. However, often, only a portion of the full cache block that is stored in a cache is used. This means DRAM and interconnect bandwidth is wasted in fetching the portions of the cache that are not accessed. In addition, fetchching full cache blocks increases the queueing delay for other requests in the system. We examine whether a dynamic granularity cache hierarchy which varies the granularity of requests (size of requests) sent to the lower levels of the memory hierarchy can be beneficial for graph algorithms on GPUs. We evaluate several mechanisms including a per-PC history based mechanism.

## *4.2 GPU Cache Hierarchy*

GPUs include a two-level cache hierarchy (Figure 17). In the Fermi architecture, each GPU has 64 KB on-chip storage for data. This storage can be partitioned into 16KB-48KB or 48KB-16KB between the data cache and a scratchpad. For writes to global data the L1 cache uses the write-evict policy i.e., any stores that hit in the L1 caches

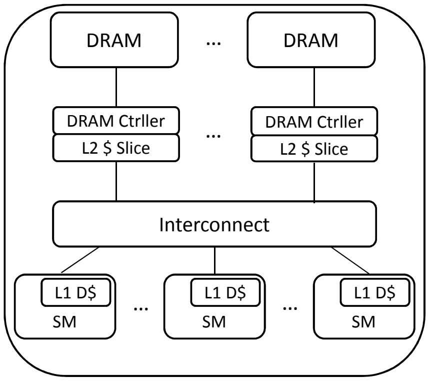


Figure 17: GPGPU Memory Hierarchy

cause eviction of the cache block from the L1; write misses use write no-allocation. L1 cache misses are forwarded to a unified 768 KB L2 cache that is shared by all the GPU cores. The L2 cache is split into 6 tiles, with each tile connected to a DRAM controller. We evaluated between a write-back cache and write-through cache with eviction for the L2 and found that the write policy does not impact the performance of most kernels expect for BFS. For BFS, the write-through with eviction policy for the L2 does better than a write back policy. Hence, our baseline uses a write evict policy for L2 write hits and write no-allocate for L2 write misses just like the L1 caches.

## *4.3 Locality in Graph Algorithms*

**Listing 2** BFS Skeleton Code

```
BFSStep(...)
begin
    ...
    for $i \leftarrow start, end$ do
        id = graphEdges[i]                              ▷ Load-A
        visited = visitedFlags[id]                      ▷ Load-B
        ...
    ...
end
```

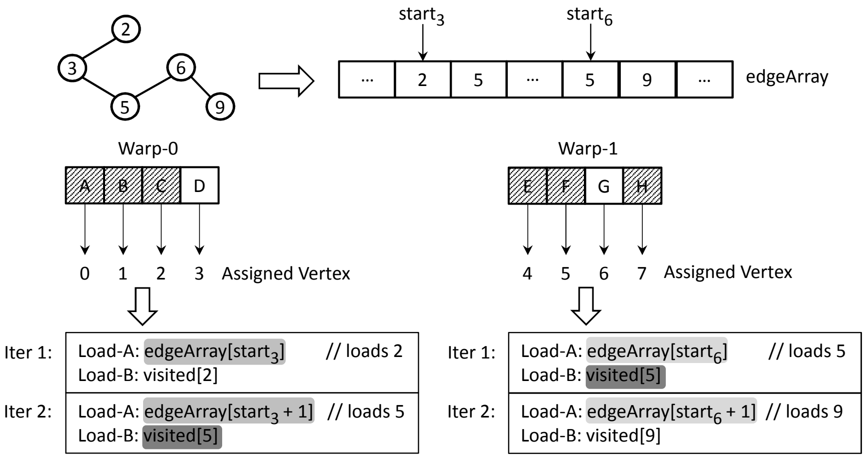


Figure 18: Intra and Inter warp locality in Graph Algorithms

Examining several graph algorithm implementations shows that some of the kernels in these algorithms have intra-thread and inter-thread locality; the inter-thread locality could be between threads that are running on the same core or on different cores. Listing 2 shows example code from a BFS implementation and Figure 18 shows the cache blocks accessed during a BFS step for an example graph. In Figure 18, a warp represents the granularity of SIMD execution on GPUs, typically a warp consists of 32 threads. Figure 18 also shows the contents of the edge array for the example graph. From Figure 18 we see that Load-A has intra-thread locality (Thread C and G access same cache blocks across iterations for Load-A), while Load-B is has some inter-thread locality (Thread C accesses visited[5] in its second iteration while Thread G accesses visited[5] in its first iteration). The locality of Load-B is dependent on the input graph and also on the scheduling of threads within and across cores and hence is difficult to predict. Depending on the algorithm there could be multiple loads such as Load-A and Load-B and there could be other loads without locality.

In the Fermi architecture, each SM can use either 48KB or 16KB of on-chip storage as data cache. For a 48KB data cache, if 1536 threads (the maximum possible number) are allocated, the cache space available per thread is less than one cache line (128B). Thus, though graph algorithms have locality, the cache space available is insufficient to exploit the locality fully.

## *4.4 Inclusion Property of Cache Hierarchy*

### 4.4.1 Introduction

As mentioned, current GPGPU architectures include a two-level write-through cache hierarchy. In the Fermi architecture [54], each of the 16 GPU cores has 64KB of on-chip storage that can be partitioned 48KB-16KB between the L1 data cache and Shared Memory (scratchpad memory) or vice-versa; a 768KB L2 cache is shared between all the cores. As described in Section 4.3 graph algorithms exhibit some locality, however, the sizes of GPU caches are extremely small to be very effective. A possible solution would be to operate the cache in an exclusive fashion in order to increase the effective cache size available and improve performance.

Though exclusive caches increase the effective cache space, their downside is that they can increase on-chip traffic considerably. Mechanisms such as Flexclusion [68], which dynamically change the cache mode between non-inclusive and exclusive depending on the benefit provided by exclusion have also been proposed. If it is determined that exclusion increases the number of cache hits, then Flexclusion operates the cache in exclusive mode, otherwise the cache is operated in non-inclusive mode since non-inclusion results in lower on-chip traffic. In addition to increasing on-chip traffic, in multi-core cache hierarchies, exclusion could also reduce the hits to data shared between cores. In case of exclusion, on a hit to a shared cache block in the L2, the block is invalidated in the L2 and is copied to the L1. If a second core were to now try and access the same cache block, the second core would have a miss. For cache blocks that are likely to be accessed by several cores before they are evicted, exclusion causes a reduction in the number of hits. However if a shared block is likely to be accessed by only a few cores before it is evicted, then exclusion is unlikely to reduce the number of hits. In addition, exclusion can increase the number of cache hits by providing additional cache space. Thus the performance improvement provided by exclusion is dependent on the number of sharers that each cache block has and the amount of extra space afforded by exclusion. Considering the on-chip traffic caused by exclusion and the likely reduction in number of hits to shared data, we also evaluate a region-based exclusion mechanism, R-EX, which decides to do exclusion or non-inclusion on a memory region basis.

For each cache block in the L2 we identify whether the cache block is private to a core or shared between cores. The attribute values for individual cache blocks is used to approximate the behavior of memory regions in the L2. This is then used to determine whether a region should be cached in non-inclusive mode or exclusive mode. Regions that are identified as private only with reuse are cached in exclusive mode, the rest of the regions cached in non-inclusive mode. By caching only private regions with reuse in exclusive mode, the amount of on-chip traffic is reduced while increasing the effectively available cache space. And, blocks from shared regions are inserted into L2 due to non-inclusive mode of operation and are likely to provide hits for other cores which access the same cache block.

We next briefly discuss exclusion and Flexclusion followed by the details of region-based exclusion and then the performance benefits of these mechanisms along with other analysis.

### 4.4.2 Exclusion and Flexclusion

Figure 19 shows the sequence of operations in the baseline non-inclusive cache hierarchy. On a L2 cache miss, a request for the missing block is forwarded to DRAM which responds with data. The cache block returned from DRAM is then inserted into both L1 and L2 caches. On eviction, cache blocks are silently dropped.

In an exclusive cache (Figure 20), the data returned from the DRAM is not inserted into the L2 cache, instead it is inserted only into the L1 cache. When a cache

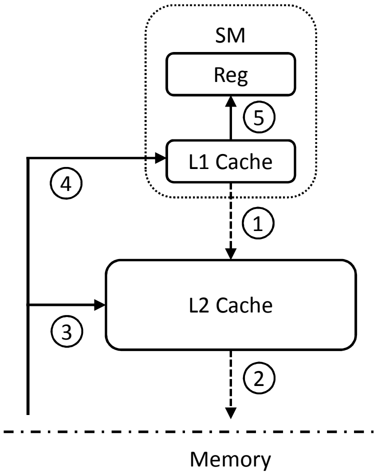


Figure 19: Flow of requests and data in non-inclusive cache hierarchy

block is evicted from L1 it is inserted into the L2. On a L2 hit, the L2 cache block is invalidated and the cache block inserted into the requested L1.

Flexclusion used set dueling monitors (SDMs) to decided between non-inclusion or exclusion. A SDM is a groups of sets that are dedicated to one specific policy. In case of Flexclusion, one SDM implements non-inclusion, while another implements exclusion. The non-SDM sets are called follower sets and these follow the policy which is currently doing better. A saturating counter tracks which of the two SDMs fewer misses and the follower sets follow the policy of the SDM with the fewer misses.

### 4.4.3 Region Based Exclusion

Below we discuss the motivation and the details for Region based exclusion.

#### *4.4.3.1 Motivation*

**Memory Allocation and Memory Access Patterns**

The CUDA programming model assumes that a GPU device has its own separate memory [56] and the kernel threads executing on the GPU access the GPU device memory for their data. The most common way of allocating memory on the GPU is via the cudaMalloc() function and its variants. Before a kernel is launched, one or

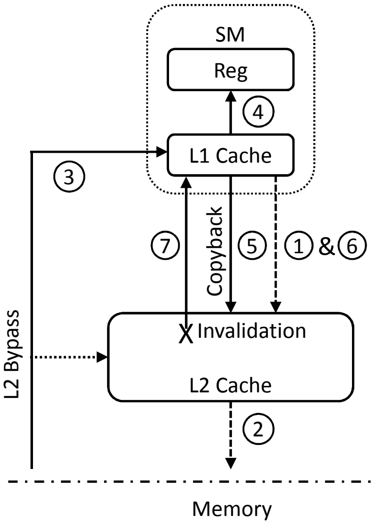


Figure 20: Flow of requests and data in exclusive cache hierarchy

more cudaMalloc*() calls are made for memory allocation. Each cudaMalloc*() call allocates a distinct region in the virtual address space of the application process and also allocates space in the device memory. After the necessary memory allocations, data is copied over into the GPU device memory from the CPU host memory via cudaMemcpy*() calls and the kernel(s) is (are) launched. When the kernel(s) has (have) completed operating on the data in the allocated memory, results are copied back to the CPU host using cudaMemcpy*() calls. Typically, CUDA applications perform only a few memory allocations. Table 9 shows the number of memory allocations performed by several CUDA applications.

As noted, the kernel threads operate on the memory regions allocated in the device memory. Each memory region could be partitioned or shared between the threads. That is, though all threads access the same memory region, each thread may access distinct addresses in the region or threads may access overlapped addresses. This can be extrapolated to say: for a given memory region, each core may access disticnt addresses or cores may access overlapped regions. Thus each memory region allocated in the device memory can have different sharing attributes. In Figure 18, graphEdges array has partitioned accesses and has reuse as well, while visitedFlags has shared accesses dependent on the graph structure. Such diversity in access patterns of regions is common in graph algorithms. Traversing the neighbors of a vertex by a thread has intra-thread reuse (and by extension, intra-core reuse) across iterations (Load-A), while accessing the attributes of the neighbors (Load-B) has inter-thread reuse (and by extension, inter-core reuse). Since cache blocks in shared regions are likely to be accessed by multiple cores, exclusion may not be suitable for such regions. However, exclusion would be suitable for regions with private accesses. In addition, since the number of regions is small unlike in the case of CPUs, a small table is likely to be sufficient for identifying region attributes.

#### *4.4.3.2 Mechanism*

The mechanism tracks and idenifies sharing attributes of different regions and decides which regions should be cached exclusively and which regions should be cache non-inclusively.

At the L2, the mechanism identifies the cache blocks have shared reuse or private reuse. Regions whose blocks have private reuse are cached exclusively. Cache blocks from such regions initially bypass the L2 and are inserted into the L1 only; on eviction from the L1 they are copied-back to the L2 cache. Such copied-back blocks are then invalidated on cache hit from the core which did the copyback.

#### *4.4.3.3 Identifying Regions*

As shown in Table 9, the number of regions allocated by many CUDA applications is often less than 32 and this information is readily available to the CUDA runtime. We assume that at the launch of a kernel the CUDA runtime populates the Region Table shown in Figure 22. In our evaluations we use a region table of size 31. Alternatively, the TLB entries can be extended to hold the region to which a virtual address belongs.

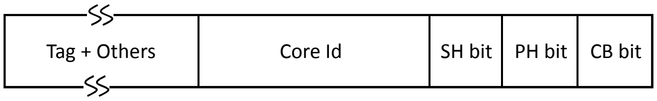


Figure 21: R-Exclusion: Extensions to L2 Tag

#### *4.4.3.4 Memory access patterns for Regions*

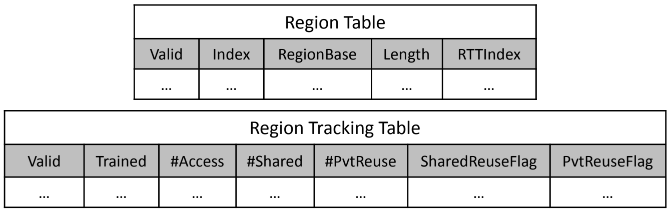
During kernel execution, the GPU threads access the allocated memory regions. In R-EX, instead of looking at accesses by each individual thread we examine the accesses made by the all threads in a core collectively. At the L2 cache level, we track accesses by individual cores rather than accesses by individual threads and identify the access patterns of cores. A cache block in the L2 has shared reuse if it is accessed by multiple cores; it has private reuse if it is accessed by the same core multiple time, otherwise, the cache block is private. A region has shared reuse if the fraction of cache blocks with shared reuse about a threshold, similarly a region has private reuse if the fraction of cache blocks with private reuse is above a threshold. We use a low cost mechanism that extends the information stored for each cache block in the L2 and uses counters in each GPU core for identifying how a region is accesses by the core.

#### *4.4.3.5 Detecting Shared-reuse and Private-reuse cache blocks*

**Listing 3** L2 cache operations

```
onCacheFill(cacheEntry, requestingCoreId)
begin
    cacheEntry.coreId = requestingCoreId
    cacheEntry.shBit = 0
    cacheEntry.phBit = 0
    cacheEntry.cbBit = 0
end

onCacheHit(cacheEntry, requestingCoreId)
begin
    if requestingCoreId != cacheEntry.coreId then
        cacheEntry.shBit = 1
end
```

To detect the shared and private attributes of a region the information stored in the tag storage for each cache block is extended as shown in Figure 21. The new fields are a core id and bits to indicate whether the cache block has shared reuse (SR bit), private reuse (PR bit) and whether the block has been copied back (CB bit) after eviction from L1. When a cache block is filled in the L2, the three bit fields are cleared and the core id field is set to the id of the core that issued the request for the block. Later on, when the cache block has a hit, the core id of the requesting core is compared with the core id stored in the tag array. If the ids are same, the PR bit is set, else the SR bit is set. When sending a response to a core, the status of the SR and PR bits are always included.



Region Table

| Valid | Index | RegionBase | Length | RTTIndex |
|---|---|---|---|---|
| … | … | … | … | … |

Region Tracking Table

| Valid | Trained | #Access | #Shared | #PvtReuse | SharedReuseFlag | PvtReuseFlag |
|---|---|---|---|---|---|---|
| … | … | … | … | … | … | … |

Figure 22: R-Exclusion: Region Tables

Each core has a Region Tracking Table (RTT) that tracks the shared attribute for upto N regions. The fields in a RTT entry are as shown in Figure 22; an RTT entry consists of a *Valid* flag, a *Trained* flag, a *SharedReuse* flag, a *PvtReuse* flag and *Access*, *Shared* and *PvtReuse* saturating counters. The counters are updated whenever there is a L1 cache fill. *Access* counts the number of accesses to the L2 for a region, while the *Shared* and *PvtReuse* counters are updated depending on whether the SR bit is set in the response. The *Access*, *SharedReuse* and *PvtReuse* counters and the *SharedReuse* flag and the *PvtReuse* flag are updated periodically. Listing 3 shows how the newly added fields to the L2 tag array are updated and Listing 4 shows the updates to the RTT on a L1 cache fill and the periodic updates as well. The L1-tags are also extended to include a copyback bit which is set for cache blocks from regions have private reuse, but no shared reuse. On eviction, such blocks are

**Listing 4** Region Table operations

```
onCacheFill(rgnId, shBit, phBit)
begin
    if shBit then
        entry.shd_hits += 1
    if phBit then
        entry.pvt_hits += 1
end

update()
begin
    for each entry in Region Table do
        if entry.acc != 0 then
            entry.acc = entry.acc / 2
            entry.shd_hits = entry.shd_hits / 2
            entry.pvt_hits = entry.pvt_hits / 2
            if entry.shd_hits > (entry.acc / 64) then
                entry.shd_reuse = true
            if entry.pvt_hits > (entry.acc / 64) then
                entry.pvt_reuse = true
end
```

copied back to the L2 and when inserted into the L2, the core id and CB bit for such blocks is set. On a future access by the core which did the copyback the cache block is invalidated.

With this mechanism a block that has shared reuse may not be identified as having shared re-use by the core that fetches the block from memory. However, subsequent cores that access the block, correctly identify the block as having shared reuse. If the first core that fetched a block acesses the block again after intervening accesses by other cores, then the core now correctly identifies the block as having shared as well as private reuse. Though the mechanism results in a core identifying some blocks with shared reuse as having no shared reuse, all blocks of a region with shared cache blocks are unlikely to be identified as private. This is because threads in a GPU kernel execute identical code without any synchronization. There is no strict order in which the cores access a shared memory region, thus requests from different cores to a shared region/cache block are likely to be interleaved. Thus this mechanism is successful in identifying memory regions with private as well as shared cache blocks.

#### *4.4.3.6 Managing large number of Regions*

To handle the situation in which the number of regions is greater the RTT size, a counter called *Untracked* counter is used to track the number of accesses to untracked regions. At the end of a period if the number of untracked accesses is greater than the number of accesses to the region with the fewest accesses, the entry for the region with the fewest accesses is deallocated and allocated to the first region that is not present in the RTT.

### 4.4.4 Methodology

Table 8: Inclusion Property: Simulated GPGPU Architecture

| | |
|---|---|
| Num. of cores | 16 |
| Front End | Fetch width: 2 warp-instruction/cycle;<br>4KB I-cache; stall on branch; 5 cycle decode |
| Execution core | 2 warp-instructions/cycle, Frequency: 1.2 GHz;<br>16-wide SIMD, in-order scheduling, latencies are modeled according to the CUDA manual [56];<br>128 KB register file |
| On-chip caches | 16/48 KB software managed cache, 16 loads/2-cycle;<br>4-cycle 6-way 48KB L1 cache, 16 loads/2-cycle<br>or 4-cycle 4-way 16KB L1 cache, 16 loads/2-cycle;<br>4-cycle 8 KB constant cache, 16 loads/2-cycle;<br>4-cycle 8 KB texture cache, 16 loads/2-cycle;<br>100-cycle round trip access to 768 KB L2 cache; |
| DRAM | 2 KB page, 16 banks with page interleaving,<br>128 reqs/bank, 8 channels, 100.8 GB/s,<br>400 cycle minimum round-trip time;<br>tCL=20, tRCD=28, tRP=12 (GDDR-5 [3]) |

We emulate several applications (Section 4.4.4.1) using GPUOcelot [23] and collect their traces which are simulated using MacSim [5], a cycle-level heterogeneous architecture simulator. The simulated architecture, shown in Table 8 is based on NVIDIA's Fermi [54].

Table 9: Inclusion Property: Evaluated kernels

| Kernel | Allocations |
|---|---|
| *Kernel* from<br>Breadth First Search [31](H-BFS) | 7 |
| *DijkstraKernel1* from<br>Single Source Shortest Path [31] (H-SSSP) | 7 |
| *Find_Min_Edge_Weight* from<br>Minimum Spanning Tree [76](H-MST) | 22 |
| *drelax* from<br>Breadth First Search [9](LS-BFS) | 10 |
| *drelax* from<br>Single Source Shortest Path [9](LS-SSSP) | 10 |
| *boruvkaone* from<br>Minimum Spanning Tree [9](LS-MST) | 25 |
| *kernel* from<br>Graph Coloring [17] (CLR) | 7 |
| *kernel* from<br>Maximal Independent Set [46] (MIS) | 7 |
| *kernel* from<br>Breadth First Search [21] (BFS-DD) | 8 |
| *kernel* from<br>Breadth First Search [34] (BFS-WC) | 7 |

Table 10: Inclusion Property: Input graph characteristics

| Input | #Vertices | #Edges | Avg. Degree | BFS Depth |
|---|---|---|---|---|
| Graph1M | 1000000 | 5999970 | 5.99 | 12 |
| RMAT20 | 1048576 | 8259994 | 7.88 | 13 |
| R4_2e20 | 1048576 | 4194304 | 4 | 16 |

#### *4.4.4.1 Benchmarks and inputs*

We evaluate kernels from several fundamental graph algorithm implementations for multiple inputs. Table 9 shows the evaluated kernels and the number of memory allocations performed by application. Note that several of these allocations are for termination flags and so on which have only a few memory accesses. H-BFS, H-SSSP and H-MST are taken from implementations provided by Harish, Vineet et al. [31, 76, 32] and LS-BFS, LS-SSSP and LS-MST are taken from LonestarGPU 0.9 [9]. We implement ST-CON [31], Graph Coloring [17, 13] and Maximal Independent Set [46, 13] algorithms based on prior work. We also evaluate two alternate implementations of BFS - a data-driven BFS implementation (BFS-DD) [21] and a warp-centric BFS implementation (BFS-WC) [34] - these implementations are discussed in Chapter 5.

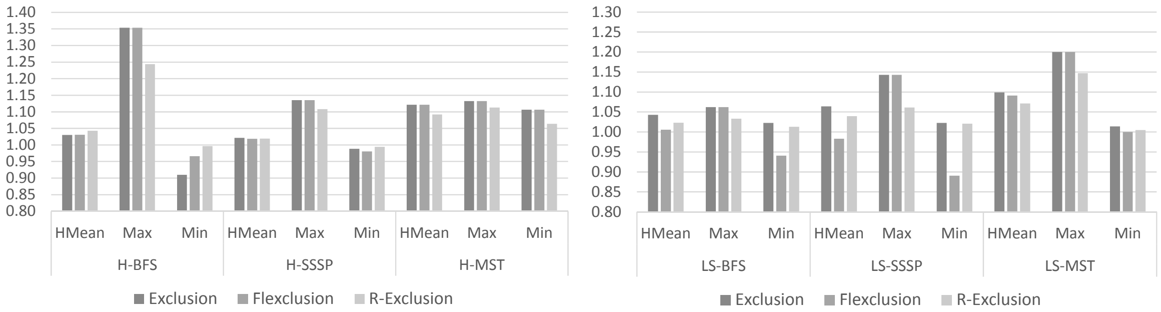
For each graph algorithm we generate traces for multiple input graphs and since graph algorithms are iterative, each kernel is invoked multiple times during a single application run. Some characteristics of the different input graphs are shown in Table 10. Graph1M is obtained from the Parboil benchmark suite [73] and is a random graph with a average vertex degree of 6, while RMAT20 and R4_2e20 are obtained from LonestarGPU 0.9 [9]. RMAT20 represents graphs found in the real world [11], it has few vertices with a large degree and a large number of vertices with a small degree; while R4_2e20 is a random graph with a average vertex degree of 4.

### 4.4.5 Results

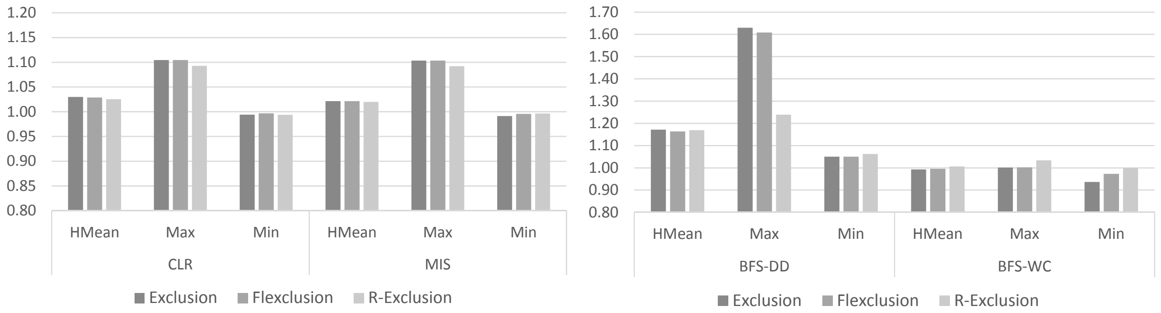
Table 11: Inclusion Property: Evaluated mechanisms

| Mechanism | Configuration |
|---|---|
| Baseline | Base cache hierarchy augmented with 1KB victim cache per L1 cache |
| Exclusion | Base cache hierarchy augmented with 1KB victim cache per L1 cache |
| Flexclusion | Base cache hierarchy augmented with 1KB victim cache per L1 cache<br>11-bit PSEL counter and 16 set SDMs |
| R-Exclusion | Base cache hierarchy with 32-entry RTT |

In this section we evaluate the mechanisms discussed in Section 4.4.2 and Section 4.4.3. The configurations for the different mechanisms are shown in Table 11. The baseline, exlusion and Flexclusion cache hierarchies are augmented with a victim cache to account for the hardware cost of R-Exclusion. For each algorithm, Instead of showing the results for individual kernel iterations with different inputs we show



(a) HIPC Kernels (b) Lonestar Kernels



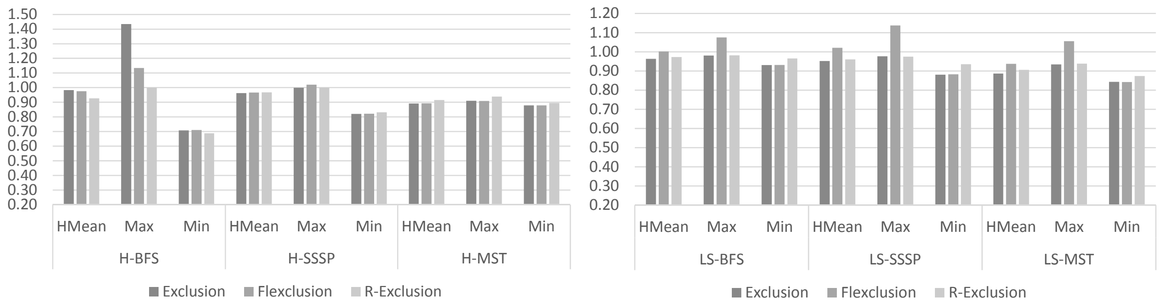
(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 23: Inclusion Property: IPC improvements relative to baseline

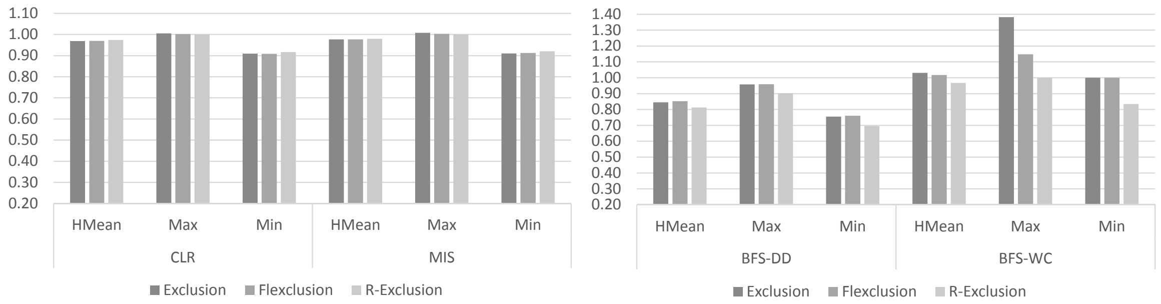
aggregated results as well as the maximum and the minimum of the metric being presented. For each algorithm, we find the mean (arithmetic mean or harmonic mean) across all kernel invocations with all inputs and then we find the mean (arithmetic mean or harmonic mean) across all the algorithms.

#### *4.4.5.1 Performance improvement with different mechanisms*

Figure 23 shows the performance of Exclusion, Flexclusion and R-Exclusion relative to the baseline. Besides showing the average relative performance across all kernel iterations for multiple inputs we also show the maximum and minimum relative performance numbers for the different mechanisms. Across all the algorithms, Exclusion, Flexclusion and R-Exclusion provide improvements of 6%, 4% and 5%. In



(a) HIPC Kernels (b) Lonestar Kernels



(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 24: Inclusion Property: Ratio of L2 MPKI relative to baseline

general, the average performance improvement is similar across the different mechanisms, however, in some cases Exclusion and Flexclusion cause performance degradation (the minimum numbers in these cases are smaller than 1).

**Analysis of performance improvements**

The evaluated mechanisms improve performance by increasing the number of L2 cache hits. Often, the mechanisms increase the number of hits to data accessed in a private fashion, while the number of hits to shared data may increase or decrease or remain about the same. In some cases, like in case of some invocations for H-BFS and BFS-WC, exclusion shows performance degradation due to reduction in the number of L2 hits to shared data without any increase in the number of hits to private data or the increase in hits to private data is not sufficient to offset the reduction in hits to shared data. Figure 24 shows the ratio of the L2 MPKI of the different mechanisms to the L2 MPKI of the baseline. On average exclusion and R-exclusion reduce the L2 MPKI by 6% while Flexclusion reduces the L2 MPKI by 4%. For some kernel invocations for H-BFS and BFS-WC, the increase in L2 MPKI is quite high for Flexclusion and exclusion due to reduction in the number of hits to shared data. In some cases, Flexclusion eliminates or reduces performance degradation due to exclusion. However, since the GPU L2 caches have only 768 sets in total and Flexclusion dedidates 16 sets to each SDM, sometimes Flexclusion cannot completely eliminate the performance reduction due to exclusion since about 2% of the sets are doing exclusion.

**H-BFS, H-SSSP and H-MST** For H-BFS, H-SSSP and H-MST all mechanisms show performance improvement on average. However, exclusion and Flexclusion show some degradation in some cases for H-BFS due to reduction in the number of L2 cache hits and R-Exclusion does not show degradation for any H-BFS kernel invocations. The maximum performance improvement due to R-Exclusion is lower than the maximum performance improvement due to exclusion or Flexclusion and the cases for which most benefit are seen are from H-BFS. For these kernels, Flexclusion often does non-inclusion when the number of active vertices are low and does exclusion when the number of active vertices are high. For H-SSSP, a large fraction of the accesses are atomic updates which are always inserted into the L2 cache and cannot be bypassed, thus, none of the mechanisms shows significant performance improvement for H-SSSP. For H-MST, exclusion and Flexcusion increase the number of L2 hits for private data while showing a slight increase to shared data as well.

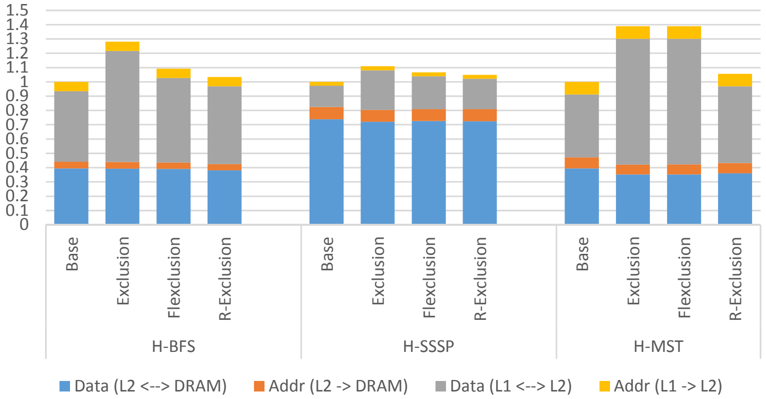
**LS-BFS, LS-SSSP and LS-MST** On average, exclusion and R-Exclusion show performance benefit for all of LS-BFS, LS-SSSP and LS-MST with no negative cases. On the other hand, Flexclusion shows performance degradation for LS-BFS and LS-SSSP for the rmat20 input. In these cases, Flexclusion does non-exclusion for a majority of the time while doing exclusion for some portion of the time.

**CLR and MIS** In case of CLR and MIS all the three mechanisms show similar benefit with there being several kernel instances for which no benefit is seen. Often, kernel instances corresponding to iterations towards the start of the algorithms show benefit and the benefit is due to the increase in the number of hits to private data.

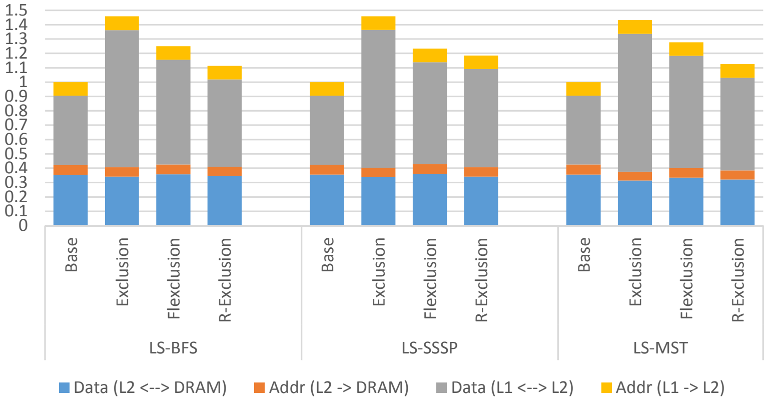
**BFS-DD and BFS-WC** For BFS-DD all mechanisms show benefit with exclusion and Flexclusion showing higher benefit in some cases and R-Exclusion showing higher benefit in other cases. The cases where R-Exclusion shows higher benefit is typically when R-Exclusion has higher number of hits for shared data than exclusion and Flexclusion. In case of BFS-WC, none of the mechanisms provide much benefit over the baseline, but we do see some degradation for exclusion and Flexclusion in some cases due to degradation in the number of hits to shared data.

#### *4.4.5.2 On-chip Traffic due to different mechanisms*

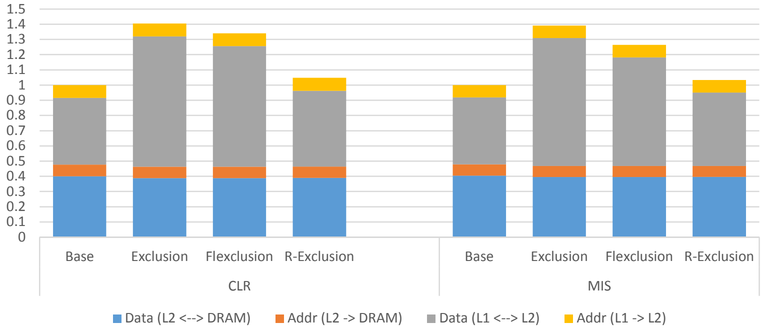
Figure 25 shows a breakdown of the average total on-chip traffic for the different mechanisms. The traffic is broken down into data traffic between L1 and L2 caches, address traffic from L1 to L2, data traffic between L2 caches and DRAM and address traffic fom L2 to DRAM. Each data packet is five flits long, while each address packet is considered to be one flit long. Inspite of the additional cache misses due to which it has more traffic between the L2 and the DRAM, the baseline has the lowest traffic for all evaluated algorithms and exclusion has the most. We see that though the traffic between the L2 caches and the DRAM is similar for the different mechanisms, the traffic between the L1 caches and the L2 caches varies significantly due to varying degrees of exclusive operation. Often R-Exclusion has lower traffic than Flexclusion which in turn has lower traffic than Exclusion. For H-MST and BFS-DD, exclusion and Flexclusion have similar traffic while Flexclusion and R-Exclusion have similar traffic for BFS-WC. The traffic due to Flexclusion depends on the mode in which Flexclusion operates. For H-MST and BFS-DD, Flexclusion operates in exclusive mode almost all the time for all kernel invocations, hence its traffic is similar to exclusion for these algorithms. For BFS-WC, Flexclusion operates in non-inclusive



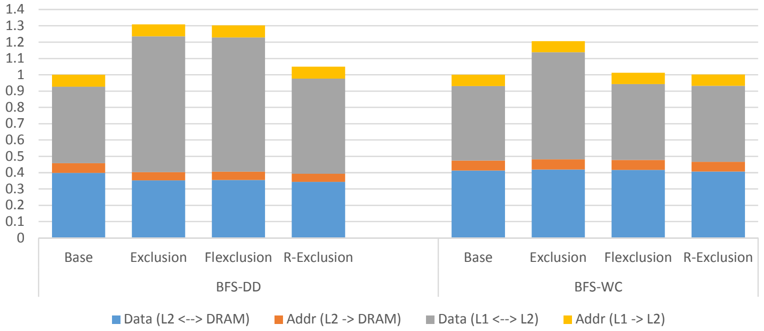
(a) HIPC Kernels



(b) Lonestar Kernels

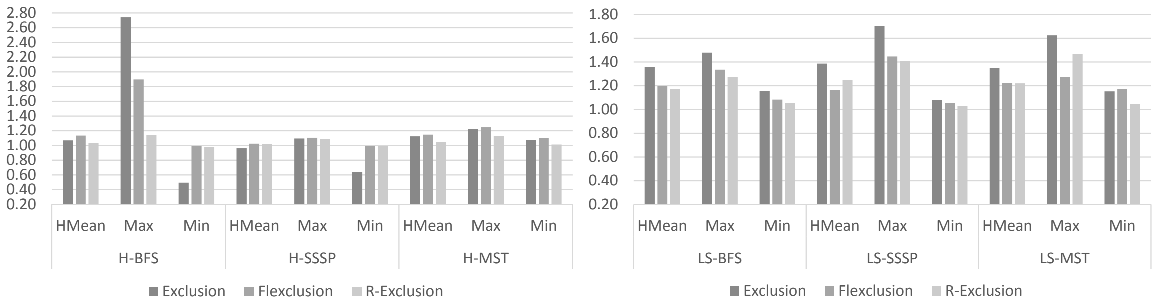


(c) CLR and MIS Kernels



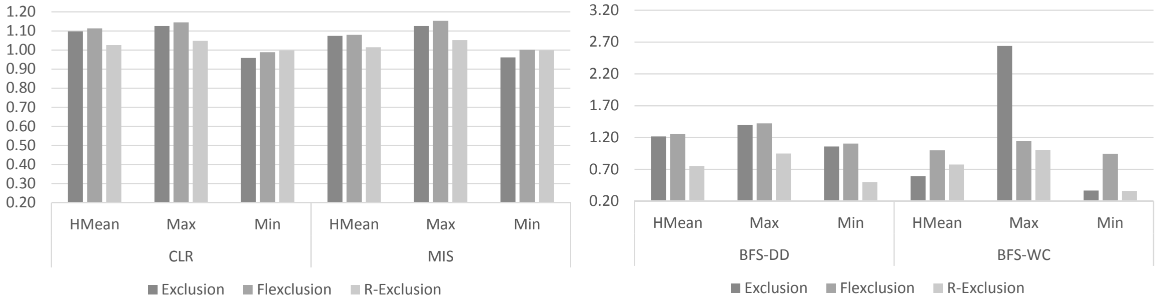
(d) Other BFS implementations

Figure 25: Inclusion Property: Traffic breakdown relative to baseline



(a) HIPC Kernels

(b) Lonestar Kernels



(c) CLR and MIS Kernels

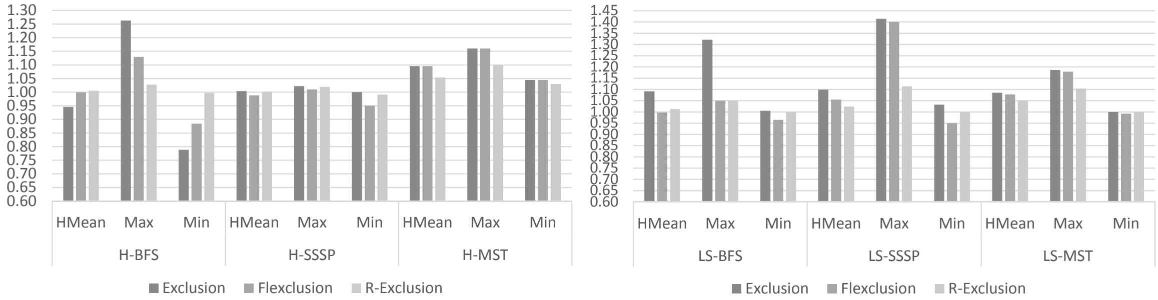
(d) Other BFS implementations

Figure 26: Ratio of L2 insertions for Exclusion, Flexclusion and R-Exclusion

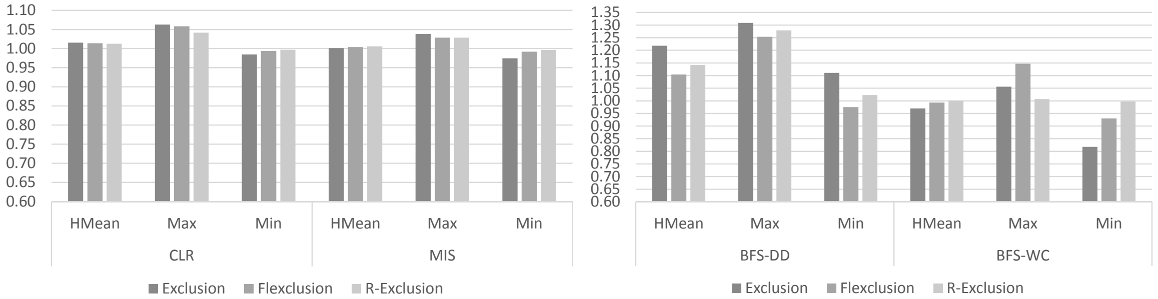
mode most of the time hence its traffic is similar to the baseline. For other algorithms, the operation of Flexclusion varies from kernel invocation to invocation. In many kernel invocations Flexclusion either does only non-inclusion or only exclusion. In some cases Flexclusion does both exclusion and non-inclusion for significant portions of the execution. In case of R-Exclusion, for many of the kernel invocations for BFS-WC, none of the regions are detected as private, thus R-Exclusion does non-inclusion for all regions and we see similar performance and traffic as the baseline. On average, exclusion, Flexclusion and R-Exclusion result in 34%, 22% and 7% additional traffic over the baseline.

**L2 insertions**

A metric related to the traffic is the number of L2 insertions. Figure 26 shows the average ratio of the number of L2 insertions for the different mechanisms relative to the baseline. On average, Exclusion does the most L2 insertions. There are some instances in which the evaluated mechanisms have fewer L2 insertions than



(a) HIPC Kernels (b) Lonestar Kernels



(c) CLR and MIS Kernels (d) Other BFS implementations

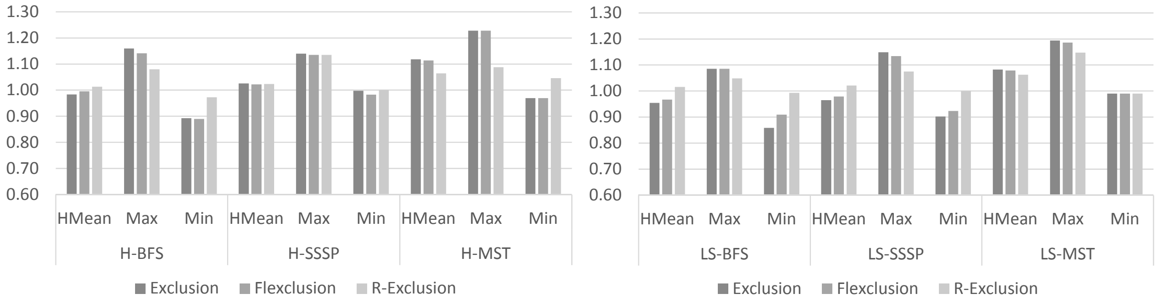
Figure 27: Inclusion Property: IPC improvements relative to baseline for a configuration with large caches

the baseline because of the baseline having additional cache misses. On average, exclusion, Flexclusion and R-Exclusion insert 6%, 13% and 0% more cache lines into L2 than the baseline.

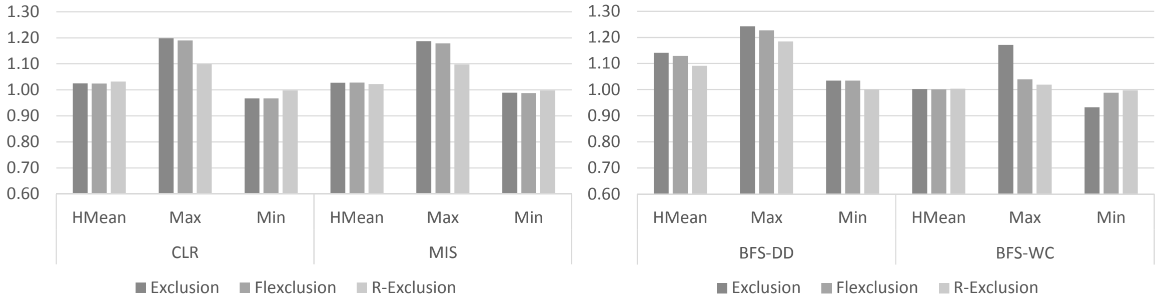
Figure 27 shows the performance of Exclusion, Flexclusion and R-Exclusion relative to the baseline with L1 caches of size 96KB and total L2 cache of size 1536MB. With the large caches, exclusion, Flexclusion and R-Exclusion provide performance benefits of 5%, 3% and 3%. While the average performance benefits reduces with large caches, the maximum performance benefit seen for exclusion improves for H-MST, LS-BFS and LS-SSSP; the negative cases for exclusion and Flexclusion for H-BFS and BFS-WC worsen.

#### *4.4.5.3 Configuration with increased bandwidth*

Figure 28 shows the performance of Exclusion, Flexclusion and R-Exclusion relative to non-inclusion for a machine configuration with a memory bandwidth of 144 GB/s.



(a) HIPC Kernels (b) Lonestar Kernels

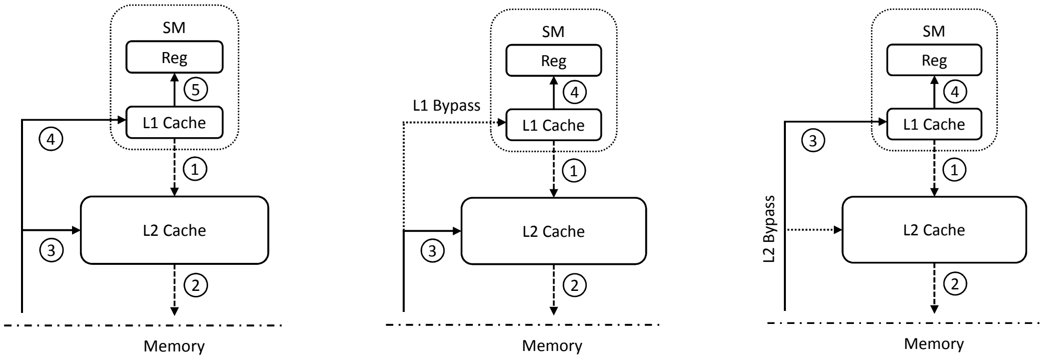


(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 28: Inclusion Property: IPC improvements relative to baseline for a configuration with higher bandwidth

Due to the increase in bandwidth, most kernel instances show an increase in IPC when compared to execution with the lower bandwidth configuration. On the higher bandwidth machine, all the evaluated mechanisms show an average performance improvement of 3%. On average, Exclusion shows peformance degradation for H-BFS, LS-BFS and LS-SSSP; while Flexclusion shows performance degradation for LS-BFS and LS-SSSP. For other kernels, Exclusion and Flexclusion show performance degradation for some kernel instances. Even, R-Exclusion shows performance degradation for some kernel instances, but typically, if there is degradation, the degradation for R-Exclusion is less than the degradation for Exclusion and Flexclusion. However, often when there is performance improvement, the improvement due to R-Exclusion is lower than the improvement due to Exclusion and Flexclusion. Due to the increase in memory bandwidth, the average memory access latency is generally reduced for a non-inclusive cache hierarchy. In case of exclusion, Flexclusion and R-Exclusion, the increased on-chip traffic increases the memory access latency and the reduction in memory access latency due to increased cache hits is sometimes not sufficient to overcome the increase in delay due to increased traffic. Hence we see several instances of degradation, especially for Exclusion and Flexclusion.

## *4.5 Cache Bypass*



(a) Baseline (a) L1 Bypass only (b) L2 Bypass only

Figure 29: Flow of requests and data through the cache hierarchy for several bypass conditions

As seen in Section 4.3, graph algorithms can likely contain both intra-thread and inter-thread locality and in addition, there could be accesses without any locality. And often, the cache sizes are too small to exploit the existing locality. Thus cache bypassing can be a viable technique for improving caching efficiency for graph algorithms. Typically, for CPU architectures, cache bypass is studied at the last level cache only. However, in case of GPUs bypassing can be beneficial at L1 caches as well due to the low hit rates of the L1 caches. Figure 29 shows the flows of cache blocks through the GPU hierarchy under different bypass conditions.

In this section we evaluate several bypass techniques that were previously proposed for the CPU, for the L1 and L2 caches in GPUs. Prior bypass techniques could be classified as based on memory addresses, PC or a combination of the two. For our evaluation we use a memory region/macroblock based mechanism that uses a Memory Address Table (MAT) [40], an address based probabilistic mechanism called Adaptive

Bypass [27] and a PC-based mechanism called Optimal Bypass Monitor [45]. We also evaluate a mechanism called B+C that uses the Region-based Exclusion (R-EX) mechanism discussed in Section 4.4 previously. We briefly discuss the evaluated techniques and then proceed with our evaluation.

### 4.5.1 Memory Address Table with macroblocks

The Memory Address Table [40] (MAT) based mechanism divides memory into macroblocks or regions. The MAT has a saturating counter, with the counter value representing the frequency of accesses to the corresponding macroblock. On a memory access, the MAT entry for the corresponding macroblock is looked up along with the cache. In case of a cache hit the counter value is ignored, in case of a miss, the MAT counter value for the cache block selected for replacement is also looked up. If the counter of the missing block is below a certain fraction of the counter value of the replacement victim, then the missing block bypasses the cache. This mechanism favors loads that are accessed more frequently, not necessarily loads that are likely to provide cache hits.

### 4.5.2 Adaptive Bypass

Adaptive bypass learns the bypass probability for each cache set individually and bypasses incoming cache blocks accordingly; the mechanism is agnostic of all attributes of PC accessing the cache blocks and the memory area to which the cache block belongs and the bypass decisions are made for each set considering only the stream of addresses mapping to the set. Each set is equipped with bypass tracking logic which determines the impact of a bypass decision on cache performance. On a cache miss, the incoming cache block (IB) can be either inserted into the cache replacing the victim cache block (VB) or the incoming cache block can bypass the cache leaving the victim cache block in the cache. Adaptive bypass determines which block between the IB and VB is likely to have a hit and updates the bypass probability accordingly.

### 4.5.3 Optimal Bypass Monitor

Optimal Bypass Monitor tries to emulate the decisions of the optimal bypass. On a cache miss OBM determines whether the incoming block should replace the victim block or should it bypass the cache. To do this OBM uses a Replacement History Table (RHT) to keep track of incoming block (IB) and victim block (VB) pairs on cache misses. IB-VB pairs are probabilistically entered into the RHT. Then depending on the reuse behavior of the incoming and victim blocks the OBM takes future bypass decisions. If between the IB and the VB, the IB was accessed next, then OBM should replace VB; if VB was accessed next then IB should be bypassed. The IB should be bypassed even if neither IB nor VB was accessed before the IB is evicted. Using the RHT, OBM learns bypass decisions for IB-VB pairs and updates Bypass Decision Counters (BDCT) that are maintained for each PC. When the bypass decision for a IB-VB pair is learnt, the BDCT counters for the PCs corresponding to the IB and VB are updated and on a future cache miss, the BDCT consters are consulted for making bypass decisions.

### 4.5.4 Bypass+Copyback

The region based exclusion mechanism discussed in Section 4.4 can detect whether memory regions have shared reuse and private reuse at the L2 cache. A region which has neither shared reuse nor private reuse can bypass the L2 cache and regions that have private reuse only can operate in exclusive mode. The bypass+copyback (B+C) mechanism uses the reuse information available from the region attribute detection mechanism for making caching and bypass decisions at the L2. In addition to detecting region properties at the L2, the mechanism also detects the properties of individual PCs at the L1 cache and uses this information for bypass decisions at the L1 caches.

At the L2 level, the mechanism identifies whether the cache blocks of a region have shared reuse, private reuse, neither or both. Regions whose blocks do not have shared reuse always bypass the L2 cache. Regions whose blocks do not have shared reuse but have private reuse in the L2, bypass the L2 cache, but are copied back into the L2 cache on eviction from the L1 cache. Such copied-back blocks are then invalidated on cache hit from the core which copied-back them. This ensures that cache blocks that don't have shared reuse but have have private reuse spend as little time as possible in the L2 cache. They bypass the cache initially and on eviction from the L1 they are copied-back to the L2 because they are likely to be accessed again. And on being accessed again by the core which copied back the cache block, the cache lines holding these blocks are invalidated.

At the L1 level, the mechanism identifies PCs which fetch cache blocks that provide hits instead of tracking PCs that have cache hits. Since a PC that has a hit for a cache block may not necesssarily be the PC that fetched the cache block, identifying such PCs that fetch cache blocks with hits is better for making bypass decisions. The mechanism bypasses cache blocks fetched by PCs that do not provide any hits.

#### *4.5.4.1 Identifying Regions and their attributes*

The identification of regions and their attributes is the same as in Section 4.4.3.2.

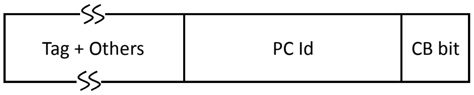
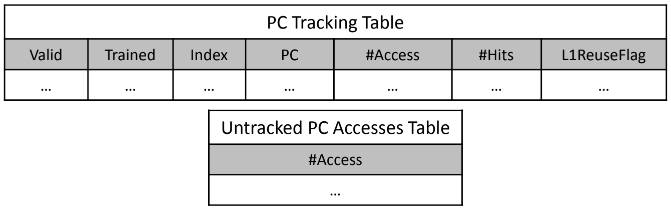


Figure 30: Bypass+Copyback: Extensions to L1 Tag



PC Tracking Table

| Valid | Trained | Index | PC | #Access | #Hits | L1ReuseFlag |
|---|---|---|---|---|---|---|
| … | … | … | … | … | … | … |

Untracked PC Accesses Table

| #Access |
|---|
| … |

Figure 31: Bypass+Copyback: PC Tables

#### *4.5.4.2 Detecting PC Properties*

To detect the reuse properties of a PC, the L1 cache tags are extended as shown in Figure 30 to included a *PC id* field This field identifies the PC that fetched the cache block into the L1 cache. Note that the region based exclusion mechanism requires the addition of a *CB* bit to the L1 tag. The *CB* bit indicates whether the cache block should be copied back into the L2 cache after eviction from the L1 cache. Each core includes a PC Tracking Table (PTT) for tracking the reuse attributes of a PC. Each entry in the PTT consists of: *Valid* and *Trained* flags, *Index* field, *Access* and *Hit* saturating counters, and the *L1Reuse* flags. The Index field gives the index of the entry within the PTT and the index values start from zero. The PC id field in the exended L1 tag contains the index of the entry in the PTT for the PC that fetched the cache block. Since the PTT is of fixed size, in kernels with a large number of load PCs, some load PCs may not be allocated entries in the PTT. Such PCs that have not been allocated entries in the PTT use the size of the PTT which is an invalid index values as their index value.

**Listing 5** L1 cache operations

```
onCacheFill(cacheEntry, pcId, cbBit)
begin
    cacheEntry.pcId = requestingCoreId
    cacheEntry.cbBit = cbBit
end
```

During the coalescing of memory requests, the PC id of the load PC is looked up. For each request issued to the cache, the Access counter of the corresponding PC is updated. In the event of a primary cache miss, the PC id is saved as part of the MSHR entry tracking the miss. When the cache block is returned and is being filled, the PC id saved in the MSHR entry is used for updating the PC id in the cache block tag (Listing 5). Later on, in the event of a cache hit, the hit counter corresponding to the saved PC index in the tag field is updated. This way the number of accesses

**Listing 6** PC Table operations

```
onCacheHit(pc, pcId)
begin
    pcTableEntry{pc}.acc += 1        ▷ update counter for entry with matching PC
    pcTableEntry[pcId].hit += 1          ▷ update counter for entry at index pcId
end

onCacheMiss(pc)
begin
    pcTableEntry{pc}.acc += 1        ▷ update counter for entry with matching PC
end

update(())
begin
    for all entry in PC Table do
        if entry.acc != 0 then
            entry.acc = entry.acc / 2
            entry.hits = entry.hits / 2
            if entry.hits > (entry.acc / 64) then
                entry.reuse = true
end
```

made by a PC and the number of cache hits provided by the cache blocks fetched by a PC is tracked. On eviction from the L1 cache, if a cache block has the CB bit set, then the cache block is copied back to the L2 cache. Periodically, the Access and Hit counters and the L1Reuse flags are updated as shown in Listing 6.

#### *4.5.4.3 Making Bypass and Copyback Decisions*

Before the L1 cache is accessed, the PTT and RTT are looked up to obtain the values of the L1Reuse bit for the PC and the ShareReuse and PvtReuse bit for the region accessed by the loads. Using the values of these 3-bits, bypass and copyback decisions are made as shown in Table 12. The decisions are then communicated to the cache hierarchy along with the cache access requests. The case where the L1Reuse bit is unknown occurs when a PC does not have an allocated entry in the PTT due to unavailable entries. Similarly, the SharedReuse and the PvtReuse bits are unknown when a region does not have an allocated entry in the RTT due to unavailable entries.

Table 12: Bypass: Bypass and Copyback Decisions of B+C mechanism

| L1 Reuse | L2 Shared Reuse | L2 Pvt Reuse | Action |
|---|---|---|---|
| No | No | No | bypass L2, bypass L1 |
| No | No | Yes | bypass L2, copyback to L2, invalidate on hit L2 |
| No | Yes | No | bypass L1 |
| No | Yes | Yes | bypass L1 |
| Yes | No | No | bypass L2 |
| No | No | No | bypass L2, bypass L1 |
| Yes | No | Yes | bypass L2, copyback to L2, invalidate on L2 hit |
| Yes | Yes | No | - |
| Yes | Yes | Yes | - |
| No | Unknown | Unknown | bypass L1 |
| Unknown | No | Yes | bypass L2, copyback to L2, invalidate on hit L2 |

### 4.5.5 Managing large number of PCs

To handle the situations in which the number of load PCs is greater than the PTT size, a set of counters organized into a table called the Untracked PC Access Table(Figure 31) is used. We wish to use the PTT for PCs that perform the most accesses and so the Untracked PC Access Table (UPAT) is used to track accesses by PCs for which we are unable to allocate entries. PCs which do not have allocated entries are hashed to one of the entries in UPAT and increment the corresponding counter for each access. At the end of a period, if the number of accesses to the PC with the fewest access in the PTT is smaller than the highest counter in the UPAT, then the PC with the fewest accesses in the PTT is replaced with the first PC which maps to the counter with the highest value in the UPAT. This way, PTT can track the PCs that make the most accesses.

### 4.5.6 Total Hardware Cost

The cost provided here includes the cost for the region attribute tracking mechanism as well. The tag extensions to the L2 cache blocks require 5376B in total; while the tag extensions to the L1 cache blocks require 288B per L1 core. The other core extensions such as the Region Table, the Region Tracking Table and the PC Tracking Table require 1KB of storage in total.

### 4.5.7 Methodology

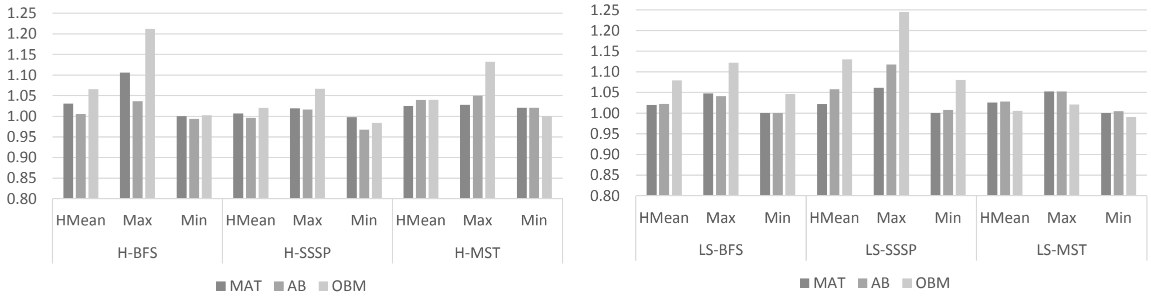
Table 13: Bypass: Evaluated bypass mechanisms

| Mechanism | Configuration |
|---|---|
| Memory Access Table (MAT) [40] | 1024 entries, 1KB Macroblocks, f = 0.99 |
| Adaptive Bypass (AB) [27] | Bypass probability = 1/64,<br>Virtual bypass probability = 1/16,<br>Min bypass probability = 1/256 |
| Optimal Bypass Monitor [45] | 64-entry RHT, RHT insert probability = 1/8<br>32-entry BDCT, 5-bit BDCT counters,<br>64-entry RHT |
| Bypass+Copyback | 31-entry RTT, 15-entry PTT |

The methodology is the same as described in Section 4.4.4. The evaluated bypass mechanisms and their configurations are as shown in Table 13.

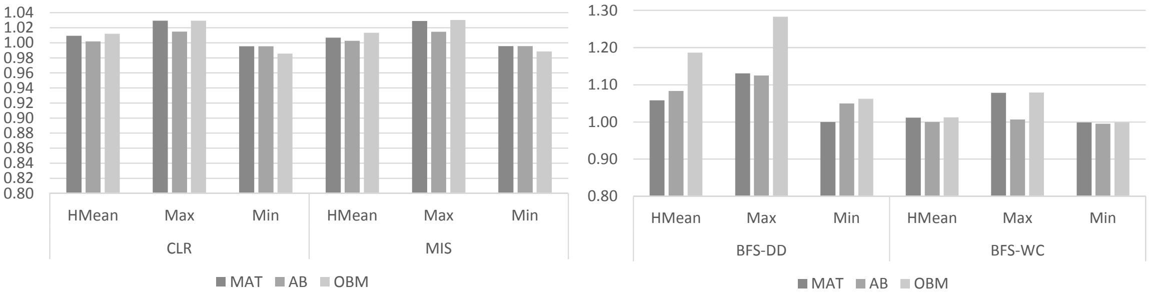
### 4.5.8 Results

Figure 32 shows the IPC of the evaluated mechanisms relative to the baseline when used for L2 bypass only. On average, MAT, AB and OBM provide a benefit of 2%, 2% and 5% across the evaluated kernels. MAT shows above 5% improvement on average for BFS-DD only, while AB shows above 5% improvement for and OBM shows above 5% improvement for LS-BFS and BFS-DD; for OBM it is H-BFS, LS-BFS, LS-SSSP. For MAT and OBM, in case of H-BFS, BFS-DD and BFS-WC the performance improvement is due to increase in the number of hits to shared data due to more frequent bypassing of private data. For rest of the kernels, OBM shows performance improvement due to increase in the number of hits to private data. Similarly, the performance improvement due to AB is because of increase in the number of hits to



(a) HIPC Kernels

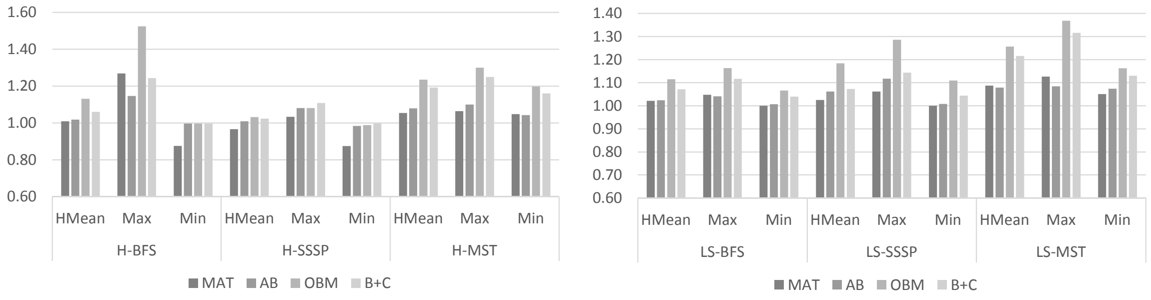
(b) Lonestar Kernels



(c) CLR and MIS Kernels

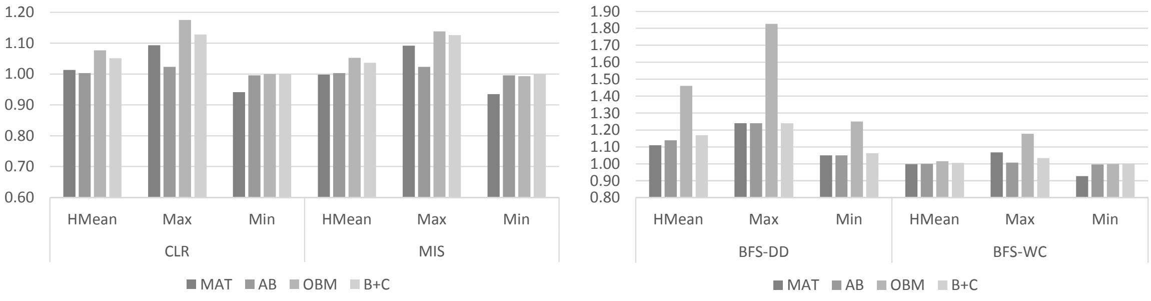
(d) Other BFS implementations

Figure 32: L2 Bypass Only: IPC improvement over baseline (MAT - Memory Access Table, AB - Adaptive Bypass, OBM - Optimal Bypass Monitor)



(a) HIPC Kernels

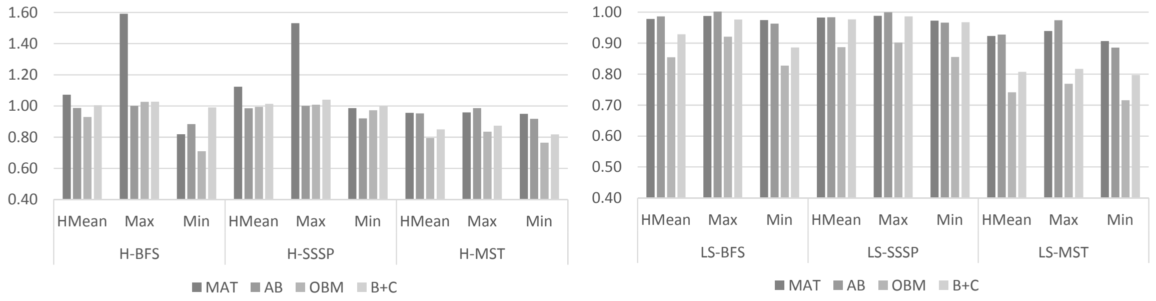
(b) Lonestar Kernels



(c) CLR and MIS Kernels

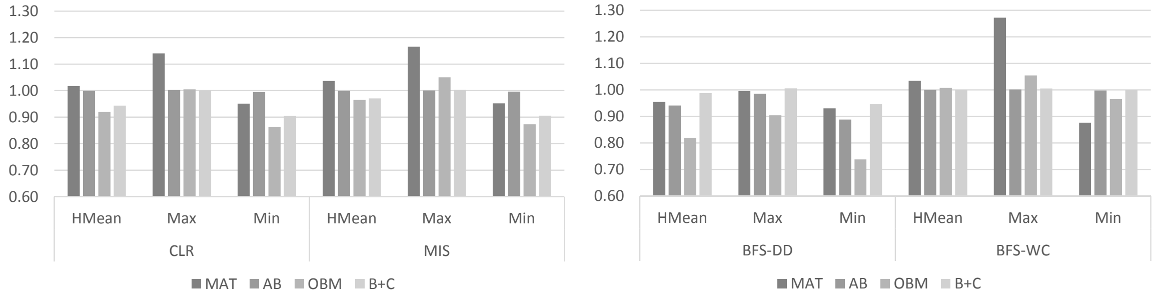
(d) Other BFS implementations

Figure 33: L1 and L2 Bypass: IPC improvement over baseline



(a) HIPC Kernels

(b) Lonestar Kernels



(c) CLR and MIS Kernels

(d) Other BFS implementations

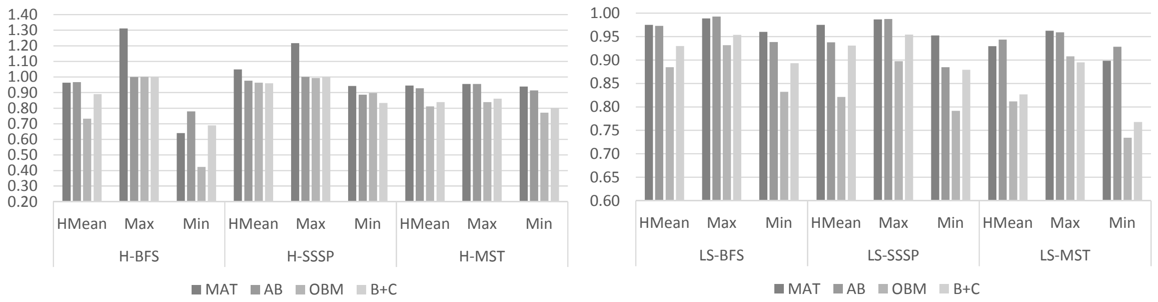
Figure 34: L1 and L2 Bypass: L1 MPKI relative to baseline

private data. B+C with L2 bypass, performs very similarly to R-Exclusion; there are individual kernel instances which show better performance with B+C than with R-Exclusion, but on average both perform the same.

As mentioned earlier, since graph algorithms have locality with low L1 hit rates, cache bypassing can be done at the L1 caches as well. Figure 33 shows the IPC of the evaluated mechanisms relative to the baseline when used for bypassing at both L1 and L2 caches. The average performance improvement for MAT, AB, OBM and B+C are 3%, 4%, 14% and 9%. Applying bypass mechanisms at both L1 and L2 caches improves the performance benefits seen from all the mechanisms. Now, almost all kernels show above 5% performance improvement for at least one of the evaluated mechanisms. The results presented henceforth are for bypassing at both L1 and L2 caches.

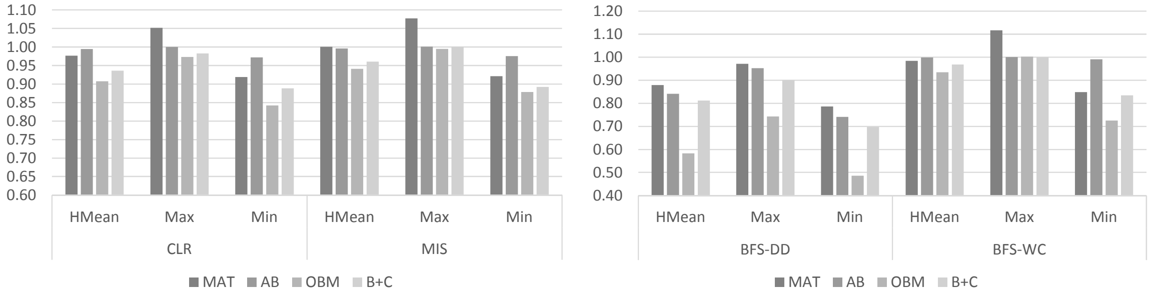
**Analysis of performance improvements**

The mchanisms improve performance by reducing the L1 and L2 miss rates by



(a) HIPC Kernels

(b) Lonestar Kernels



(c) CLR and MIS Kernels

(d) Other BFS implementations

Figure 35: L1 and L2 Bypass: L2 MPKI relative to baseline

bypassing lines that are unlikely to have reuse. Figure 34 and Figure 35 show the L1 and L2 MPKI for the evaluated mechanisms relative to the baseline. For MAT there is an increase in L1 and/or L2 MPKI for some invocations of H-BFS, H-SSSP, CLR, MIS and BFS-WC resulting in performance degradation in some cases. MAT with L2 cache bypass only does not show degradation for any of the kernel invocations. However, with the introduction of L1 cache bypassing, MAT shows significant performance degradation in some cases. The frequency based approach of MAT does not seem to work well for bypassing for the L1 caches and in many cases loads from PCs that provide hits are bypassed resulting in perofrmance degradation. For the other mechanisms there is typically a reduction in the L1 and L2 MPKI with the addition of L1 cache bypassing.

AB learns the cache bypass probablity of each set individually; in case bypasses are effective, the probability of bypass is increased, in case bypasses are ineffective, the probability of bypass is reduced. In case bypasses are neither effective nor ineffective, the bypass probability remains unchanged. Experimental results show that for Adaptive Bypass the performance improvement improves with the addition of L1 cache bypass.

OBM is able to identify PCs that provide hits and those that do not. In addition it can also adapt to the changes in the reuse behavior of individual PCs very quickly. Like AB, OBM tries to learn whether a bypass was effective or not. In case of AB, when the cache block corresponding to the incoming tag is evicted, it means that the effectiveness of bypass could not be determined and the bypass probabilites are unaffected. In a similar situation, OBM concludes that reuse distances of both the incoming block and the victim block are greater than cache size and increases the counter corresponding to the PC of the incoming block. Thus OBM is likely to bypass more cache lines than others.

At the L2 caches, B+C identifies regions that have no reuse and cache blocks from such regions bypass the L2. While at the L1 caches, B+C identifies PCs that fetch cache lines without reuse and bypasses caches fetched by those PCs. In addition B+C uses exclusion for memory regions that have been identified as private. Unlike MAT, AB and OBM, B+C only bypasses from regions that have been identified as having no reuse, while the other mechanisms can bypass lines even from regions that have reuse.

**H-BFS, H-SSSP, H-MST** For the HIPC kernels, except for MAT, all mechanisms show either performance improvement or no degradation for all kernel invocations. For MAT, as mentioned above in some cases the number of L1 hits is reduced - due to frequency based bypassing MAT is more likely to bypass cache blocks from PCs/regions that have higher hit ratio but fewer access than cache blocks that are cached - and thus performance is reduced.

**LS-BFS, LS-SSSP, LS-MST** For the Lonestar kernels, none of the mechanisms show any degradation, with OBM and B+C showing the best performance improvement. MAT shows performance improvement for LS-MST from the L2 bypass only case due to effective bypassing at the L1 cache. In this case neighbor attribute load is bypassed more often than the neighbor load even though it is more often to improve performance. AB, OBM and B+C improve performance by increasing hits to data with intra-thread locality and in some cases by increasing hits to data with inter-thread locality as well.

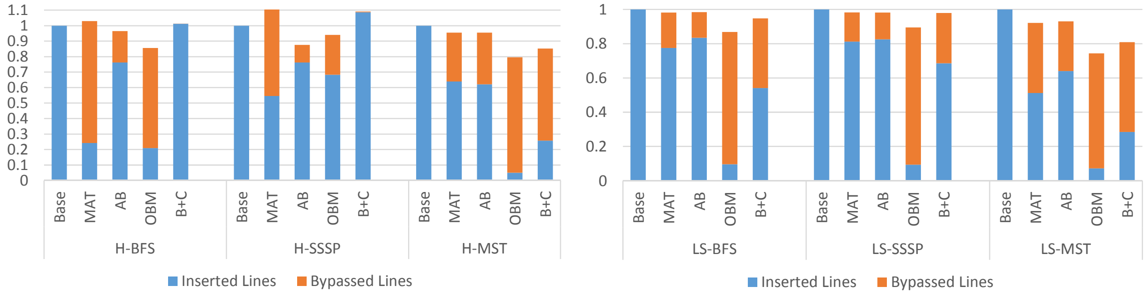
**CLR, MIS** In case of CLR and MIS all mechanisms except MAT either show performance improvement or no performance degradation. For MAT, same as in the case of HIPC kernels, even for the CLR and MIS kernel cache blocks that are more likely to provide hits are bypassed in favor of cache blocks from regions that have more frequent accesses.

**Other BFS implementations** For BFS-DD all mechanisms benefit from L1 bypassing while in case of BFS-WC, MAT shows degradation in some cases due to bypassing loads with intra-thread locality which have higher hit ratio but less frequent.

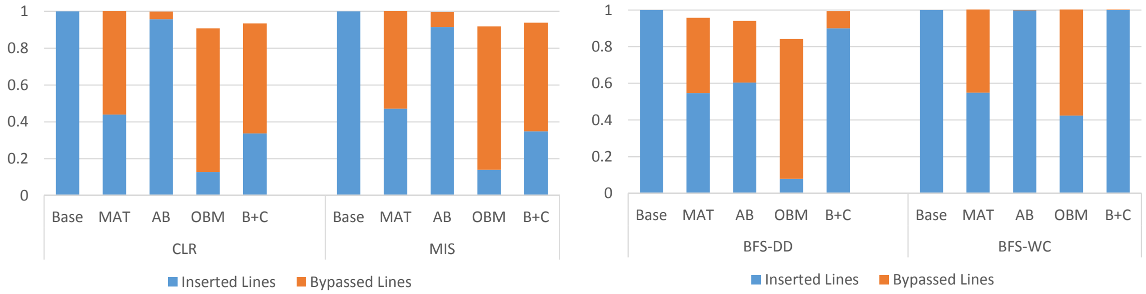
#### *4.5.8.1 Fraction of bypassed lines*

Figure 37 show the number of bypassed and inserted lines at the L2 level for the different mechanisms as a fraction of the number of lines fetched by the baseline; while Figure 36 shows the same metric for the L1 cache. In case of the baseline, cache lines are always inserted into the L2 cache, hence the columns for the baseline always show 100% of the lines as inserted. At the L1 level, MAT sometimes fetches more cache blocks than the baseline for H-SSSP and BFS-WC due to ineffective bypassing which results in fetching more cache blocks from memory into the L2 as well.

AB tends to bypass fewer lines than MAT and OBM. At the L1 cache level, AB detects that bypasses are often ineffective rather than effective and thus bypasses

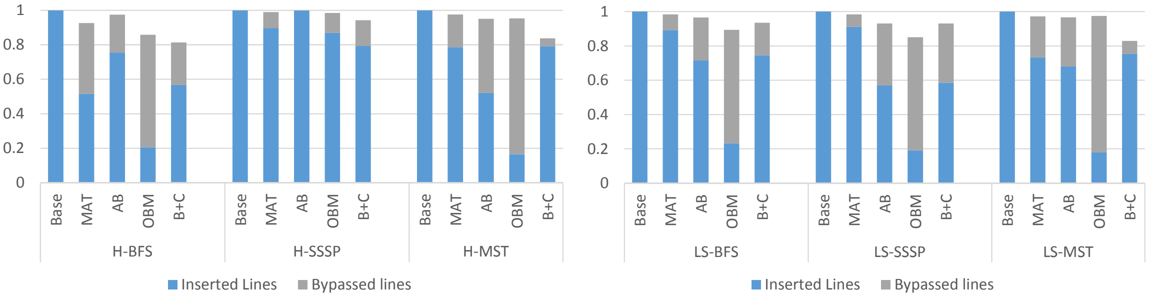


(a) HIPC Kernels (b) Lonestar Kernels

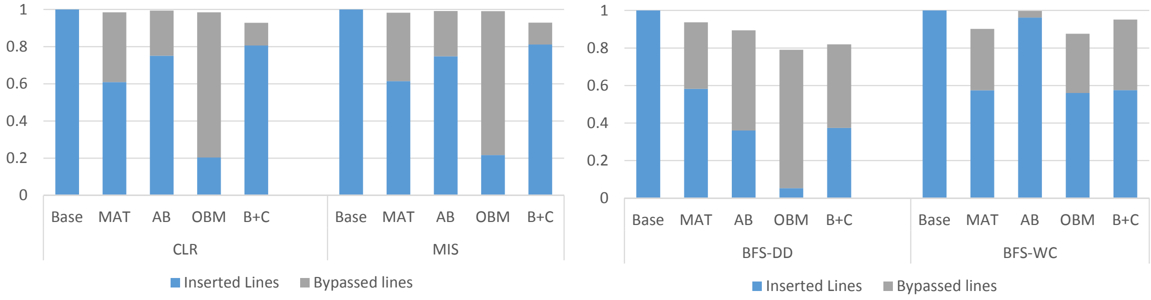


(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 36: L1 and L2 Bypass: Number of inserted and bypassed cache lines at the L1 cache relative to the number of lines inserted into the L1 cache for the baseline



(a) HIPC Kernels (b) Lonestar Kernels



(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 37: L1 and L2 Bypass: Number of inserted and bypassed cache lines at the L2 cache, relative to the number of lines inserted into the L2 for the baseline

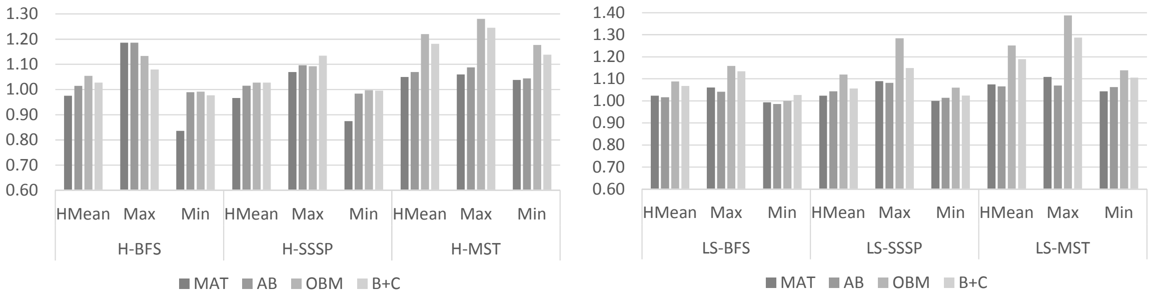
fewer lines. However, for a majority of the time AB is unable to determine whether the bypass is effective or not, since neither the incoming line or the victim line have a hit before the incoming line is evicted and tracking terminates. Similarly at the L2 level, AB detects that bypasses are more effective than ineffective for H-MST, LS-BFS, LS-SSSP LS-MST, CLR, MIS and BFS-DD and thus bypasses a bigger fraction of lines for these kernels than others.

At the L1 cache level, OBM bypasses the most cache lines for most benchmarks and at the L2 level too it bypasses a significant fraction of the lines fetched from DRAM. OBM not only bypasses lines from PCs that do not have any reuse it also bypasses lines from PCs that have reuse as well depending on their recent behavior. Since the L2 has six slices and a monitor is used for each slice, OBM has the benefit of being able to adapt to the behavior in each individual slice.

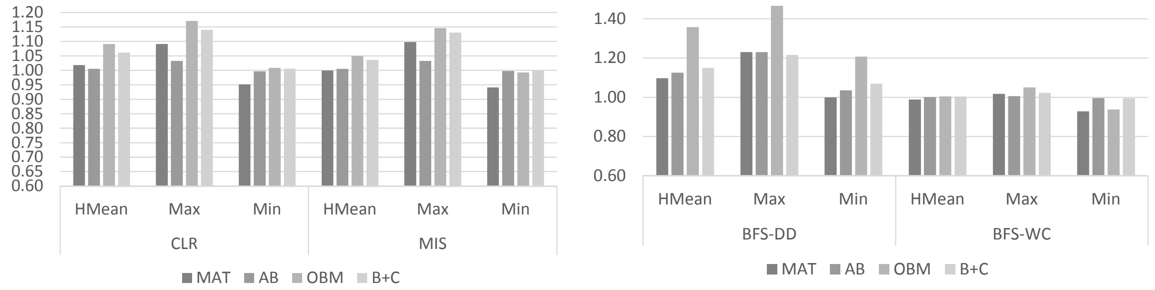
B+C does not bypass insertion of many/any cache lines into the L1 for H-BFS, H-SSSP, BFS-DD and BFS-WC. All PCs in these benchmarks tend to have some locality in the L1 cache. While for other benchmarks B+C bypasses the neighbor attribute arrays at the L1 caches depending on whether locality is detected and in case of LS-SSSP and LS-BFS with r42e20 input cases bypasses cache blocks from the neighbor array also.

#### *4.5.8.2 Configuration with increased bandwidth*

Figure 38 shows the performance improvement for MAT, AB, OBM and B+C when used for bypassing at both L1 and L2 relative to no-bypass for a machine configuration with a memory bandwidth of 144 GB/s. On average, MAT, AB, OBM and B+C show performance improvements of 2%, 3%, 12% and 8%. On the machine configuration with higher bandwidth, the different mechanisms show similar performance trends compared to the machine configuration evaluated earlier with a slight reduction in performance for some kernel instances.



(a) HIPC Kernels (b) Lonestar Kernels

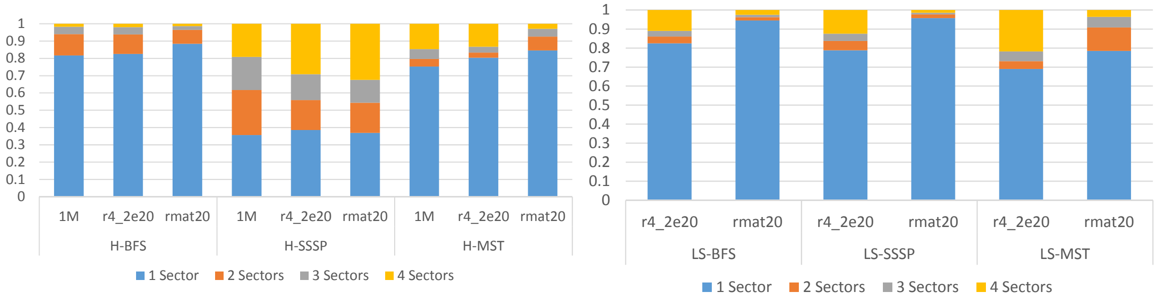


(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 38: L1 and L2 Bypass: IPC improvement over baseline for configuration with higher bandwidth

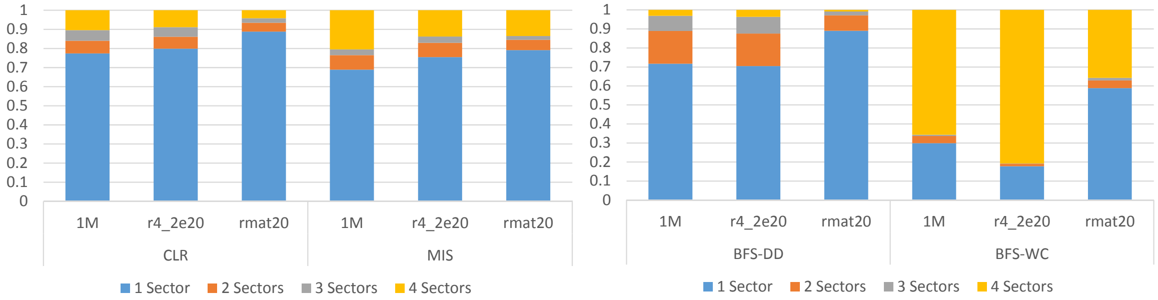
## *4.6 Dynamic Granularity Memory Hierarchy*

The third and final design aspect we consider is the granularity of the cache hierarchy. Cache hierarchies in GPUs (and CPUs as well) operate at the coarse granularity of cache blocks - in the event of a cache miss an entire cache block is fetched from the memory hierarchy and stored in the cache. Fetching entire cache blocks can be beneficial when data has spatial locality. In the absence of locality, a coarse granularity cache hierarchy wastes DRAM and interconnect bandwidth. A side effect of a coarse granularity cache hierarchy is increase in the queueing delays of memory requests. Graph algorithms kernels are irregular with data dependent memory accesses and accesses by threads executing these kernel often contain low spatial locality (both intra-thread and inter-thread). In addition, data dependent accesses by the large number of threads running concurrently results in accesses to many unique cache blocks making it difficult to exploit whatever locality may be present in the access



(a) HIPC Kernels

(b) Lonestar Kernels



(c) CLR and MIS Kernels

(d) Other BFS implementations

Figure 39: Number of sectors read from cache blocks inserted into the L1 cache

stream. Figure 39 shows the number of sectors read for each cache block inserted into the L1 cache. For all kernel except H-SSSP and BFS-WC for more than 50% of the cache blocks only one out of four cache sectors are read. The main loop with H-SSSP has three loads and a atomic instruction. Cache blocks updated by atomic instructions are fetched into the L2 only and thus the L1 is populated only by cache blocks fetched by the other three loads in the loop. Among the three loads, two loads have spatial locality and the other has temporal locality and could have spatial locality as well. In case of BFS-WC, a warp is assigned a chunk of vertices and each vertex is expanded in parallel by the threads in a warp. Thus for vertices whose neighbor list spans across cache sectors, multiple sectors are likely to be accessed in parallel. From Figure 39 we see that there is considerable potential for designing a cache hierarchy with fine grained accesses; such a hierarchy could utilize memory bandwidth more efficiently and reduce queueing delays as well.

Below we briefly discuss the mechanisms evaluated in this section.

### 4.6.1 Sector Caches

Sector caches [10, 33, 64] were initially introduced to reduce the overhead of tags for the data stored in a cache. In a sector cache, a tag is associated with each sector while the sector itself is divided into multiple equal sized sub-sectors which are the units of memory transfer. Each sub-sector in a sector has a valid bit which tracks the presence of the sub-sector. On a cache miss, only the requests subsector is fetched from memory. In case the sector to which the subsector belongs does not have an entry allocated in the cache an entry is allocated for the sector. The sector size in early sector caches were fairly large (1024B in IBM System/360 Model 85 [10]), this resulted in considerable waste of cache space and sector caches were done away with and were replaced by associative caches. The sector cache we consider uses cache block sized sectors which results in much less waste of cache space when compared to the original sector cache. The cache block size in our evaluations is 128B and the size of each subsector/subblock and the unit of memory transfer is 32B, thus the sector cache maintains a 4 bit mask for each cache block indicating the presence of sub-blocks. Sector caches can be beneficial for applications which do not have significant spatial locality.

### 4.6.2 Spatial Pattern Predictor

The sector cache described above does not consider any spatial locality that could exist in accesses to a sector. Suppose that intially sub-block A of a cache block is accessed, later on, sub-block B of the same cache block is accessed. The second access would result in a cache miss since the sector cache fetches only the requests sub-blocks. The Spatial Pattern Predictor (SPP) [14, 83] tries to tackle this by predicting the spatial access patterns of cache blocks and fetches not only the requested sub-block but also other sub-blocks of the cache block that are predicted to be accessed. If the predictions of the SPP are correct, then miss to a sub-block such as B above can be avoided thereby improving performance.

The SPP builds on a sector cache and like the sector cache it maintains a bit-mask/pattern indicating the sub-blocks present for each cache block in the cache. The access patterns of all the blocks in the cache logically form the Current Pattern Table (CPT). On a cache miss, an index is generated for the access, and the index is used to look up a Pattern History Table (PHT). The PHT stores previously seen access patterns and also provides predictions in case of a cache miss. If a matching PHT entry exists, the PHT entry is read to make a prediction about the spatial access pattern for the cache block being accessed; if a matching PHT entry is not found, then a default prediction is made and a PHT entry is allocated for the access. Along with the current access pattern, the CPT also stores a pointer to the PHT entry for the access. When a cache block is evicted, the corresponding PHT entry is updated with the access pattern for the cache block.

One of the factors determining the performance of the SPP is how the index to access the PHT is created; a combination of the PC of the memory instruction causing the access and the offset of the accessed sub-block within the cache block is known to work well for CPUs applications. In our evaluation we use a combination similar to the one used in Dynamic Granularity Memory System [83].

### 4.6.3 Locality Aware Memory Hierarchy

The locality Aware Memory Hierarchy (LAMAR) [61] takes a different approach to the SPP. At a high level, LAMAR uses a Bimodal Granularity Predictor (BIM) that tracks whether a cache block had coarse-grained of fine grained access the last time it was resident in the cache. The next time a cache block is accessed, LAMAR checks the BIM to determine whether the cache block had fine-grained or coarse-grained access in the past. If the cache block had fine-grained access, then only the request sub-block is fetched, else if the cache block had coarse-grained access, then the entire cache block is fetched. A cache block is deemed to have fine-grained access if two or fewer sub-blocks are accessed, else the cache block is deemed to have coarse-grained access. The intuition behind BIM is that due to low hit rates seen in GPGPU applications, SPP is unable to capture any locality that may exist in memory accesses. Thus rather than focus on collecting the pattern histories for PCs, LAMAR considers spatial locality in accesses of individual cache blocks.

The BIM starts with a default prediction that all cache blocks have coarse grained accessed. As cache blocks are evicted their sub-block access patterns are examined. If an evicted cache block is deemed to have fine-grained access, then the address of the cache block is entered into a bloom-filter. On a cache miss, the bloom-filter is queried for the address of the missing block. If the bloom-filter returns a hit, it indicates that the cache block had fine-grained access the last time it was the cache; for such blocks LAMAR fetches only the required sub-block. If the bloom-filter returns a miss then it indicates that the cache block previously had coarse-grained access and LAMAR fetches the full cache-block. If the ratio of insertions into the bloom-filter over the total number of evictions is above a threshold, LAMAR switches the bloom-filter to track coarse-grained blocks and default prediction to fine-grained accesses. Similarly, if the current predition is fine-grained, the bloom-filter would be tracking coarse-grained blocks and if the insertion ratio into the bloom filter is high, then BIM flips the default prediction and the predicate for insertion into the bloom-filter.

### 4.6.4 Neighbor Access Predictor

We also evaluate a alternate PC based mechanism called Neighbor Access Predictor. For each PC in application, Neighbor Access Predictor maintains an entry in the Pattern History Table (PHT). The PHT entry tracks how likely are the neighboring sub-blocks of a given sub-block are going to be accessed. This information is built over all the cache blocks accessed by the PC and thus is a summary of the history of the PC. While the SPP only stores the last pattern that was seen for a PC plus sub-block offset combination, Neighbor Access Predictor saves information from each cache block accessed by a PC so far and updates that information whenever a cache block fetched by that PC is evicted.

To be able to track the history of spatial patterns for a PC Neighbor Access Predictor uses a set of counters for each PC. On the eviction of a cache block fetched by a PC, the counters in the PHT entry for the PC are updated using the spatial pattern of the cache block and the index of the sub-block that had the first miss. Each counter in a PHT entry indicates whether a particular neighboring subblock of the subblock being accessed is likely to be accessed or not. When the pattern for an evicted block is being updated, if a neighboring subblock was accessed, the counter corresponding to that neighbor is incremented, on the otherhand if the neighbor was not accessed the corresponding counter is decremented. Upon a cache miss, the PC and the index of the sub-block are used to obtain a prediction from the PHT. If the counter value is above a threshold then the neighbor is likely to be accessed and Neighbor Access Predictor requests for the neighboring sub-block to be fetched as well. If a counter value is less than or equal to the threshold value then the neighbor is not fetched.

### 4.6.5 Methodology

The methodology is the same as described in Section 4.4.4. All the results presented use a sub-block or sub-sector size of 32B. The configurations of the evaluated dynamic granularity prediction mechanisms are as shown in Table 14.

### 4.6.6 Results

Figure 40 shows the IPC relative to the baseline for the different mechanisms applied when applied at the L1 caches only. When applied at the L1 caches only, the mechanisms (other than sector cache which does not do any predictions) predict the cache

Table 14: Fine-grained accesses: Evaluated mechanisms

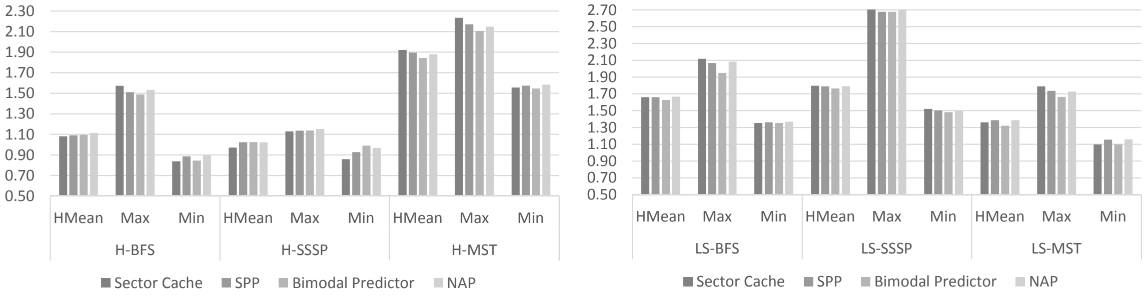
| Mechanism | Configuration |
|---|---|
| Sector Cache [10, 33, 64] | Fetch only the requested sector |
| Spatial Pattern Predictor (SPP) [14, 83] | 256-entry, 8 way set associative PHT |
| Bimodal Granularity Predictor (BIM) [61] | 2K-bit per bit array,<br>refresh period of 512 insertions<br>Skew threshold of 0.7,<br>Fine-grained threshold of 2 sectors |
| Neighbor Access Predictor | 32-entry PHT with<br>7 4-bit counters per PHT entry |

sectors that would be accessed only for L1 cache misses. All the evaluated mechanisms show similar performance behavior across all the benchmarks. Though there is performance improvement on average, there are cases with performance degradation as well. These cases usually occur when the number of active vertices is low. In addition to showing performance degradation when the number of active vertices are small, the sector cache also shows performance degradation with H-SSSP for r4_2e20 and rmat20 inputs. On average, the performance improvement provided by the Sector cache, SPP, Bimodal Predictor and NAP are 39%, 39%, 38% and 40%.

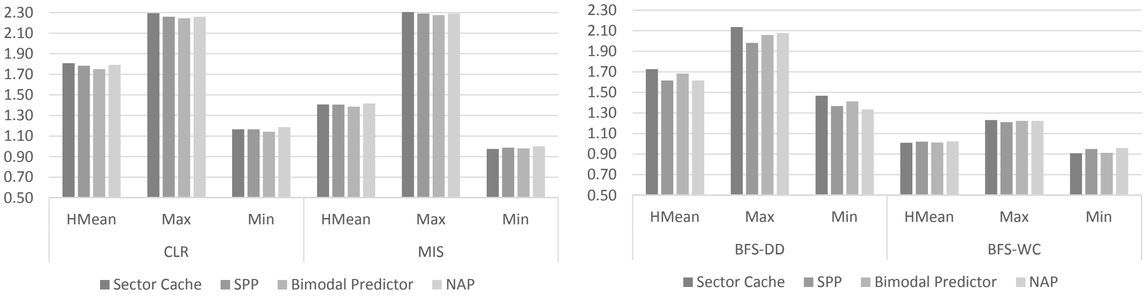
**Analysis of performance improvements**

The performance improvement due to fine-grained caching stems from the reduction in the DRAM and interconnect bandwidth consumption. H-BFS and BFS-WC which have lower MPKI compared to the other kernels shower lower performance improvement; in addition, there is a significant increase in MPKI for these kernels due to fine-grained accesses. Other kernels have higher MPKI and often lower increase in MPKI due to fine-grained accesses, thus fine-grained accesses show higher performance improvement for them.

Figure 41 shows the IPC relative to baseline when the mechanism are applied at both L1 and L2 caches. In this case the mechanisms predict the sectors that will be accessed on both L1 and L2 cache misses, not L1 misses only. Each cache

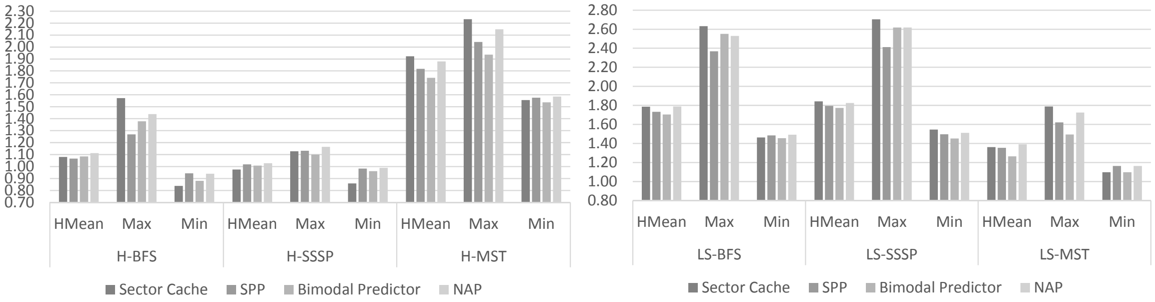


(a) HIPC Kernels (b) Lonestar Kernels

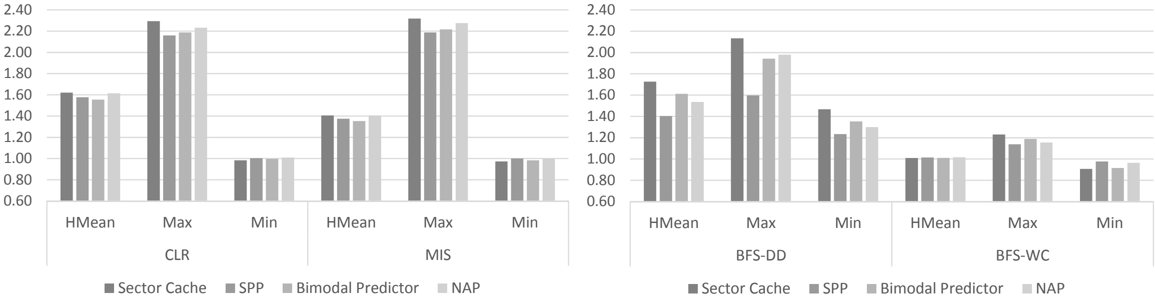


(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 40: Fine-grained accesses with predictions for L1 misses only: IPC improvement relative to baseline

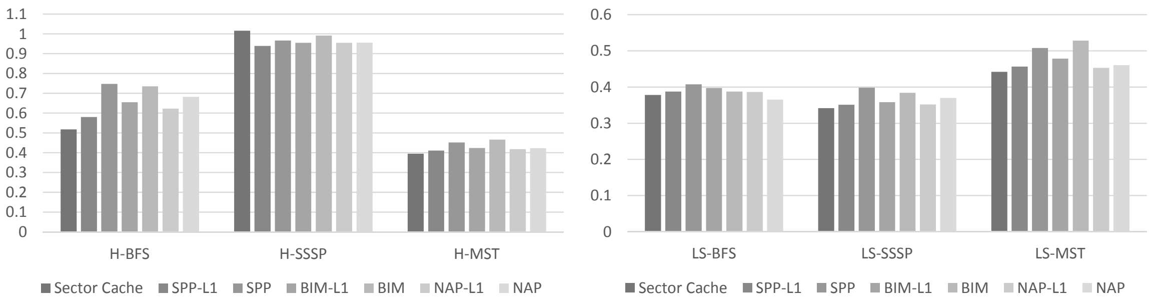


(a) HIPC Kernels (b) Lonestar Kernels



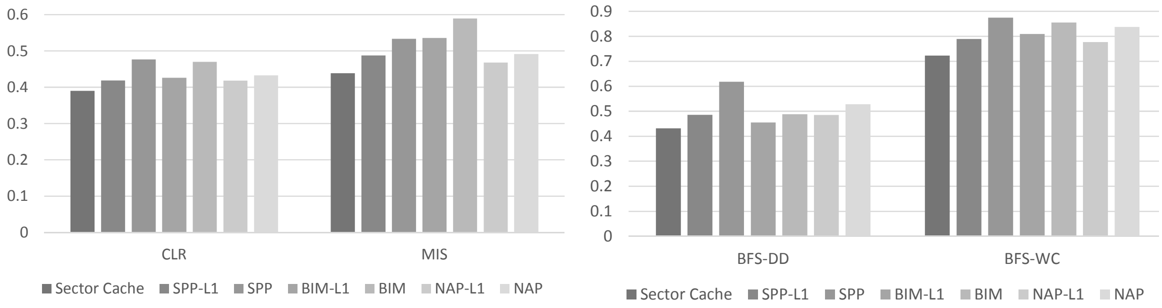
(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 41: Fine-grained accesses with predictions for L1 and L2 misses: IPC improvement relative to baseline



(a) HIPC Kernels

(b) Lonestar Kernels



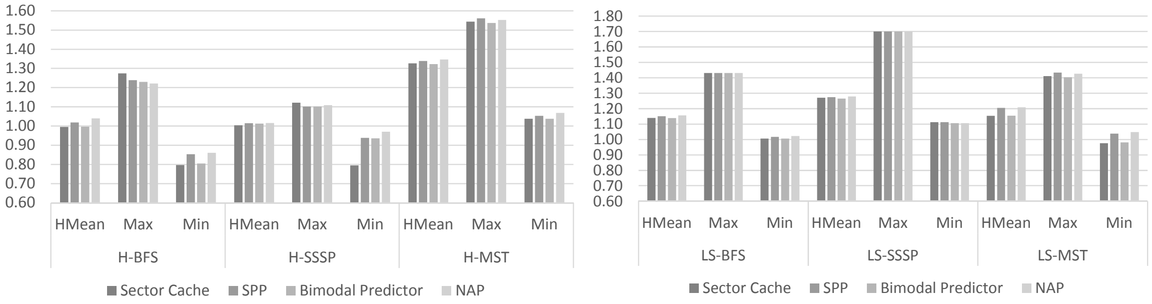
(c) CLR and MIS Kernels

(d) Other BFS implementations

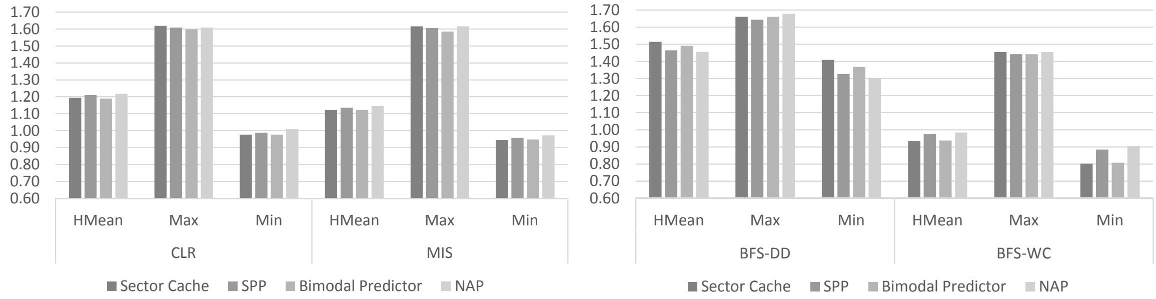
Figure 42: Fine-grained accesses with predictions for L1 misses only: Number of bytes read from DRAM relative to the bytes read by the baseline

has its own set of hardware structures required for making the predictions. The performance of the sector cache is the same in both Figure 40 and Figure 41 since the sector cache does not predict the cache block sectors that would be accessed on a cache miss. For the other mechanisms there is a slight reduction in the performance improvement seen when compared to applying the mechanisms at the L1 cache only with SPP and Bimodal Predictor showing a larger reduction in performance than the NAP. When the mechanisms are applied to the L2 caches, the predictive mechanisms overfetch data i.e. they predict and fetch sub-sectors which are not accessed later on. This overfetch increases bandwidth consumption and queuing delays and reduces the performance improvement.

Figure 42 shows the number of bytes read from DRAM by the different mechanisms relative to the baseline. *mechname-L1* denotes DRAM read traffic when *mechname* is applied to L1 caches only, *mechname* denotes DRAM read traffic when *mechname*



(a) HIPC Kernels (b) Lonestar Kernels



(c) CLR and MIS Kernels (d) Other BFS implementations

Figure 43: Fine-grained accesses with predictions for L1 misses only: IPC improvement relative to baseline for a configuration with higher bandwidth

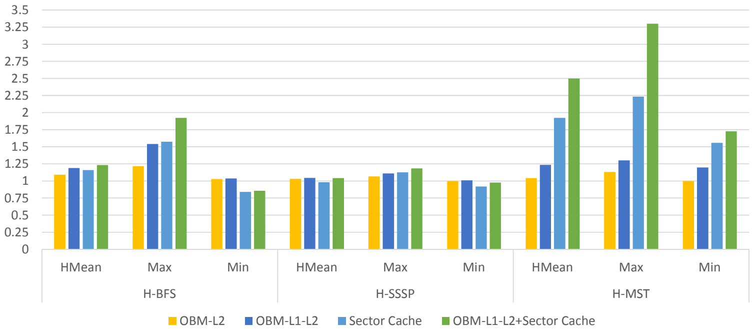
is applied to both L1 and L2 caches. From Figure 42 we see that the number of bytes read increases when mechanisms are applied to both L1 and L2 caches due to incorrect predictions.

#### *4.6.6.1 Configuration with increased bandwidth*

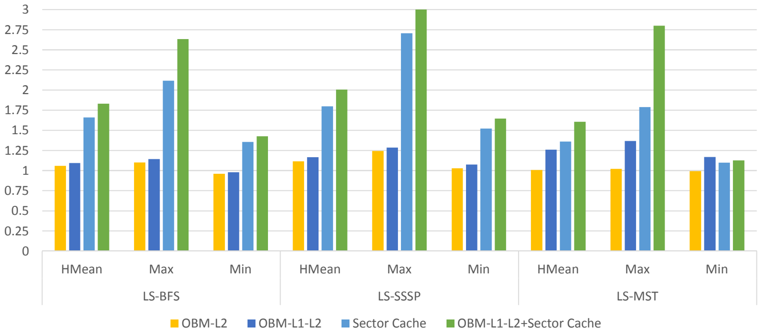
Figure 43 shows the performance relative to a coarse-grained hierarchy for sector cache, SPP, Bimodal Predictor and NAP on a machine configuration with 144 GB/s of memory bandwidth. On average, sector cache, SPP, Bimodal Predictor and NAP provide performance improvements of 14%, 15%, 14% and 17% over the baseline. These performance improvements are much lower than the improvements provided by the mechanisms for the earlier evaluated configuration which has a much lower bandwidth. At the lower bandwidth configuration, with coarse-grained memory accesses, kernel instances for many of the algorithms are bandwidth limited or have large bandwidth consumption because of which queueing delays for memory accesses are significant. The use of fine-grained alleviates the bandwidth problem considerably and in some instances where the use of fine-grained accesses does not increase the cache misses considerably, the performance improvement is quite significant (more than 2x). On the higher bandwidth machine, kernel execution with a coarse-grained memory hierarchy is not as severely bandwidth limited as in case of the lower bandwidth machine and often kernel execution with fine-grained accesses cannot utilize a significant portion of the available bandwidth. Thus, the performance improvement with a fine-grained memory hierarchy is not as signficant as in the case of the lower bandwidth machine. On the otherhand, the cases which show performance degradation are more often and show higher degradation in case of the higher bandwidth machine. These often tend to be the instances where the number of memory accesses are fewer compared to other kernel instances - in such cases SPP and NAP tend to perform better than the sector cache and bimodal predictor.

## *4.7 Combining the evaluated mechanisms*

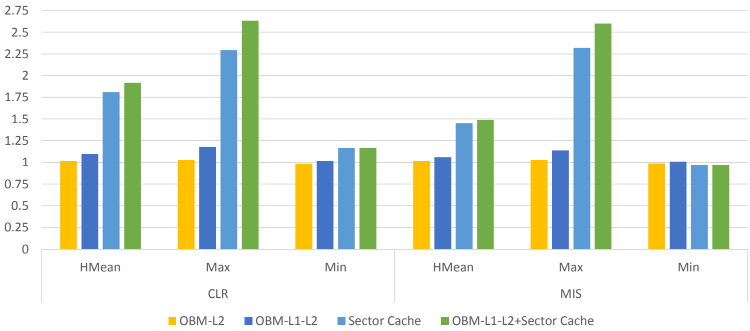
The evaluated mechanisms show benefit to varying extents. While exclusion provides some benefit for graph algorithms, the best performing cache bypass mechanism and a fine-grained memory hierarchy provide more benefit in comparison to exclusion. Note that we applied cache bypassing to both L1 and L2 caches instead of using cache bypassing at the L2 caches only. We try to see whether combining the different evaluated mechanisms can provide more benefit. Since the use of exclusion complicates the bypass mechanism, we combine bypassing at the L1 and L2 caches with the use of a fine-grained memory hierarchy. Figure 44 show the performance of combining cache bypassing with a fine-grained memory hierarchy relative to the baseline (OBM-L1-L2+Sector Cache). Also shown are the performances due to cache bypassing at L2 caches only (OBM-L2), at both L1 and L2 caches (OBM-L1-L2) and



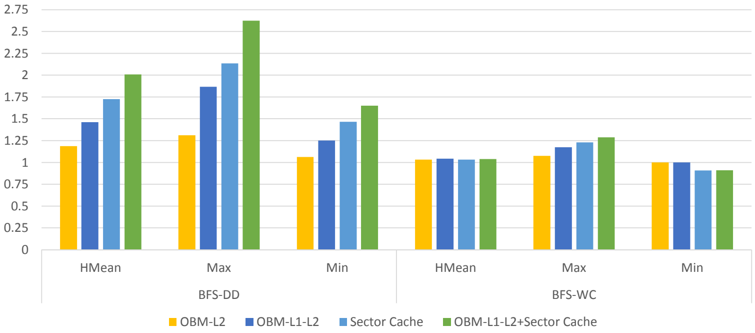
(a) HIPC Kernels



(b) Lonestar Kernels



(c) CLR and MIS Kernels



(d) Other BFS implementations

Figure 44: Bypass, fine-grained accesses and their combination: IPC improvement relative to baseline

the use of a fine-grained memory hierarchy (Sector Cache). On average, OBM-L1-L2+Sector Cache provides a performance improvement of 54% over the baseline. The performance of the combined mechanism is 42% more than what is obtained with bypassing alone, and 12% more than what is obtained with using a fine-grained memory hierarchy alone. For H-BFS, H-SSSP, BFS-WC the perfomance improvement is lower compared to the other algorithms. For these kernels, the sector cache itself does not provide much performance improvement due to the fact that for these kernels a significant fraction of sector misses are seen i.e. a way has been allocated in the cache for the cache block containing the sector but the sector is not present.

## *4.8 Summary*

In Section 4.4 we evaluated different options for the inclusion property of a GPU cache hierarchy. In addition to the baseline non-inclusive cache we evaluated an exclusive cache, Flexclusion [68] - a cache hierarachy which chooses between non-inclusion and exclusion dynamically at runtime - and R-Exclusion - a mechanism that does exclusion or non-inclusion for individual memory allocation regions. Exclusion, Flexclusion and R-Exclusion provide performance improvements of 6%, 4% and 5% over the baseline, with R-Exclusion having much lower on-chip traffic than either exclusion or Flexclusion and avoiding some of the negative cases seen with exclusion and Flexclusion.

In Section 4.5 we evaluated cache bypass mechanisms for the GPU on-chip cache hierarchy for graph algorithms. We evaluated an address based mechanism (MAT) [40], a PC-based mechanism (OBM) [45], a memory region and PC agnostic mechanism (AB) [27] and a scheme that tries to use sharing attributes of memory regions allocated in a GPU (B+C). Our results show that for graph algorithms, bypassing at the L1 caches in addition to bypassing at the L2 caches provides additional opportunities for performance improvement. Of the evaluated mechanisms, Optimal Bypass

Monitor (OBM), the PC-based mechanism shows the best performance improvement with an improvement of 14% across all evaluated kernels when it is applied to both L1 and L2 caches. When OBM is applied to only the L2 cache, the performance improvement is 5%.

In Section 4.6 we evaluated different mechanisms for a dynamic granularity (or fine-grained) memory hierarchy. We evaluated a sector cache [10, 33, 64], the Spatial Pattern predictor [14, 83], the Bimodal Insertion Predictor [83] and a predictor similar to the SPP call the neighbor access predictor. The different predictors build on top of the sector cache and they predict for each cache miss which cache sectors other than the requested sectors are likely to be accessed. The predictors can be applied to the L1 cache and to both the L1 and L2 caches. If no predictions are applied at the L2, then requests from the L1 caches are forwarded as is when there is a cache miss. Applying the predictors to the L2 cache could help in the case different cores access different sectors of the same cache block. Our results show that applying the predictors to the L1 cache only is more beneficial that applying them to both L1 and L2 since when the predictors are applied to the L2 caches they make many incorrect predictions resulting in many additional bytes of data being fetched without being accessed. When the mechanisms are applied to the L1 cache only, their performances are similar to the baseline sector cache which provides a performance benefit of 39% on average.

Finally, in Section 4.7 we evaluated a combination of cache bypass and a dynamic granularity memory hierarchy. Our results show that combining the two provides benefit that exceeds the benefit provided by the individual mechanisms. A combination of OBM and Sector caches provides a benefit of 54%. While the two mechanisms individually provide benefits of 14% and 39%. Thus graph algorithms can benefit from cache hierarchies that do fine-grained memory access and cache bypass.

# CHAPTER V

# CHARACTERIZATION OF GRAPH ALGORITHM IMPLEMENTATION STRATEGIES

In this chapter we examine the impact of different graph algorithm implementation strategies on graph algorithm performance, power and energy consumption. We explain these implementation strategies using Breadth First Search (BFS) as an example. In fact, these implementation strategies were originally employed for developing different implementations of BFS, however, we evaluate the same strategies for Single Source Shortest Path (SSSP) and Graph Coloring (CLR) as well.

First we start of with a brief description of the different implementation strategies that we evaluate using BFS as an example.

## 5.1 Topology Driven implementation

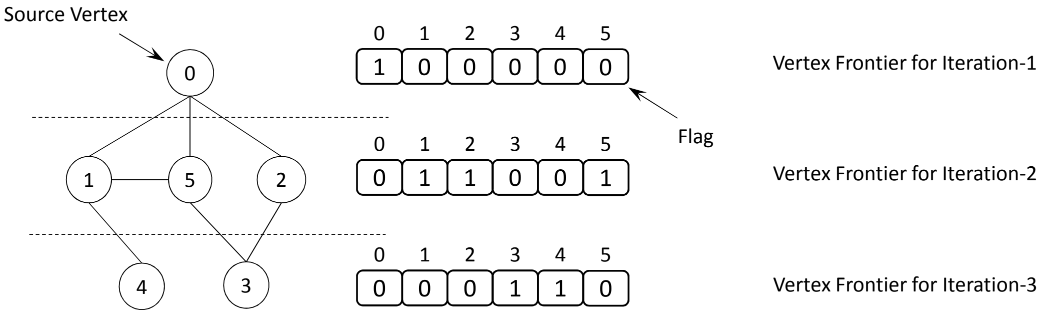
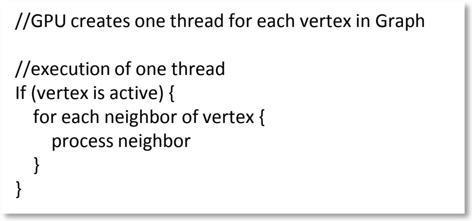

Source Vertex
0
1
5
2
4
3
0 1 2 3 4 5
1 0 0 0 0 0
Flag
Vertex Frontier for Iteration-1
0 1 2 3 4 5
0 1 1 0 0 1
Vertex Frontier for Iteration-2
0 1 2 3 4 5
0 0 0 1 1 0
Vertex Frontier for Iteration-3

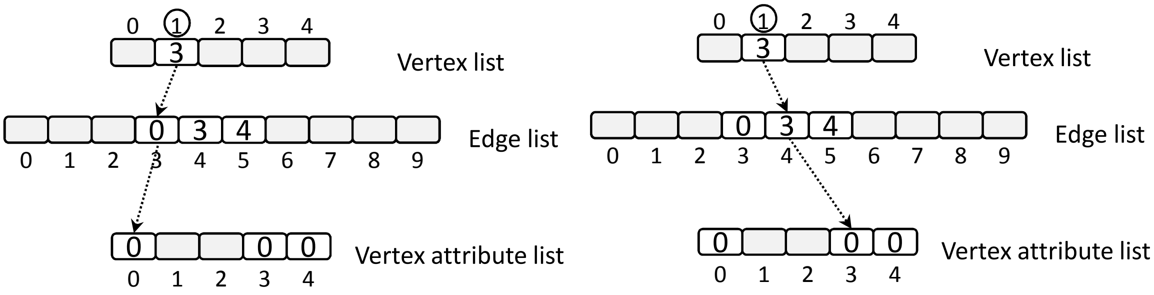
Figure 45: Topology Driven BFS

In the topology driven [58, 52] approach the graph operation is applied to all vertices in the graph, however, some vertices may not have any work to do. In one approach, the vertices that have work to be performed are identified using a vertex frontier which is an array of flags. In an iteration of the algorithm, the flag for the



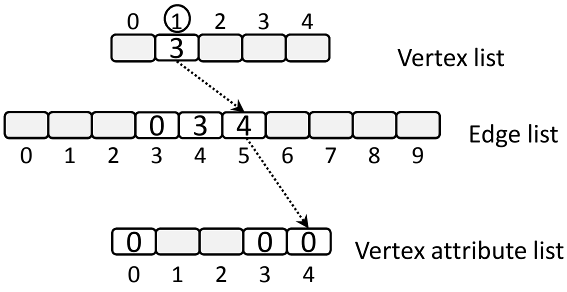
```
//GPU creates one thread for each vertex in Graph

//execution of one thread
If (vertex is active) {
    for each neighbor of vertex {
        process neighbor
    }
}
```

Figure 46: Pseudocode for Topology Driven BFS



(a) First iteration

(b) Second iteration



(c) Third iteration

Figure 47: BFS in a Thread-Centric implementation

vertices active in the iteration are set, while the flags for the inactive vertices are cleared. Figure 46 shows the pseudocode for a BFS search kernel implemented using the topology driven approach. Figure 45 shows BFS on an example graph with vertex 0 as the source or root. The vertex frontier array is shown on the right; it's size is equal to the number of vertices in the graph. In the first iteration of BFS, only vertex 0 is active, in the second iteration vertices 1, 2 and 5 are active and so on.

### 5.1.1 Thread Centric vs Warp Centric

In a thread-centric approach, each GPU thread processes one vertex in the graph. In case of BFS, processing a vertex means expanding the list of neighbors of the vertex. In a thread-centric approach, a thread expands the neighbor list of the vertex assigned to it in a loop, one vertex in each iteration. Figure 47 shows the processing of vertex 0 by a thread. In the first loop iteration, neighbor vertex 1 is processed, in the second loop iteration vertex 5 is processed and so on. Other threads in the warp/block/grid process the vertices assigned to them. In our experiments we label the thread-centric topology driven approach with a vertex frontier [31] as topology driven.

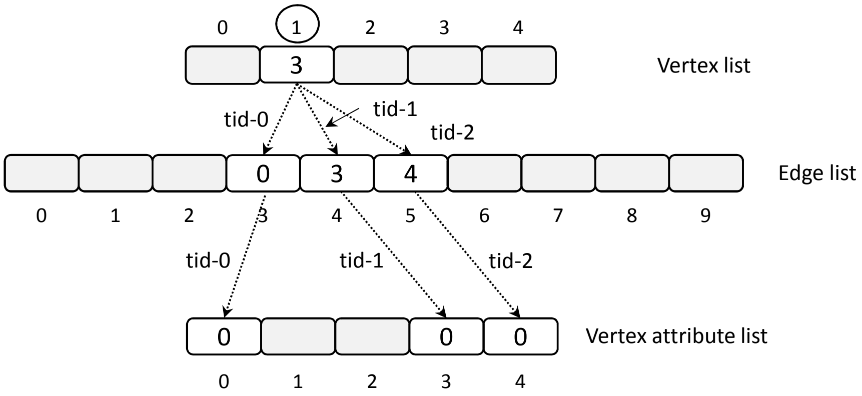
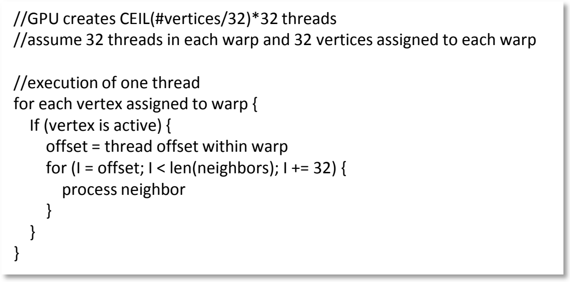


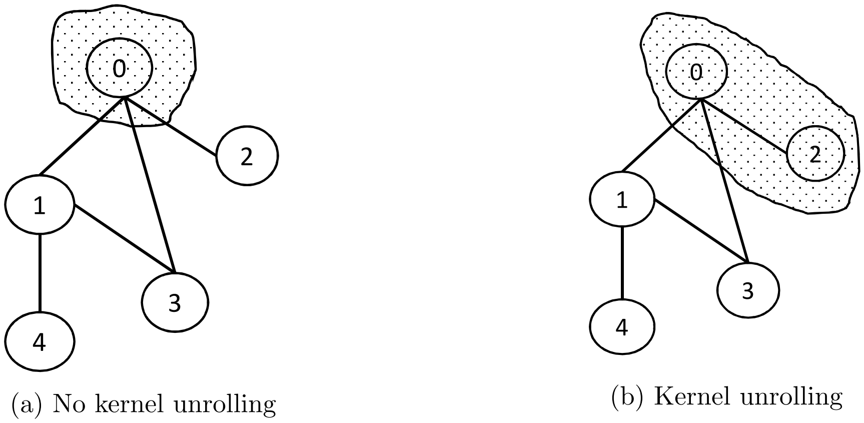
Figure 48: Warp Centric BFS



```
//GPU creates CEIL(#vertices/32)*32 threads
//assume 32 threads in each warp and 32 vertices assigned to each warp

//execution of one thread
for each vertex assigned to warp {
  If (vertex is active) {
    offset = thread offset within warp
    for (I = offset; I < len(neighbors); I += 32) {
      process neighbor
    }
  }
}
```

Figure 49: Pseudocode for Warp-Centric BFS

In a warp-centric [35, 21] approach, rather than only one single thread expanding a vertex, a warp of threads collectively expand one vertex. The neighbors of the vertex being expanded are processed in parallel by all the threads in a warp. Figure 49 shows



(a) No kernel unrolling

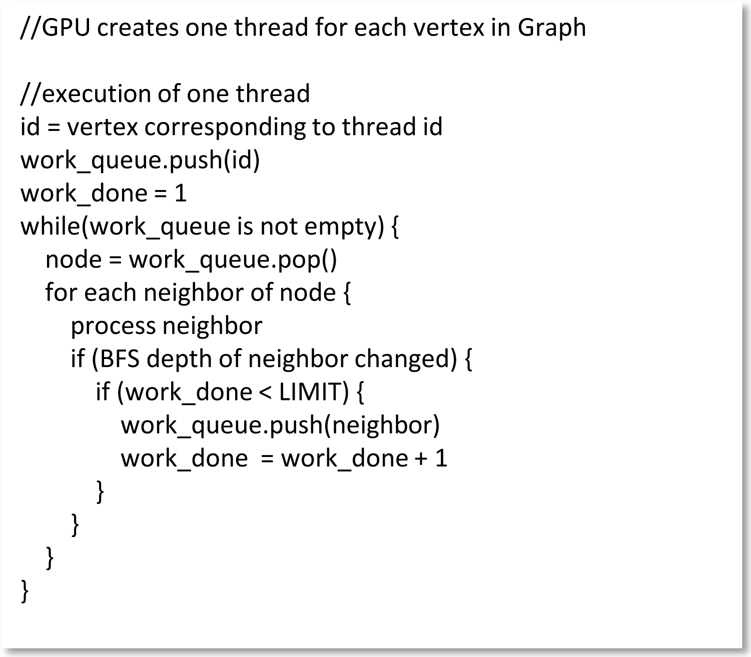
(b) Kernel unrolling

Figure 50: No Kernel Unrolling vs Kernel Unrolling : Vertices processed by a thread in one kernel invocation

the pseudocode for the warp-centric BFS we evaluate. In the warp-centric approach, each warp is assigned a chunk of vertices and the warp collectively processes one vertex after another from the chunk. Figure 48 shows the parallel processing of the neighbors of vertex 0 by the threads in a warp. In our evaluations we label the warp-centric, topology driven implementation with a vertex frontier as warp-centric. The warp-centric approach has the benefit of processing a vertex's neighbors in parallel, hence it is more suited for processing vertices that have large degree. In addition, since the compact adjacency list representation stores the edge list (i.e. neighbors) as an array, accessing the neighbors by the threads in a warp results in coalesced accesses. In case of the thread centric approach, each thread accesses consecutive array locations across different loop iterations while accesses by threads in the same warp are likely to be uncoalesced.

### 5.1.2 Kernel Unrolling

An alternate approach to using a vertex frontier in a topology driven implemention is to not using a vertex frontier at all. In such an implemention, all vertices are expanded in all iterations of the algorithm. In every iteration, each vertex tries to update the distance of its neighbors from the source vertex. The absence of a



```
//GPU creates one thread for each vertex in Graph

//execution of one thread
id = vertex corresponding to thread id
work_queue.push(id)
work_done = 1
while(work_queue is not empty) {
  node = work_queue.pop()
  for each neighbor of node {
    process neighbor
    if (BFS depth of neighbor changed) {
      if (work_done < LIMIT) {
        work_queue.push(neighbor)
        work_done  = work_done + 1
      }
    }
  }
}
```

Figure 51: Pseudocode for BFS with Kernel Unrolling

vertex frontier allows the use of an optimization called Kernel unrolling [4]. A vertex frontier requires that all vertices active in the current BFS level are expanded before the vertices in the next level are expanded. Without the vertex frontier, nodes in different BFS levels can be expanded in the same kernel launch, effectively unrolling the kernel launches. Figure 51 shows the pseudocode for the unrolled BFS kernel we evaluate. The kernel unrolled version we evaluate uses one thread to process one vertex. Each thread also processes additional vertices whose distance it has updated and there is an upper limit on the number of vertices processed by a thread. Figure 50 shows the vertices processed by a thread without and with kernel unrolling. In the kernel unrolling case, the thread could process additional vertices as well. Since all nodes are expanded in all kernel launches, an implementation without a vertex frontier does considerable additional work when compared to an implementation with a vertex frontier. In addition, the unrolling itself may end up doing useless work depending on the graph structure.

## *5.2 Data Driven implementation*

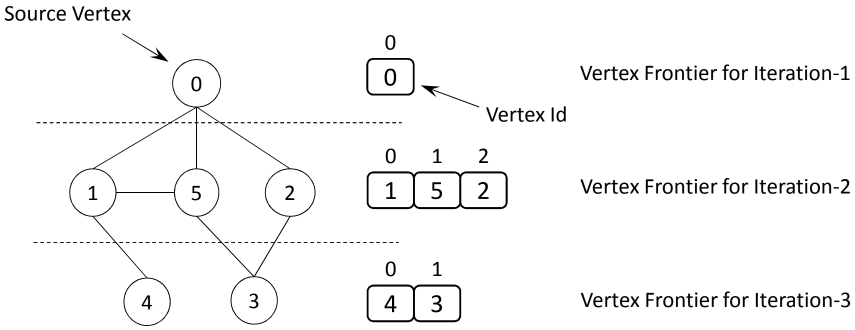
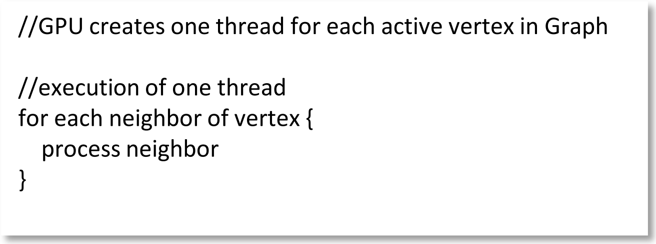

Source Vertex
0
0
0
Vertex Id
Vertex Frontier for Iteration-1
1
5
2
0 1 2
1 5 2
Vertex Frontier for Iteration-2
4
3
0 1
4 3
Vertex Frontier for Iteration-3

Figure 52: Data Driven BFS



```
//GPU creates one thread for each active vertex in Graph

//execution of one thread
for each neighbor of vertex {
   process neighbor
}
```

Figure 53: Pseudocode for Data-Driven BFS

Unlike in a topology driven implementation, in a data driven [58] implementation the graph operation is applied to only those vertices that have work to perform; these vertices are identifed using a vertex frontier. However, unlike the vertex frontier in a topology driven implementation, a vertex frontier in the data driven implemention contains the id of vertices that are active rather than having a flag for each vertex in the graph. The creation of a vertex frontier for a data-driven implementation requires the use of prefix sum which can be expensive where a large number of vertices are present in the vertex frontier. The pseudocode for a data-driven BFS is shown in Figure 53. The data driven implementation [49, 21] that we evaluate first updates newly visited vertices in a per thread block queue (vertex frontier) maintained in shared memory using shared memory atomic operations. If the local queue is full, a queue maintained in global memory (vertex frontier) is updated using atomic operations. At the end of the execution of all threads in a block, all the vertices in the local queue in shared memory are copied to the global queue (vertex frontier) with the help of atomic operations. The data driven approach has the benefit that only as many threads as number of vertices in the vertex frontier are created and could be suitable for graphs which could have small vertex frontiers.

## *5.3 Methodology*

Table 15: Characterization: Input Graphs

| Input | Category | #vertices | #Edges | Avg. Degree |
|---|---|---|---|---|
| in-2004 | Clustering | 1382908 | 27182946 | 19.66 |
| coPapersDBLP | Social Networks | 540846 | 30491458 | 56.41 |
| great-britain | Street Networks | 7733822 | 16313034 | 2.11 |
| er-fact1.5-scale22 | Erdos Reyni Graph | 4194304 | 95966450 | 22.88 |
| rgg_n_2_22_s0 | Random Graph | 4194304 | 60718396 | 14.48 |

Table 16: Characterization: Execution Statistics

| Input | BFS | SSSP | CLR |
|---|---|---|---|
| | Iterations | Iterations | Iterations |
| in-2004 | 26 | 64 | 708 |
| coPapersDBLP | 15 | 76 | 423 |
| great-britain | 6389 | 6432 | 12 |
| er-fact1.5-scale22 | 7 | 40 | 68 |
| rgg_n_2_22_s0 | 1141 | 2549 | 38 |

We measure the execution time and power measurement for the different BFS implementations discussed above for a large number of input graphs from the 10th DIMACS Implementation Challenge [1] on a Tesla K40c GPU. From the measured execution time and power we estimate the energy consumed by each implementation strategy for the different inputs. Though we evaluate for a large number of graphs, below we present data for only input of each evaluated graph category. The input graphs for which we present data are shown in Table 15. In addition, we also evaluate the same implementation strategies for SSSP and also Graph Coloring (CLR). For SSSP we implement the data driven and warp centric versions in addition to the existing topology driven versions. For Graph Coloring we implement all the evaluated versions.

### 5.3.1 Measuring Execution Time

The execution time of an implementation is measured using the Cuda Event API provided by the CUDA Runtime Library [2]. The Cuda Event API enables programmers to measure the time elapsed between events in ms resolution. Often, the execution time of the evaluated algorithms is very small and there are likely to be variations in the execution time of an algorithm. To avoid errors from measuring single algorithm runs, we repeat an algorithm multiple times until the total execution time on the GPU is about 20s. We then divide the total execution time on the GPU with the number of repetitions to get the execution time for one complete algorithm execution. To repeat algorithm execution, we modify the application code to reinitialize the state of the data structures used by the algorithm and launch the algorithm kernels again. The number of times the algorithm is to be repeated can be passed on as a commandline argument.

### 5.3.2 Power Measurement

We develop an NVIDIA Management Library-based (NVML) [6] tool for measuring the GPU power consumption. NVML provides a set of API for querying and managing the GPU devices attached to a CPU host. We use the *nvmlDeviceGetPowerUsage*() function to query the power consumption of a GPU. The resolution of power measurements provided by NVML is 1 mW with the power values being updated at the frequency of 60Hz. The tool continuously queries the GPU power consumption in a while loop and logs the average power value over a 500ms interval to a file.

### 5.3.3 Estimating Energy

Using the trapezoidal rule we estimate the energy consumed by the GPU from the power values collected by the power tool and the start of execution and the end of execution timestamps collected during the timing runs.

## *5.4 Results*

We first present the results and analysis for different BFS implementations and then present the results for SSSP and CLR.

### 5.4.1 Evaluation of different BFS implementations

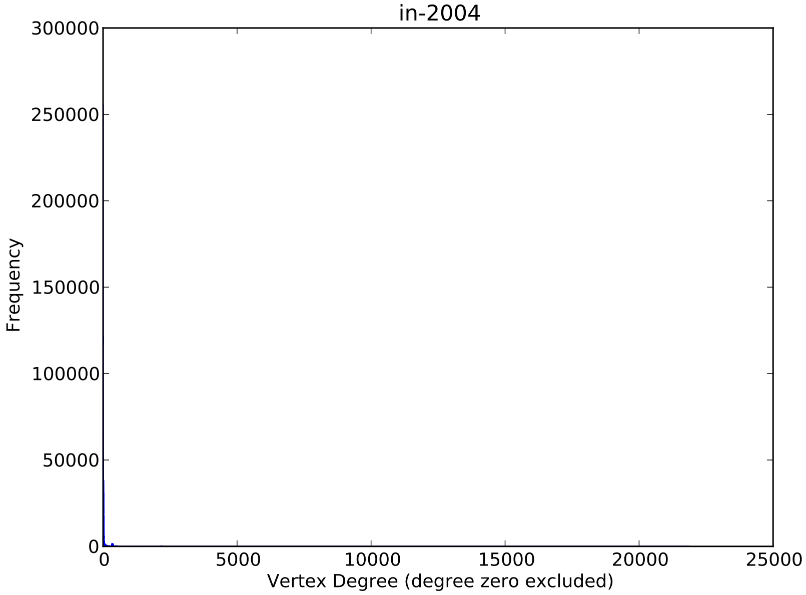
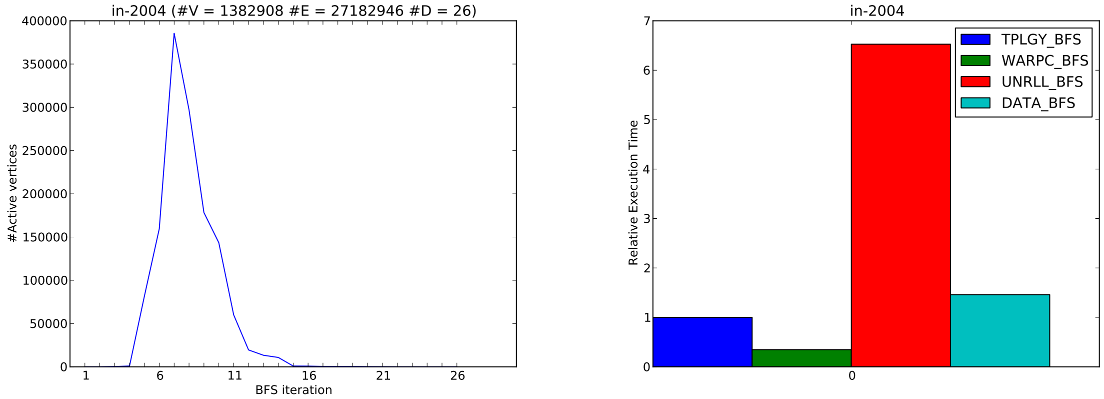


Figure 54: Frequency of Vertex Degrees for in-2004

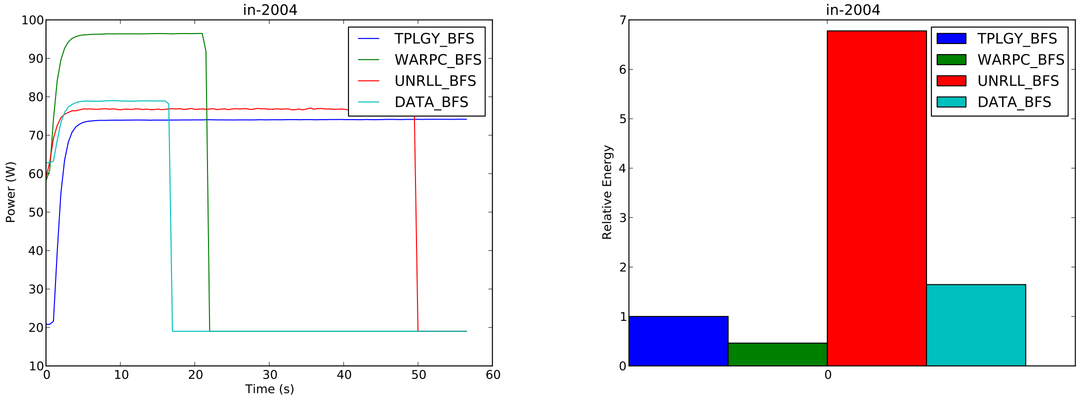
#### *5.4.1.1 Input in-2004*

Figure 54 shows the vertex degree distribution for input in-2004; we see that a large number of nodes have very small degree, but many nodes have large degree as well. Figure 55a shows the number of active vertices in each iteration of a topology/data-driven BFS and also the relative performances of the different implementations. We



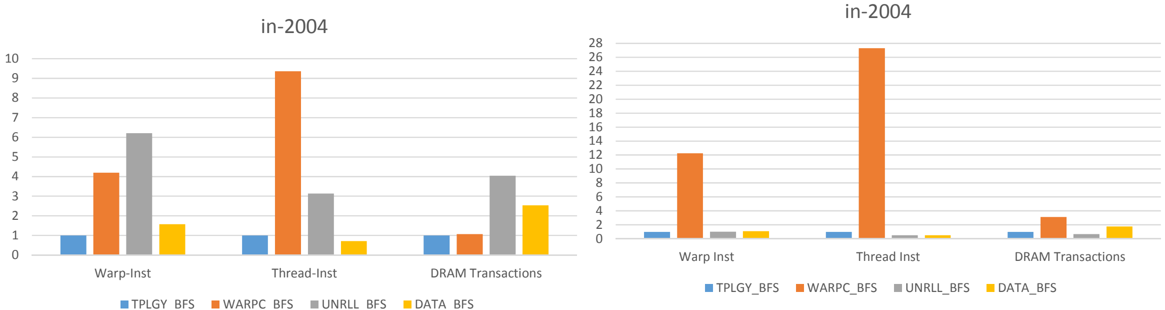
(a) Active vertices in each BFS step

(b) Relative execution time



(c) Power consumption

(d) Relative energy consumption time



(e) Instruction count and DRAM transcations

(f) Instruction count and DRAM transactions per cycle

Figure 55: BFS: Results for execution with in-2004

see that for in-2004 the BFS search lasts only 26 iterations and we also see that among the evaluated BFS implementations, the warp-centric BFS performs the best, followed by the topology driven BFS and then the data driven BFS and finally the kernel unrolled version. Since many nodes with a large degree are processed within a few BFS iterations, the warp-centric BFS was likely to do well for in-2004. Figure 55e shows the number of warp-instructions (incremented by one for each instruction executed by a warp), the number of thread-instructions (incremented by one for each active thread in an executed instruction and the number of DRAM transactions for the different implementations. We see that the kernel unrolled version not only executes the most instructions, it also has the most DRAM transactions as well. The warp-centric approach also executes many warp-instructions since it processes each vertex one after the other.

When it comes to energy consumption (Figure 55d), the relative energy consumption by the different versions follows the trend in execution time. However, in terms of power consumption (Figure 55c), warp-centric BFS has the highest power consumption (around 95W), where as the other implementations consume lower (between 70W and 80W). From among the four implementations, the number of warps instructions executed per cycle, and the number of DRAM transactions per cycles are highest for warp-centric BFS (Figure 55f), we could hence argue that it has the highest power consumption as well. Warp-centric BFS also has more active threads per instruction than the other implementations.

#### *5.4.1.2 Input coPapersDBLP*

The degree distribution in case of coPapersDBLP (Figure 56) has a curve instead of a sharp knee as in case of in-2004. There are a large number of vertices with small degree, some vertices with large degree with many vertices with an intermediate degree value; however, the degree values are not as high as in the case of in-2004. Still, the

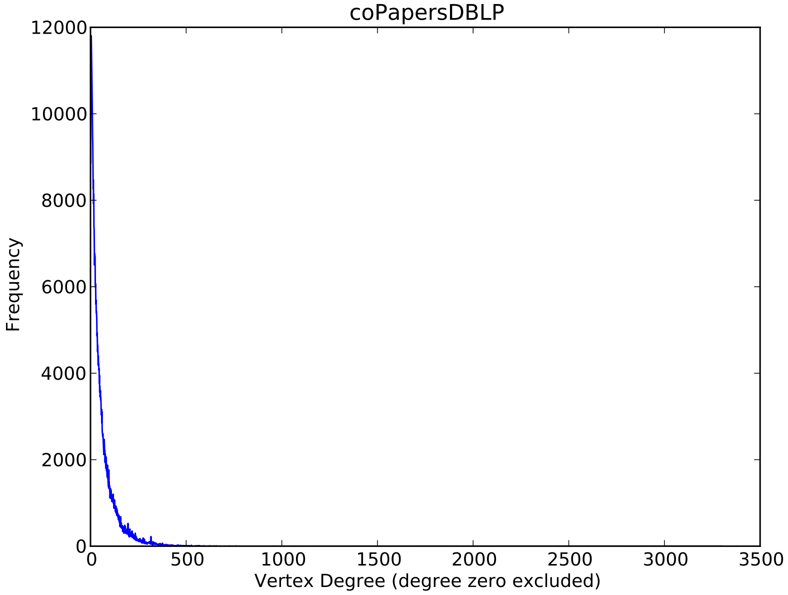
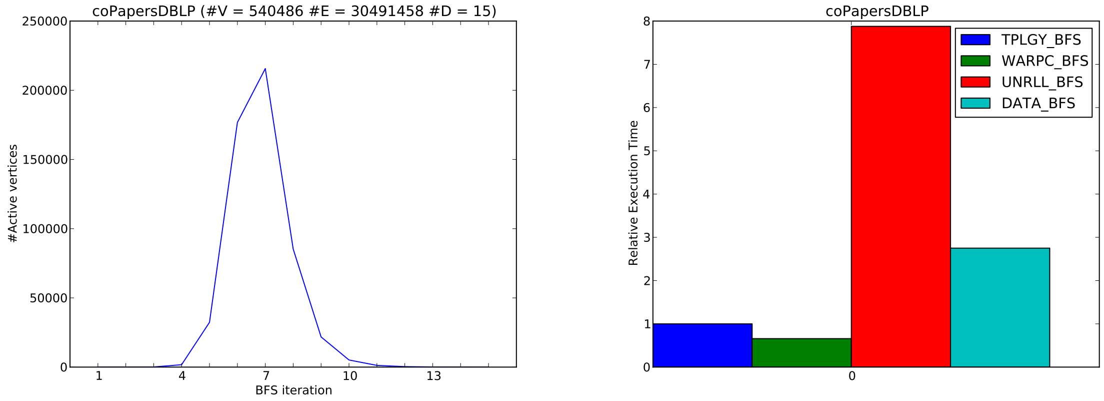


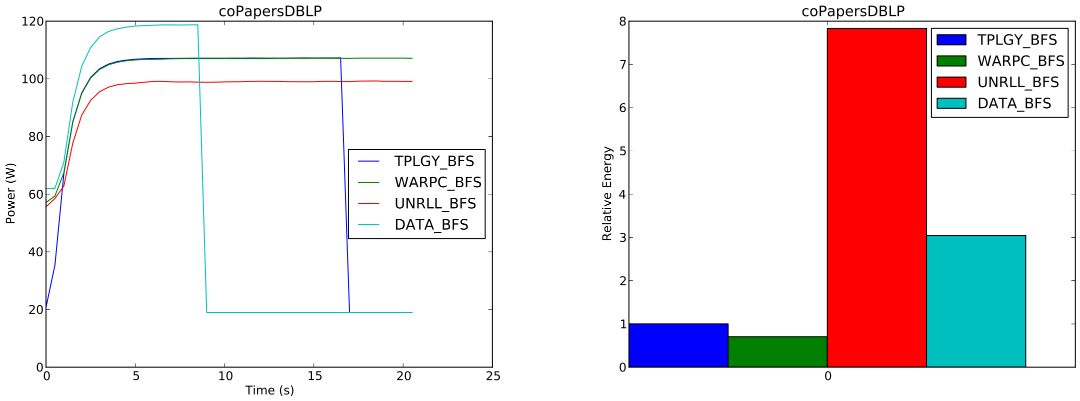
Figure 56: Frequency of Vertex Degrees for coPapersDBLP

warp-centric BFS still continues to show the best performance and energy consumption among the evaluated implementations. However, the power consumption trends are different when compared to the trends for in-2004. While in case of in-2004, the warp-centric BFS has the highest power consumption, in case of coPapersDBLP, it is the data-driven BFS which has the highest power consumption. For coPapersDBLP, the data-driven BFS has the most DRAM transactions per cycles, this could be a reason for the high power consumption of data-driven BFS for coPapersDBLP. When comparing the degree distribution of coPapersDBLP to the degree distribution of in-2004 we see that coPapersDBLP has many nodes with intermediate degree as well, this could mean that in case of coPapersDBLP, that the atomic updates to the global frontier queue are more common than in case of in-2004 and hence data-driven BFS has more DRAM transactions per cycle for coPapersDBLP than in case of in-2004.



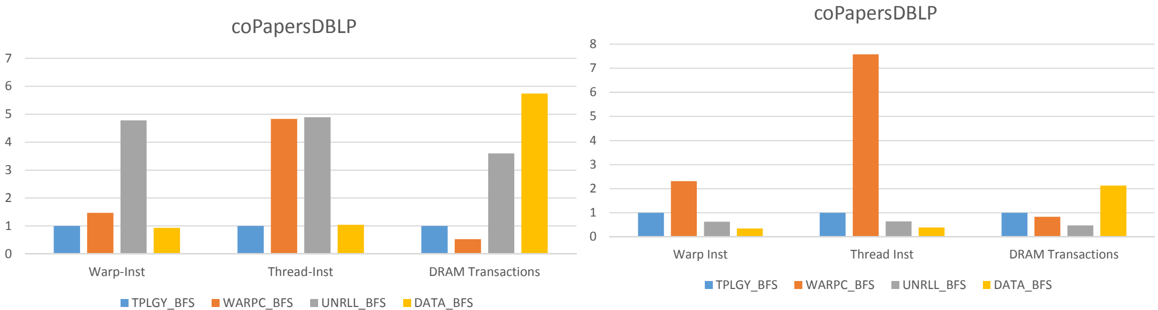
(a) Active vertices in each BFS step

(b) Relative execution time



(c) Power consumption

(d) Relative energy consumption time



(e) Instruction count and DRAM transcations

(f) Instruction count and DRAM transactions per cycle

Figure 57: BFS: Results for execution with coPapersDBLP

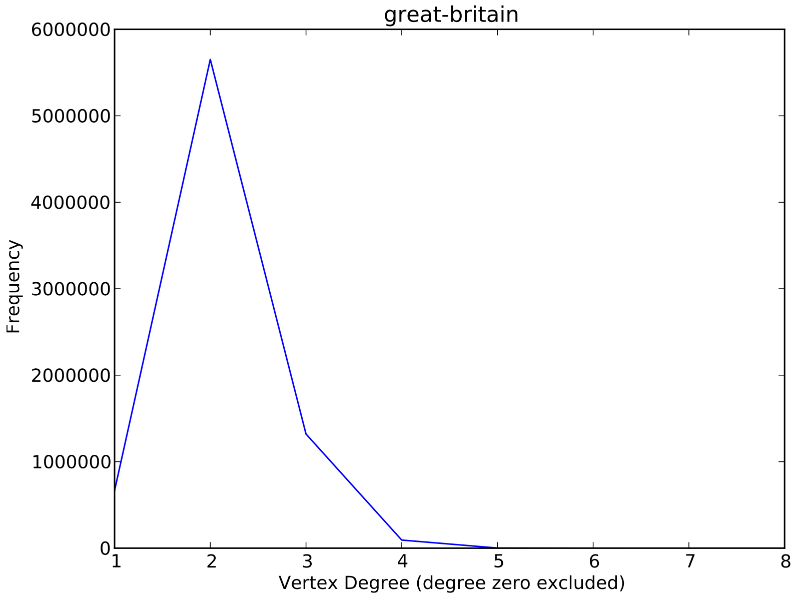
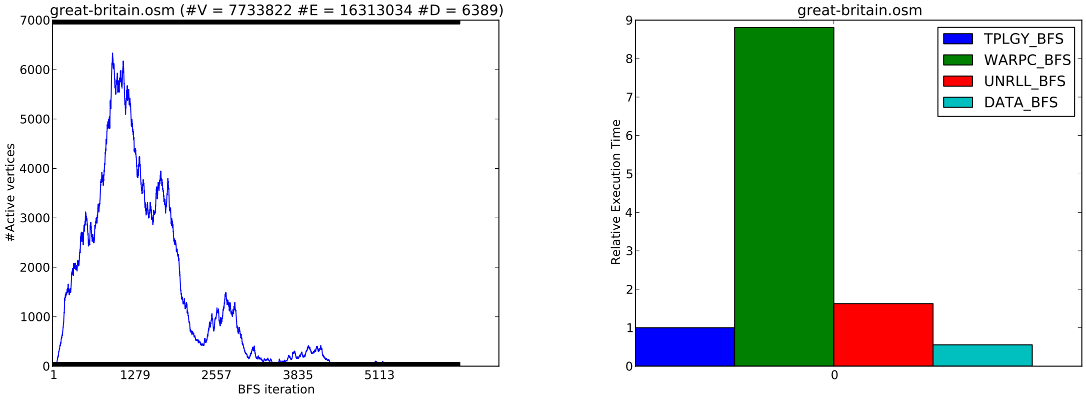


Figure 58: Frequency of Vertex Degrees for great-britain

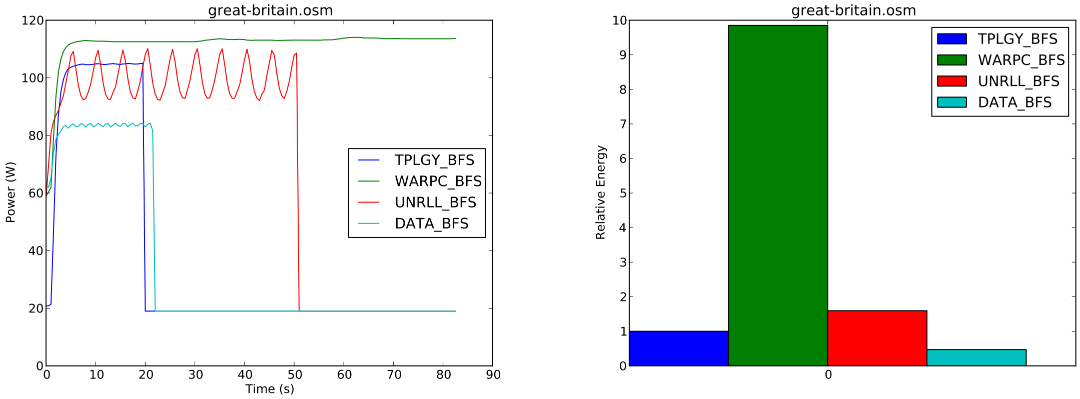
#### *5.4.1.3 Input great-britain*

great-britain is a street network and the degrees of all vertices in the graph are very small, most nodes have degree between 1 and 3 (Figure 58). Due to the small degrees of vertices there is no explosion in the number of active vertices after a few iterations as seen in case of other inputs so far; the BFS search progresses slowly, and take around 6400 iterations to complete (Figure 59b). Since the vertex frontier in each iteration is extremely small compared to the total number of vertices, the data-driven approach performs the best for great-britain. The toplogy-driven BFS is the next best, while the warp-centric BFS is the worst. In case of the warp-centric BFS, there is no thread level parallelism in processing of vertices assigned to a warp, There is parallelism is expansion of vertices, but expansion of individual vertices is serialized unlike in the case of other implementations. The parallelism in the expansion of individual vertices does not yield much benefit in case of great-britain due to small vertex degrees and sequential processing of individual vertices leads to slowdown. In terms of power consumption (Figure 59c), the data-driven approach has the lowest values, while the



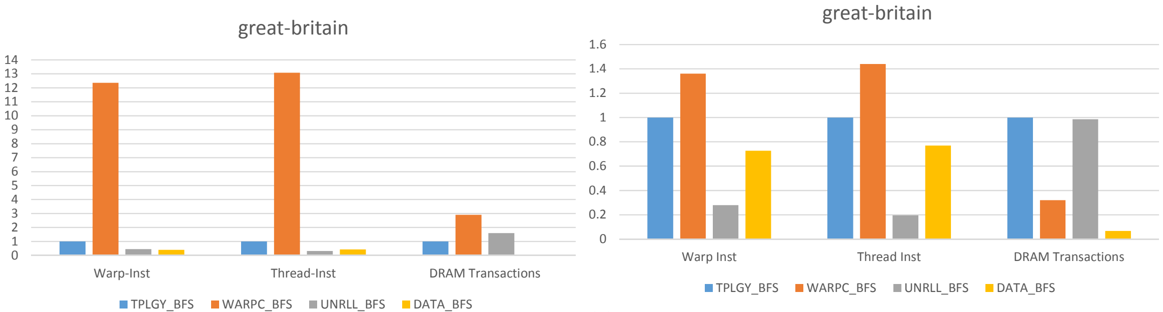
(a) Active vertices in each BFS step

(b) Relative execution time



(c) Power consumption

(d) Relative energy consumption time



(e) Instruction count and DRAM transcations

(f) Instruction count and DRAM transactions per cycle

Figure 59: BFS: Results for execution with great-britain

warp-centric approach has the highest. Among the evaluated versions, the data-driven BFS has the fewest instructions executed and the fewest DRAM accesses and hence it has the lowest power consmption as well (Figure 59f). In each BFS iteration, the data-driven aproach has at most a few thousand GPU threads created, while the other approaches create as many threads as the number of vertices in the graph. And even though most of the created threads will be inactive they still execute some instructions and access memory to check whether they are active in the current iteration or not. In case of the data driven BFS it is unlikely that the GPU has full occupancy and it has the least power consumption.

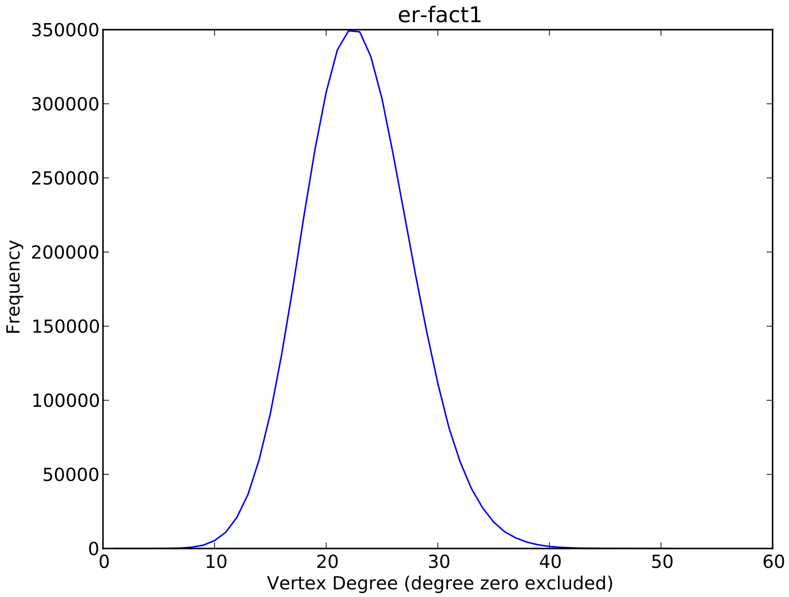
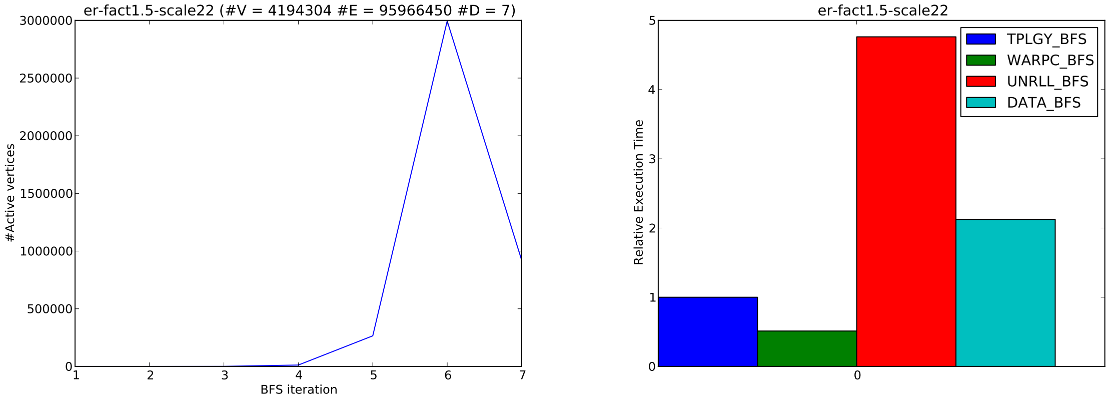


Figure 60: Frequency of Vertex Degrees for er-22

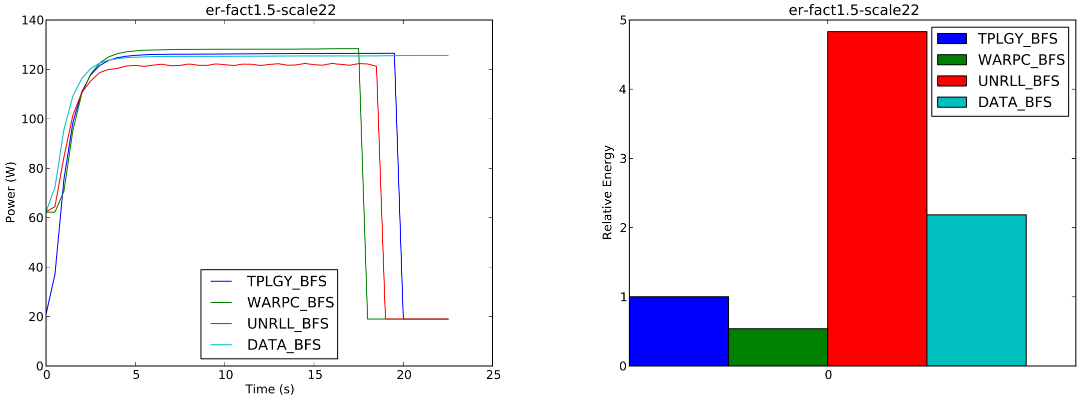
#### *5.4.1.4 Input er-fact1.5-scale22*

For the evaluated Erdos-Reyni graph most of the vertices have a degree larger than 10 (Figure 60) and the BFS seach terminates after only 7 iterations with most vertices being active in iterations 5-7 (Figure 60). Similar to the other graphs such as in-2004 and coPapersDBLP which have vertices with degree comparable or greater



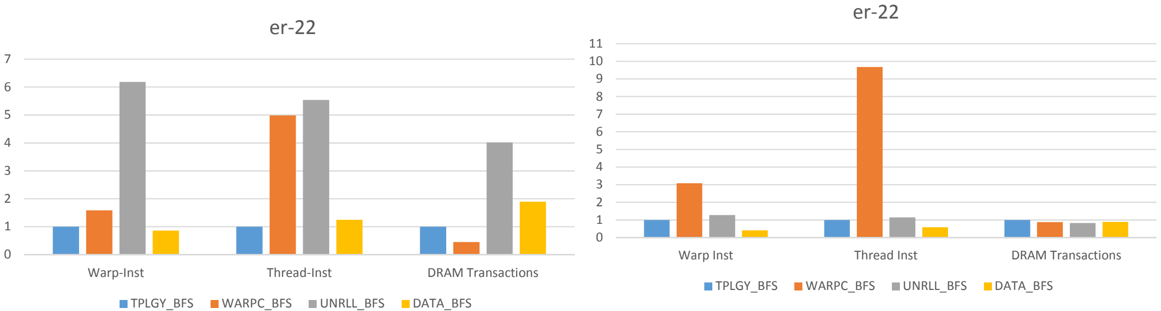
(a) Active vertices in each BFS step

(b) Relative execution time



(c) Power consumption

(d) Relative energy consumption time



(e) Instruction count and DRAM transcations

(f) Instruction count and DRAM transactions per cycle

Figure 61: BFS: Results for execution with er-22

than the warp width, even for er-fact1.5-scale22 the warp-centric approach shows the best performance (Figure 61b). The topology driven approach is the next best version, while the kernel unrolled version is the last. In terms of power consumption all the implementations consume the same amount of power (Figure 61c). While the instructions executed per cycle varies between the different implementations, the number of DRAM requests per cycle is about the same across all implementations (Figure 61f). This could be an indication that the power consumption is more affected by the number of memory requests rather than the number of instructions executed.

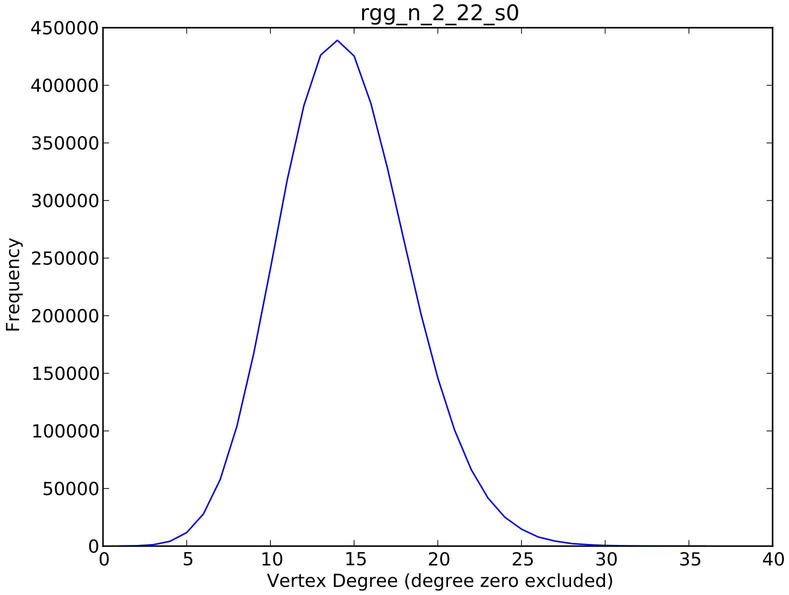
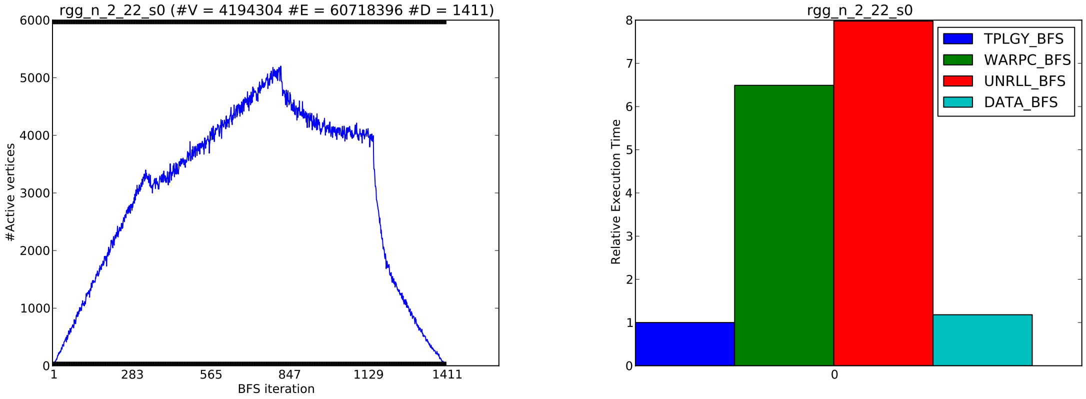


Figure 62: Frequency of Vertex Degrees for rgg-22

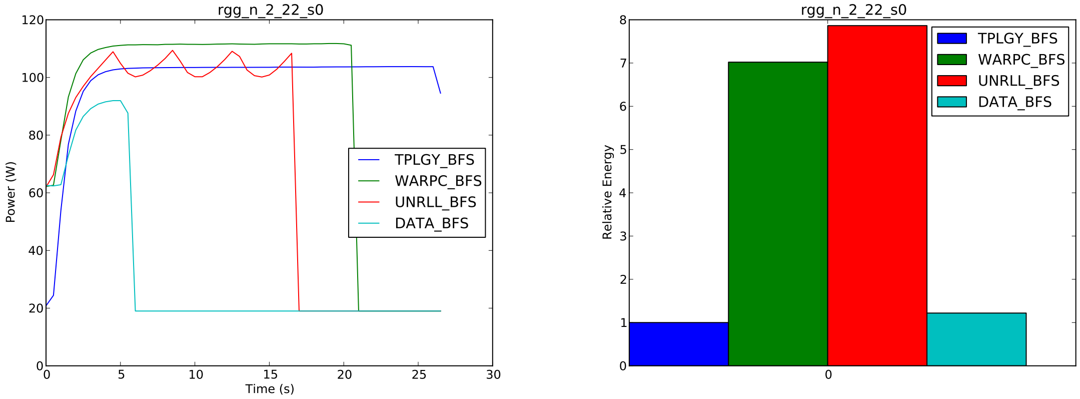
#### *5.4.1.5 Input rgg_n_2_22_s0*

In terms of node degree distribution, rgg_n_2_22_s0 is similar to er-fact1.5-scale22, but the node degrees are smaller in rgg_n_2_22_s0 (Figure 62). However, in case of rgg_n_2_22_s0, BFS takes around 1400 steps where as er-fact1.5-scale22 takes only 7 iterations (Figure 63b). In terms of performance and energy, the topology-driven BFS performs the best. Because of the small vertex frontier in each iteration, the data



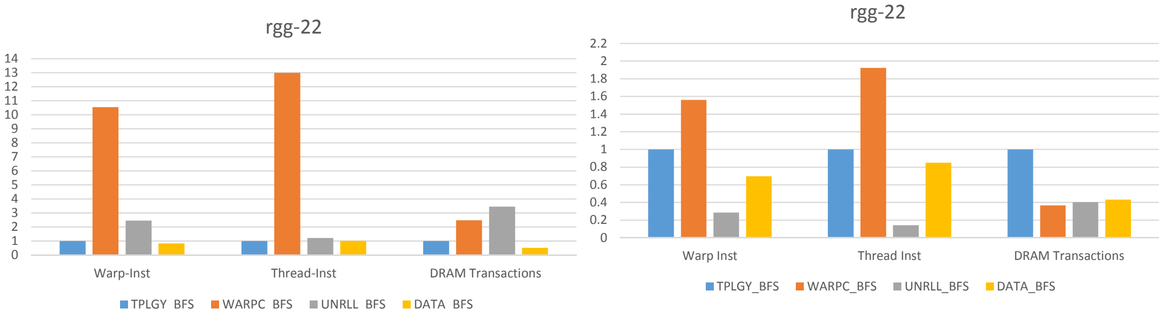
(a) Active vertices in each BFS step

(b) Relative execution time



(c) Power consumption

(d) Relative energy consumption time



(e) Instruction count and DRAM transcations

(f) Instruction count and DRAM transactions per cycle

Figure 63: BFS: Results for execution with rgg-22

driven approach also performs well, but it is slightly slower than the topology driven approach (Figure 63b). Though there are many vertices with degree comparable to the warp width, the warp-centric approach peforms much worse than the topology-driven and data-driven approaches primarily due to very few such vertices being active in each iteration and the sequential processing of vertices allocated to a warp. The data-drive approach consumes the smallest amount of power due to the small number of threads being created, while the other mechanisms consume similar amount of power (Figure 63c).

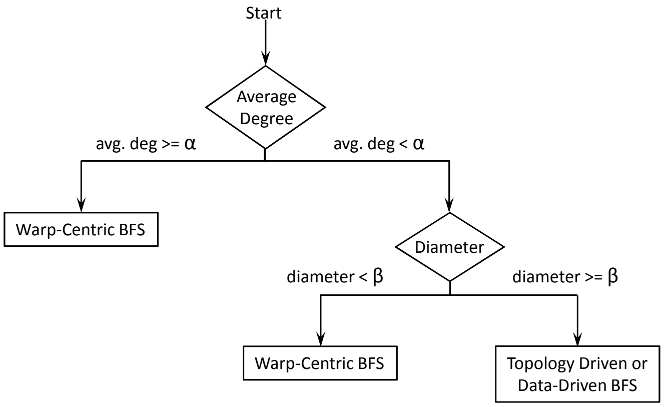
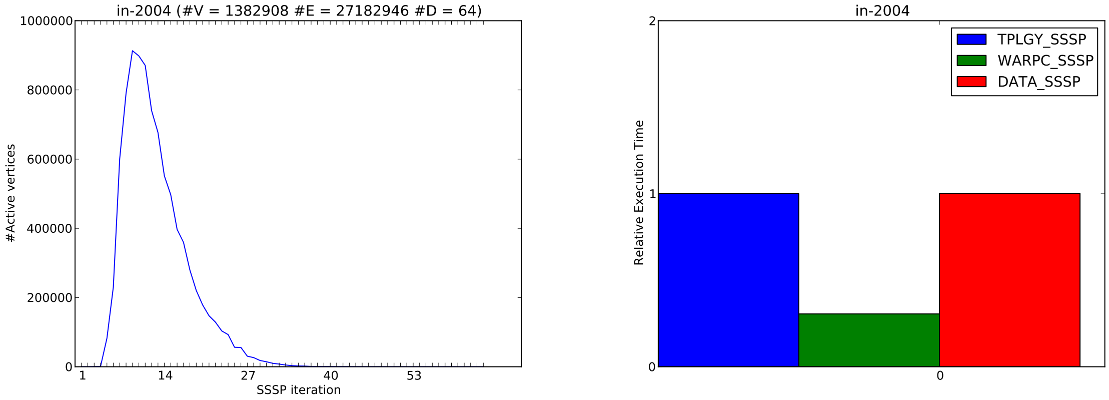


Figure 64: Predicting Best Performing BFS implementation

### 5.4.2 Predicting best BFS implementation

Predicting the best performing BFS implementation for an input graph requires more information in addition to the number of vertices and the degree of each vertex. However, the average vertex degree can be used in some cases to identify which implementation would be better for a given input graph. If the average vertex degree is large then warp-centric BFS is likely to be the best performing implementation as in the case of in-2004, coPapersDBLP and er-22. However, rgg-22 also has an average degree that could be considered large, but topology-driven BFS and data-driven BFS do better than warp-centric BFS for rgg-22. However, rgg-22 is likely to have a larger diameter than in-2004, coPapersDBLP and er-22 (1141 BFS iterations vs 26, 15,



(a) Active vertices in each SSSP step

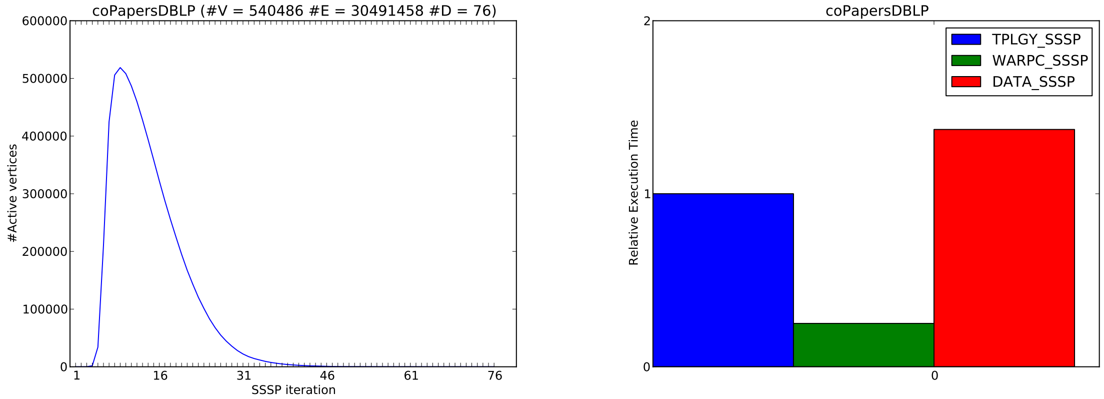
(b) Relative execution time

Figure 65: SSSP: Vertex frontier size and relative execution time for in-2004

and 7). If we know or can predict the diameter of a graph reasonably accurately, then we could determine which BFS implementation is likely to do well. Note that determining the diameter of a graph is more expensive than performing a BFS search, however, based on the source (origin) of the graph, or based on the results of a partial BFS search if we can predict the diameter of the graph then we may be able to predict the best performing BFS implementation for a given input graph. Figure 64 shows a simple chart that could be used for predicting the best BFS implementation for a given input graph. The values of $\alpha$ and $\beta$ have to be determined after measuring the performance for the different implementations for a large number of input graphs. Alternatively, the BFS implementation that is used could be dynamically changed for each iteration depending on the number of active vertices and their properties.

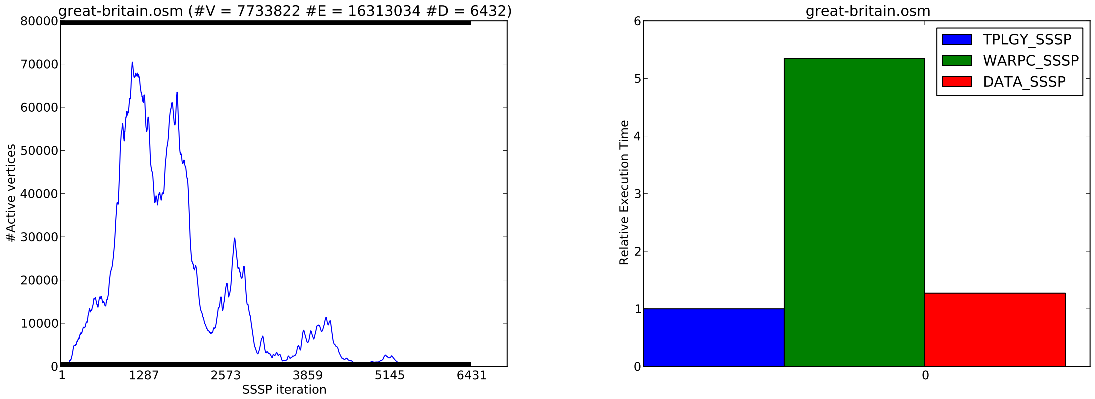
### 5.4.3 Evaluation of different SSSP implementations

Figures 65, 66, 67, 68 and 69 show the size of the vertex frontiers in each iteration of SSSP for different inputs and also the relative execution time, power consumption and the relative energy consumption for different versions of SSSP. For SSSP, we evaluate the topology, data-driven and warp centric approaches. Typically SSSP takes more iterations than BFS since shortest path in terms of number of hops from a source



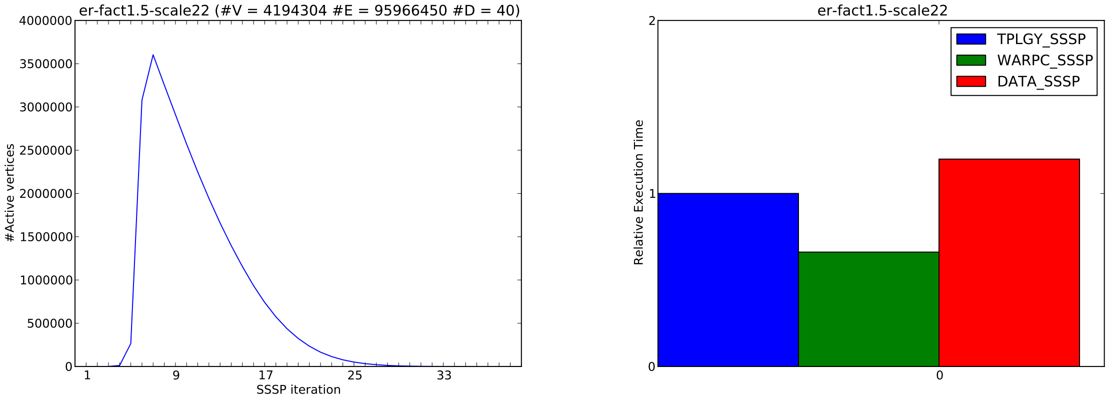
(a) Active vertices in each SSSP step (b) Relative execution time

Figure 66: SSSP: Vertex frontier size and relative execution time for coPapersDBLP



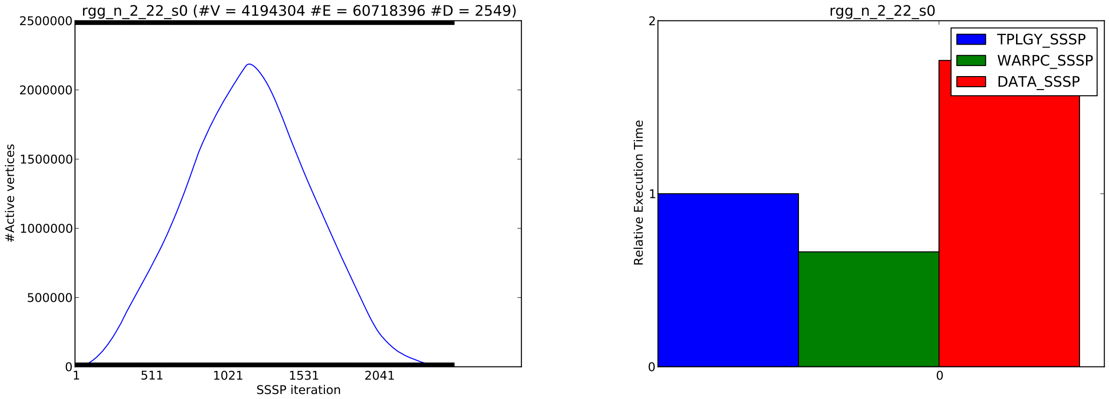
(a) Active vertices in each SSSP step (b) Relative execution time

Figure 67: SSSP: Vertex frontier size and relative execution time for great-britain



(a) Active vertices in each SSSP step (b) Relative execution time

Figure 68: SSSP: Vertex frontier size and relative execution time for er-22

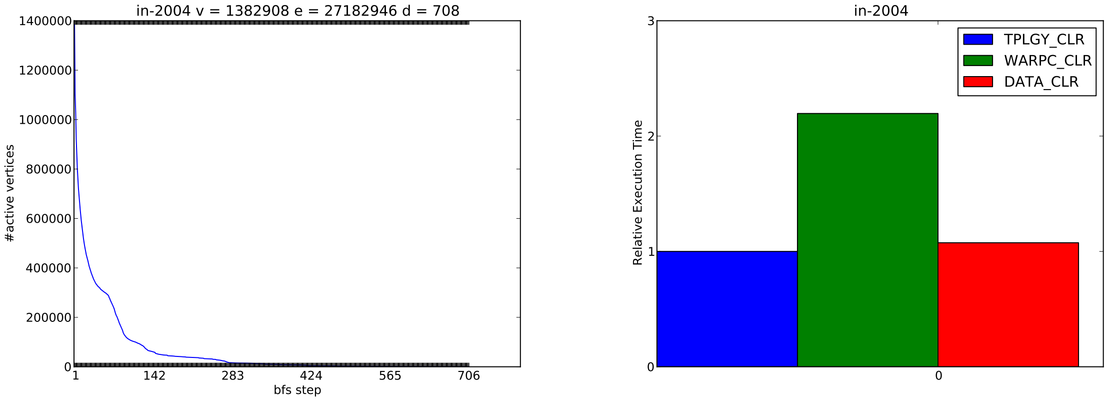


(a) Active vertices in each SSSP step

(b) Relative execution time

Figure 69: SSSP: Vertex frontier size and relative execution time for rgg-22

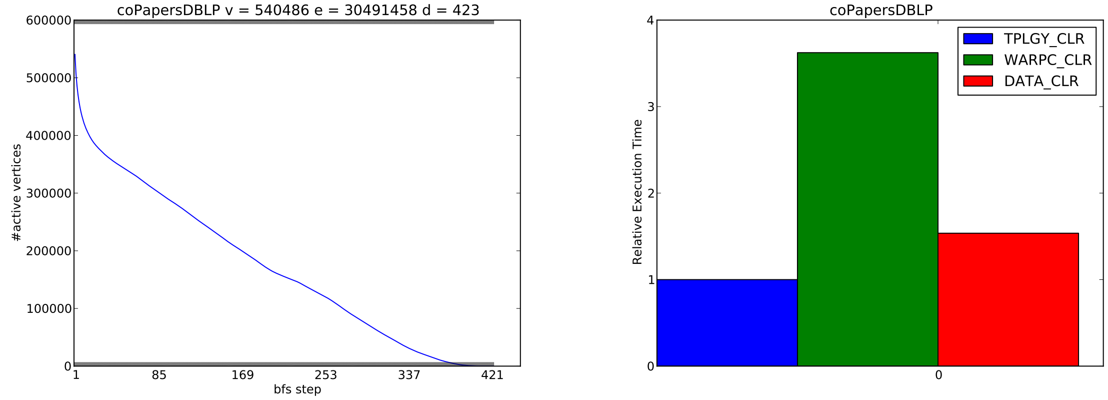
to a give vertex as found by BFS need not necessarily be the shortest path from the source to the given vertex. In case of BFS, the depth of a vertex is updated only once (in topology driven/data driven and warp centric approaches), while in case of SSSP the cost of a vertex from a source vertex can be updated multiple times once for each time a path with a lower cost is determined. And once the cost of node is updated in SSSP, the costs of its neighbors may also be updated. Table 16 shows that for all inputs except great-britain the number of iterations has alteast doubled. In terms of peformance, the warp centric approach does best for in-2004, coPapersDBLP, er-22, rgg-22 inputs. These graphs include vertices with large degree and such vertices are suitable for processing by the warp-centric approach. For great-britain, the topology driven approach gives the best performance, however the difference the between the topology driven approach and the data driven approach is not significant and that is the case for other inputs as well. SSSP consists of two kernels, one kernel updates the costs of vertices and the other kernel builds the vertex frontier for the next iteration. In case of topology driven SSSP, the kernel that updates the cost takes up most of the time, while in case of data-driven SSSP, the kernel that builds the vertex frontier takes up most of the time.



(a) Active vertices in each CLR step

(b) Relative execution time

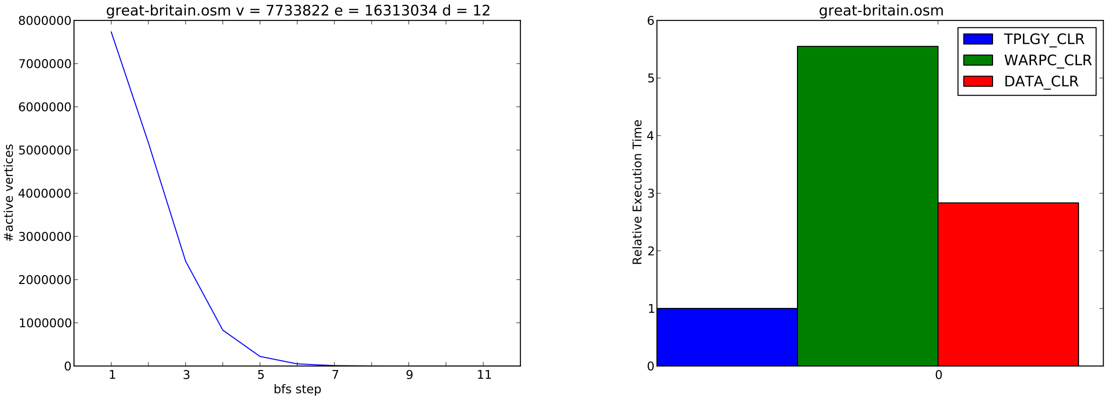
Figure 70: CLR: Vertex frontier size and relative execution time for in-2004



(a) Active vertices in each CLR step

(b) Relative execution time

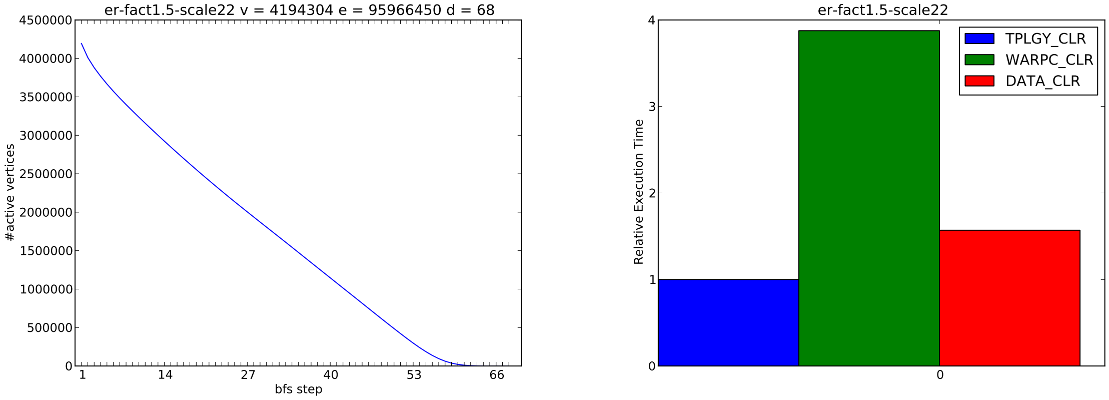
Figure 71: CLR: Vertex frontier size and relative execution time for coPapersDBLP



(a) Active vertices in each CLR step

(b) Relative execution time

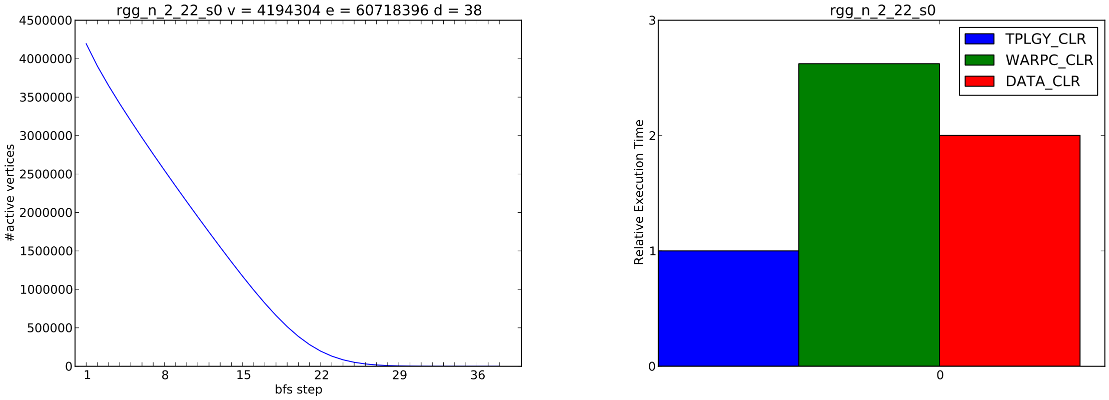
Figure 72: CLR: Vertex frontier size and relative execution time for great-britain



(a) Active vertices in each CLR step

(b) Relative execution time

Figure 73: CLR: Vertex frontier size and relative execution time for er-22



(a) Active vertices in each CLR step

(b) Relative execution time

Figure 74: CLR: Vertex frontier size and relative execution time for rgg-22

#### *5.4.3.1 Evaluation of different CLR implementations*

Figures 70, 71, 72, 73 and 74 show the size of the vertex frontiers in each iteration of CLR for different inputs and also the relative execution time, power consumption and the relative energy consumption for different versions of CLR. For CLR, we evaluate the topology, data-driven and warp-centric approaches. In case of CLR, typically the topology driven approach performs the best while the data-driven approach is comparable to the topology driven approach for all inputs. The warp-centric approach performs worse than the topology driven and data driven approaches for all inputs. In case of CLR, when a vertex is being processed all the neighbors of the vertex may not be processed, while in case of BFS and SSSP, all the neighbors of a vertex are always processed. Thus in case of BFS and SSSP, the warp-centric approach is suitable for processing vertices with large degree, while in case of CLR, a warp-centric approach may not always be suitable for processing large degree vertices because it is not guaranteed that all the neighbors of the vertex being processed will be accessed.

#### *5.4.3.2 Execution time of BFS, SSSP and Graph Coloring*

Table 17: Characterization: Execution Duration (ms) of BFS, SSSP and CLR

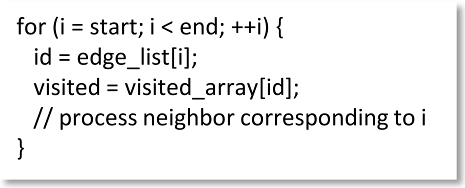
| | BFS | | SSSP | | CLR | |
|---|---|---|---|---|---|---|
| | TPLGY | WARPC | TPLGY | WARPC | TPLGY | WARPC |
| in-2004 | 117.28 | 40.76 | 346.72 | 105.84 | 3104.86 | 6817.73 |
| coPapersDBLP | 16.67 | 10.97 | 357.56 | 89.57 | 428.10 | 1551.16 |
| great-britain | 3090.12 | 27220.40 | 5616.61 | 30045.40 | 17.56 | 97.45 |
| er-fact1.5-scale22 | 151.58 | 77.54 | 1393.94 | 921.59 | 731.83 | 2836.51 |
| rgg_n_2_22_s0 | 512.62 | 3327.49 | 19760.71 | 13120.24 | 125.33 | 328.74 |

Table 17 shows the execution duration in ms for different implementations of BFS, SSSP and Graph Coloring (CLR) for multiple inputs. In BFS an vertex is active only once, however, in SSSP, a vertex can be active multiple times, thus SSSP always takes longer than BFS to complete. On the other hand, CLR can either take longer than BFS or can finish earlier than BFS. The relative execution time of BFS and CLR depends somewhat on the number of iterations that each algorithm takes to complete. For great-britain and rgg_2, BFS takes a large number of iterations compared to CLR and also takes a longer time to complete. The situation is reversed for the other inputs shown.

## 5.5 *Impact of software prefetching and loop unrolling*

In this section we examine the impact of different optimizations such as software prefetching and loop unrolling on the performance of graph algorithms. We evaluate these optimzations using a topology driven implementation of BFS.

### 5.5.1 Inserting software prefetches and Loop unrolling

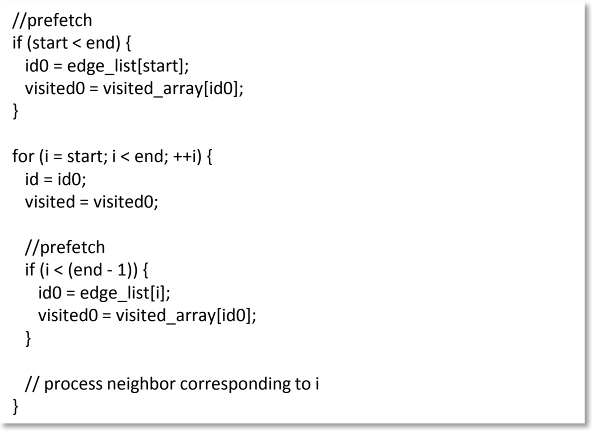


```
for (i = start; i < end; ++i) {
  id = edge_list[i];
  visited = visited_array[id];
  // process neighbor corresponding to i
}
```

Figure 75: Baseline BFS implementation

Let us examine how software prefetches can be inserted for the considered graph algorithms using the BFS as an example. The code snippet in Figure 75 shows the code executed by each thread processing an active vertex in the BFS kernel. As explained in the spare register aware prefetching mechanism, prefetching the neighbor load only would not be very beneficial, the attribute load would also have to be prefetched. Prefetches for both the neighbor load and the attribute load can be inserted as shown in Figure 76.

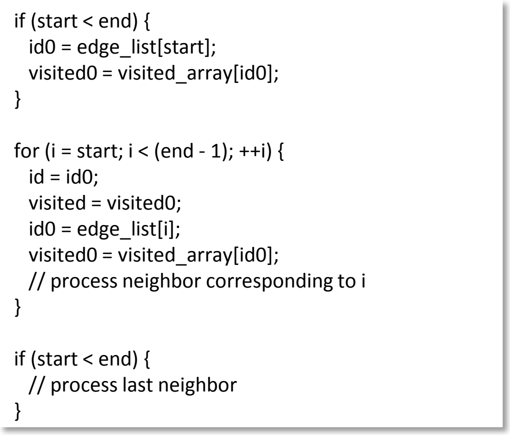
Figure 76 shows two checks - one check outside the loop and one check inside the loop - to ensure that elements outside the array bounds are not accessed since that could result in a memory access fault. With this version of software prefetching, all neighbors of a vertex are processed inside the for loop and no additional code is



```
//prefetch
if (start < end) {
  id0 = edge_list[start];
  visited0 = visited_array[id0];
}

for (i = start; i < end; ++i) {
  id = id0;
  visited = visited0;

  //prefetch
  if (i < (end - 1)) {
    id0 = edge_list[i];
    visited0 = visited_array[id0];
  }

  // process neighbor corresponding to i
}
```

Figure 76: BFS with software prefetches

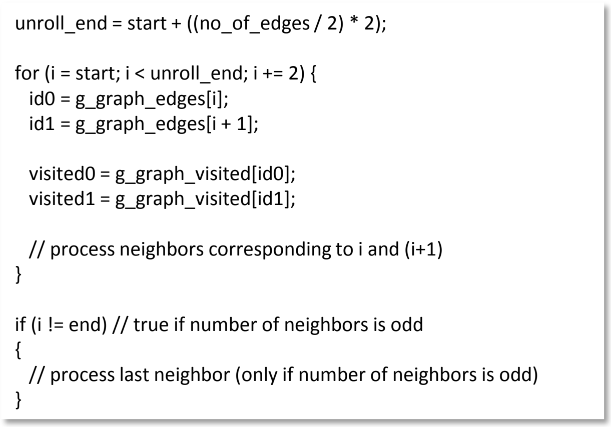
required outside the for loop except for prefetching the first neighbor. A downside of this approach is that the bounds check inside the for loop gets repeated every loop iteration and increases instruction count considerably.



```
if (start < end) {
  id0 = edge_list[start];
  visited0 = visited_array[id0];
}

for (i = start; i < (end - 1); ++i) {
  id = id0;
  visited = visited0;
  id0 = edge_list[i];
  visited0 = visited_array[id0];
  // process neighbor corresponding to i
}

if (start < end) {
  // process last neighbor
}
```

Figure 77: BFS with software prefetches without bound checks

The code snippet in Figure 77 shows an improved version of software prefetching. In this version the bounds check inside the for loop is eliminated by changing the end condition of the loop. The bounds check can be eliminated by reducing the number of loop iterations by the prefetch distance. However, this means that the last *prefetch_distance* elements have to be processed outside the for loop. This means code in the for loop is duplicated outside the loop as well.



```
unroll_end = start + ((no_of_edges / 2) * 2);

for (i = start; i < unroll_end; i += 2) {
  id0 = g_graph_edges[i];
  id1 = g_graph_edges[i + 1];

  visited0 = g_graph_visited[id0];
  visited1 = g_graph_visited[id1];

  // process neighbors corresponding to i and (i+1)
}

if (i != end) // true if number of neighbors is odd
{
  // process last neighbor (only if number of neighbors is odd)
}
```

Figure 78: BFS with loop unrolling

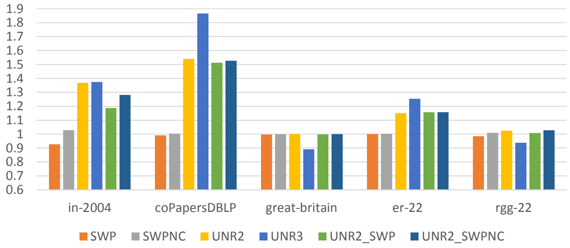
The code in Figure 78 shows the unrolling of the loop in BFS with a degree of two. In this case the number of iterations is halved with each iteration processing exactly two neighbors. If the number of neighbors is odd then the last neighbor is processed outside the for loop. Compared to software prefetching, unrolling increaseses code size and register usage to a larger extent.

### 5.5.2 Results

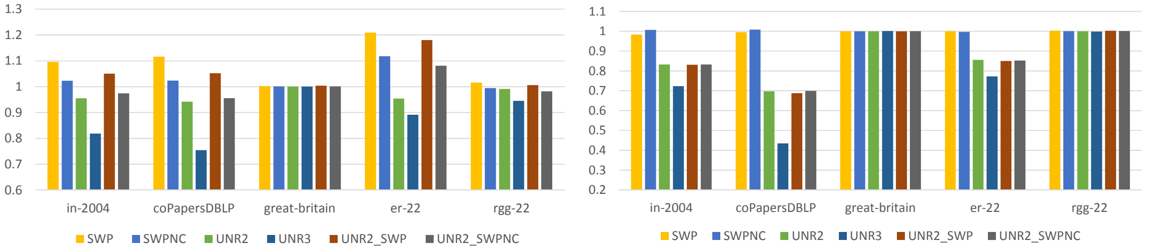
Table 18: Characterization: Register Usage and Occupancy with software optimizations

| | Topology Driven BFS | |
|---|---|---|
| | Reg | Occ |
| Base | 13 | 3 |
| SwPref (SWP) | 15 | 3 |
| SwPref - No check (SWPNC) | 16 | 3 |
| Unroll (Deg-2) (UNR2) | 19 | 3 |
| Unroll (Deg-3) (UNR3) | 24 | 2 |
| Unroll (Deg-2) + SwPref (UNR2_SWP) | 20 | 3 |
| Unroll (Deg-2) + SwPref - No check (UNR2_SWPNC) | 20 | 3 |

Table 18 shows the different versions of topology driven BFS that are evaluated along with the register usage per thread and the occupancy for each version. Software prefetching and unrolling increase the number of registers required for each GPU



(a) Relative performance with different software optimizations



(b) Total instruction count relative to baseline (c) Total DRAM transactions relative to baseline

Figure 79: Performance, instruction count and DRAM transactions relative to baseline

thread. All versions except UNR3 maintain the same occupancy as the baseline; UNR3 on the otherhand reduces occupancy to two thread blocks.

Figure 79a shows the performance of different versions of toplogy driven BFS relative to the baseline for multiple input graphs. While Figure 79b and Figure 79c show the total instruction count and the total DRAM transactions for the different versions relative to the baseline. Below we discuss the performance with the different versions of topology driven BFS.

**SWP and SWPNC**

SWP results in performance degradation for input in-2004 while not affecting the performance with other inputs. SWPNC eliminates the performance reduction seen with SWP for in-2004 while not affecting the performance with other inputs. Due to the bounds check within the loop, SWP increases instruction count significantly for in-2004, coPapersDBLP, er-22 and rgg-22 which are graphs that have nodes with large degree and are likely to spend a significant portion of the execution time within the loop. SWPNC which eliminates the bounds check from within the loop has a smaller instruction count than SWP. SWP and SWNPC have the same number of DRAM trasactions as the baseline.

**UNR2 and UNR3**

UNR2 and UNR3 not only reduce the instruction counts for in-2004, coPapers-DBLP and er-22 due to eliminate of predicate set and branch, but also reduce the number of DRAM requests due to issuing multiple loads to same cache block in the same iteration (consecutive edge list accesses often map to same cache block). Since the loads are to the same cache block and are issued close to each other, the loads following the first load in the iteration are likely to be secondary misses reducing the number of DRAM requests. Thus we see a performance benefit for these versions. However, in case of great-britain and rgg-22, there is no reduction in the number of DRAM requests, a small reduction in the number of instructions and we do not any performance improvements.

**UNR2_SWP and UNR2_SWPNC**

UNR2_SWP and UNR3_SWP have performance characteristics similar to UNR2 but have higher instruction counts due to software prefetching which reduces performance numbers slightly compared to UNR2.

## *5.6 Summary*

In Section 5.4.1 we measured the performance, power and energy consumption of multiple implementations of BFS with different inputs. The best peforming implementation depends on the input graph. For graphs that have nodes with large degree and small BFS tree heights the warp-centric approach seems to do better than others. in-2004, coPapersDBLP (Social Network) and er-22 (Erdos-Reyni graph) are examples of such graphs, they have nodes with large degree and have small BFS tree heights of 26, 15 and 7. On the other hand for graphs with vertices with small degrees and large BFS tree heights, the data-centric approach is better; great-britain (Street Network) is an example of such a graph. The topology driven approach is often comparable to the best performing implementation and is the best implementation for rgg-22 (Random geometric graph), a graph with large vertex degrees but large BFS tree height. Power consumption is often dependent on the number of DRAM transactions issued per cycle or on the occupancy of the GPU.

In Section 5.4.3 and Section 5.4.3.1 we repeated the experiments in Section 5.4.1 for different implementations of SSSP and CLR. The behavior of SSSP is similar to that of BFS, the warp-centric approach typically performs better for graphs with large degrees vertices. While the thread-centric approaches are better for graphs with small degree vertices. For CLR, the thread-centric topology driven approach is often the better than others since in CLR a vertex often does not process all its neighbors.

In Section 5.5.2 we examined the impact of software prefetching and loop unrolling on a thread-centric, topology-driven implementation. While software prefecthing did not show any performance improvements, loop unrolling showed performance benefits in case of graphs with large degrees due to a reduction in instruction count as well as a reduction in the number of DRAM transactions.

# CHAPTER VI

# RELATED WORK

This chapter discusses related work.

## *6.1 Graph algorithm implementations*

There has been a recent spate of work on the implementation of graph algorithms for GPUs. The algorithms that have been implemented include Breadth First Search [31, 32, 21, 51, 4], Minimum Spanning Tree [76, 32, 4], Single Source Shortest Path [31, 32, 4], All Pairs Shortest Path [31, 32], Max-Flow Min-Cut [32], S-T Connectivity [32], Bipartite Matching [75], Traveling Salesman Problem [57].

## *6.2 Related work on Prefetching*

### 6.2.1 Hardware GPU Prefetching

Not much work has been proposed for GPU specific hardware prefetching mechanisms. Lee et al. [44] propose Many-Thread Aware Prefetcher (MTAP) for GPGPU applications. MTAP uses a table based mechanism to detect strides in thread accesses and does both intra-warp and inter-warp prefetching. The prefetching mechanism proposed in this thesis uses a tag based approach for detecting target loads and their properties and does intra-thread prefetching only. MTAP assumes that all threads are active in every warp, which is not the case with graph algorithms. Also, in case of graph algorithms, MTAP would be unlikely to detect any inter-warp strides due to data dependent memory accesses. The Intra-warp component of MTAP may detect strided accesses by the head-loads, but prefetching only the head-loads would not be beneficial.

### 6.2.2 Software GPU Prefetching

Ryoo et al. [65] show that binding prefetches can improve performance, and also propose a set of metrics to predict the performance improvement with different program optimizations which are applied GPGPU benchmarks. Their work does not propose any prefetching mechanisms for GPGPUs.

Yan et al. [80] propose a compiler which does register based prefetching that is similar to the proposal in this thesis. While their approach is completely software based, the proposed mechanism (i.e., mechanism proposed in this thesis) can be entirely hardware-based. While the compiler-based prefetching mechanism has to use a fixed prefetch distance, the proposed mechanism uses a dynamic prefetch distance which can help improve performance. In addition, the compiler based mechanism requires additional registers to be available for all warps to be able to insert prefetches. The proposed mechanism can be modified to do prefetching for only a subset of the warps when insufficient spare registers are available by allocating spare registers to only some of the warps.

Using CPUs to prefetch for GPUs by doing pre-execution has also been proposed [81]. While this mechanism can potentially prefetch all memory accesses, it can work only on fused CPU-GPU architectures. On the other hand, the proposed mechanism prefetches only a subset of the accesses, but does not require the assistance of a CPU.

### 6.2.3 Pointer-based Prefetching

Several pointer-based hardware and software prefetching mechanisms [48, 63, 62, 18, 20, 24, 78] have been proposed for CPUs. Similar to these mechanisms, the proposed solution can also be considered to be doing pointer-based prefetched. In a load-pair, the head load points to the index to be accessed by the tail load. Some of these mechanisms augment the data structure being prefetched with additional fields (jump-pointers) while the proposed mechanism does not modify the prefetched array. In addition, these mechanisms place the prefetched data into cache and not registers and are likely to consume additional bandwidth.

Prefetcing via pre-execution by newly spawned helper threads [47, 66] has also been proposed to improve the performance of pointer intensive applications. They also suffer from early evictions. Furthermore, unlike the proposal in this thesis such mechanisms require compiler assistance, high overhead of creating helper threads, or re-execution of most applications. The proposed mechanism also results in execution of additional instructions but by targeting only the common graph algorithm patterns, the cost of additional instructions is much less.

A hardware-software cooperative prefetching scheme which uses hint bits generated by the compiler to regulate an aggressive hardware prefetching engine has also been proposed [77]. In a version of the proposal in this thesis the compiler identifies the target loads and their properties, instead of providing hints about different loads. Also, we propose a prefetching mechanism rather than a mechanism to reduce bandwidth consumption by prefetches.

### 6.2.4 Support for irregular memory accesses in GPUs

Not much research has been done for irregular applications in GPUs unlike CPU applications. Run-ahead of execution of threads with cache hits in a memory divergent warp has been proposed to issue (subsequent) memory requests ahead of time [71, 50]. These mechanisms are similar to the proposed mechanism in that memory requests are issued ahead of the time, however the proposed mechanism issues memory requests ahead in time for all threads in a warp and does not do run-ahead execution. These mechanisms require complex hardware to support divergence and to merge split warps as well.

### 6.2.5 Work related to register files in GPUs

Recently, there have been some proposals related to unallocated registers. Abdel-Majeed et al. proposed to reduce power consumption by turning off unallocated registers [7]. Gebhart et al. proposed several solutions to utilize register files such as using register file caches [28, 29] or modified register file hierarchies [29, 30]. On the otherhand, the proposal in this thesis utilizes unallocated registers to store prefetched data. Furthermore, the remaining unallocated registers can also be turned off to save power.

## *6.3 Related work on Caching*

### 6.3.1 Prior Bypass Mechanisms

#### *6.3.1.1 CPU Bypass*

Several Bypass mechanisms have been proposed for CPUs and they could be classified as PC-based, address-based, a combination of the two or mechanisms that are neither address based nor PC based.

**PC-based**

Tyson et al. [74] propose a mechanism in which a 2-bit counter is associated with each load instruction. The counter associated with a load is incremented/decremented whenever the load has a cache miss/hit. When the counter for a load reaches its maximum possible value, the load is marked Cacheable/Non-Allocatable (C/NA) meaning cache blocks for which the load has a miss will not be allocated in the cache. However the cache blocks can be brought into the cache by other load instructions. In GPUs most loads have a low hit ratio and thus the counter for each load reach its maximum value soon and all loads would be allocated in the cache, leading to performance degradation.

**Address-based**

Johnson et al. [40] propose a microarchitecture scheme which determines the placement of data within the cache hierarchy based on dynamic referencing behavior. A Memory Address Table (MAT) maintains the access patterns of macroblocks (each macroblock is a contiguous block of memory of a fixed size). The MAT has a saturating counter, with the counter value representing the frequency of accesses to the corresponding macroblock. On a memory access, the MAT entry for the corresponding macroblock is looked up along with the cache. In case of a cache hit the counter value is ignored, in case of a miss, the MAT counter value for the cache block selected for replacement is also looked up. If the counter of the missing block is below a certain fraction of the counter value of the replacement victim, then the missing block bypasses the cache. The mechanism by Johnson et al. favors loads that are accessed more frequently, not necessarily loads that are likely to provide cache hits. This mechanism is one of the mechanisms we evaluate in Section 4.5.

**Others**

Kharbutli et al. [42] propose counter based predictors that use both PC as well as the accessed addresses. Because of the large number of memory accesses by a PC in GPU workloads (same PC is executed by 100s of threads), the predictors require large prediction tables to be able to collect information about different cache blocks without significant aliasing.

#### *6.3.1.2 GPU Bypass*

Very few GPU-specific bypass mechanisms have been proposed so far. Jia et al. [38] characterize the performance of a set of GPU applications on caches and provide a taxonomy for memory access locality in GPGPU applications. They propose a compile-time algorithm for determining the locality type of each instruction and deciding whether the instruction should use the cache or not. Xie et al. [79] propose a compiler framework for bypassing on GPUs. The framework profiles applications multiple times and identifies loads which are likely to increase L2 traffic if they are cached in the L1, such loads are marked to bypass the L1 cache in the PTX code that is generated, while other loads are cached.

Jia et al. [37] propose MRPB, a memory request prioritization and cache bypass mechanism for GPUs. MRPB reorders memory requests in a core to reduce cache conflicts and bypasses cache acesses for requests that experience cache associativity stalls. MRPB assumes a allocate on miss cache which reserves a cache way for a miss when the miss occurs. An associativity stall occurs when all ways in a cache set are reserved, and none have been filled yet. Our mechanism assumes a allocate on fill cache, and associativity stalls do not happen in such caches.

Choi et al. [16] propose a write-buffering and read-bypassing mechanism for GPGPUs. Programmers annotate loads whose addresses are referred to by atmost one thread block during kernel execution. Accesses by such loads bypass the L2 but are inserted into the L1. Store instructions whose addresses are referred by only one thread block during successive kernel execution are buffered in the L2 cache. The goal of this mechanism is to reduce the amount of data that is read from the memory at the launch of the next kernel and thereby improve performance.

### 6.3.2 Cache Space Management

There has been considerable work on managing cache space for CMPs.

Victim Replication [84] tries to combine the benefits of both shared and private last level caches. Victim Replication assumes a shared last level cache (L2) with each core in a CMP containing a slice of the total L2 storage. L1 victims of a core are replicated in the local L2 slice. Hits to the local L2 slice have much smaller access latency than hits to a remote slice.

CMP-NuRAPID [15] uses a enlarged private tag array and shared data array. The enlarged tag array is used to track remote cache blocks shared by the local core with other cores. If remote cache blocks are reused by the local core then they are replicated and the replica is placed closer to the core. CMP-NuRAPID also does capacity stealing to bring frequently accessed private data blocks close to the core. CMP-NuRAPID does replication and movement of data to bring data accessed by a core close to the core.

Cooperative Caching [12] starts with a CMP that has a private last-level cache for each core in the CMP and try to achieve the capacity advantage of shared last level caches while retaining the latency benefit of private last level caches. Blocks evicted from a private L2 cache (last level cache) are spilled to a remote L2 cache which provides hits when the cache block is referred again.

### 6.3.3 Dynamic Granularity Memory Hierarchy

Sector Caches [10, 33, 64] were briefly discussed in Section 4.6.1, they were originally introduced to reduce the overhead of tags in on-chip caches but were discarded when they were found to be inefficient compared to set associative caches. Rothman et al. [64] evaluated the design of sector caches for uniprocessors and found that sector caches were not useful for first level caches, but could be useful for multilevel cache hierachies where the data for the second level was stored off chip but the tags were stored on-chip. Using a sector cache for the second level reduced the amount of storage required for the on-chip tags.

In earlier sector cache designs where each sector consisted of multiple cache lines, all the cache lines in a cache sector belonged to the same memory sector, this often resulted in wastage of cache space. In a decoupled cache sector [67], the cache lines in a cache sector can belong to multiple memory sectors, thereby making better use of cache space.

Kumar et al. [43] introduce Ameoba cache an adaptive cache design enabling the use of variable number of cache blocks, each with possibly different granularities. This is achieved by completely removing the tag array and treating the entire memory as a uniform morphable array between tags and data. Adjustments are performed based on the spatial locality in the application. They show that their technique reduces miss rates, miss bandwidth and also reduces energy requirements.

Yoon et al. [82] attempt to combine the best features of fine-grained and coarse-grained memory accesses. They present a technique which allows each page in virtual memory to have varying access granularities based on spatial locality through the use of sector caches and sub-ranked memory systems. Further, they show how incorporate their technique into memory access scheduling.

Qureshi et al. [59] also recognize the problems that can arise due to poor spatial locality and unused words in cache lines. They present a technique to filter unused words in order to allow storing a larger number of cache lines. Their filtering technique retains only active words discarding unused words. They propose a cache organization, distill cache, to utilize this newly available capacity.

Other related work such as SPP [14, 83] and LAMAR [61] are discussed and evaluated in Section 4.6.

# CHAPTER VII

# CONCLUSION

Graph algorithms are components of several important applications. As more and more applications are GPU enabled, efficient execution of graph algorithms on GPUs becomes critical. However, owing to some of their characteristics, graph algorithms have low execution efficiency on GPUs. Graph algorithm implementations have irregular control flow, non-uniform work distribution and irregular and data-dependent memory accesses. The large number of data-dependent memory accesses result in idle cycles. In this theis we present a graph algorithm characteristic aware prefetching mechanism and explore the design of the cache hierarchy to improve the memory access latency tolerance of graph algorithms. We also examine the impact of different implemenation strategies on graph algorithm performance, power and energy consumption.

- Chapter 3 introduces a spare register aware prefetching mechanism that prefetches a data dependent access pattern found commonly in graph algorithms. Prefetching in GPUs can be made more effective by prefetching data into the unused spare registers and reading from the spare registers when data is required. By using spare registers for holding prefetched data the mechanism eliminates the loss of prefetched data due to cache evictions.

- Chapter 4 explores the design of the cache hierarchy for graph algorithms. We examine the impact of inclusion property, cache bypassing and a fine-grained memory hierarchy. Varying the inclusion property from non-inclusion can show small performance benefits while increasing on-chip traffic. Increased on-chip traffic can reduce benefits for machine configurations with higher bandwidths.

Cache bypassing at both L1 and L2 caches is beneficial for graph algorithms and fine-grained memory accesses can improve performance significantly by reducing bandwidth requirements. Cache bypassing and fine-grained accesses can be combined to improve performance more than when the mechanisms are applied individually.

- Chapter 5 evaluates the impact of graph algorithm implementation strategy on performance, power and energy consumption. For algorithms such as Breadth First Search and Single Source Shortest Path which process all the neighbors of a vertex, a warp-centric approach which uses multiple threads for processing one vertex in parallel is best for graphs (eg. social networks) with vertices having large degree. For graphs having vertices with small degree (eg. street networks) a topology or a data driven approach which uses one thread for processing each vertex does better. For algorithms such as Graph Coloring which do not process all the neighbors of a vertex (depending on the graph), a thread-centric approach that uses one thread per vertex is best.

# REFERENCES

[1] "10th dimacs implementation challenge - graph partitioning and graph clustering." http://www.cc.gatech.edu/dimacs10/downloads.shtml.

[2] "CUDA Runtime API." http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[3] "HYNIX GDDR5 SGRAM." hynix.co.kr/inc/pdfDownload.jsp?path =/datasheet/pdf/graphics/ H5GQ1H24AFR(Rev1.0).pdf.

[4] "LonestarGPU." `http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu/`.

[5] "MacSim." code.google.com/p/macsim/.

[6] "NVIDIA Management Library (NVML)." https://developer.nvidia.com/nvidia-management-library-nvml.

[7] Abdel-Majeed, M. and Annavaram, M., "Warped register file: A power efficient register file for gpgpus," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture*, HPCA '13, 2013.

[8] Brunie, N., Collange, S., and Diamos, G., "Simultaneous branch and warp interweaving for sustained gpu performance," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[9] Burtscher, M., Nasre, R., and Pingali, K., "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 141–151, 2012.

[10] Case, R. P. and Padegs, A., "Architecture of the ibm system/370," *Commun. ACM*, vol. 21, pp. 73–96, Jan. 1978.

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

[12] Chang, J. and Sohi, G., "Cooperative caching for chip multiprocessors," in *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pp. 264–276, 2006.

[13] Che, S., Beckmann, B. M., Reinhardt, S. K., and Skadron, K., "Pannotia: Understanding irregular gpgpu graph applications," 2013.

[14] Chen, C. F., Yang, S.-H., Falsafi, B., and Moshovos, A., "Accurate and complexity-effective spatial pattern prediction," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, (Washington, DC, USA), pp. 276–, IEEE Computer Society, 2004.

[15] Chishti, Z., Powell, M. D., and Vijaykumar, T. N., "Optimizing replication, communication, and capacity allocation in cmps," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pp. 357–368, 2005.

[16] Choi, H., Ahn, J., and Sung, W., "Reducing off-chip memory traffic by selective cache management scheme in gpgpus," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pp. 110–119, 2012.

[17] Cohen, J. and Castonguay, P., "Efficient graph matching and coloring on the gpu." developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0332-GTC2012-Graph-Coloring-GPU.pdf.

[18] Collins, J. D., Sair, S., Calder, B., and Tullsen, D. M., "Pointer cache assisted prefetching," in *MICRO-35*, (Los Alamitos, CA, USA), pp. 62–73, IEEE Computer Society Press, 2002.

[19] Cooksey, R., *Content-Sensitive Data Prefetching.* PhD thesis, University of Colorado, Boulder, 2002.

[20] Cooksey, R., Jourdan, S., and Grunwald, D., "A stateless, content-directed data prefetching mechanism," in *Proc. of the 10th Int'l. conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 279–290, ACM, 2002.

[21] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S., "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 63–74, 2010.

[22] Diamos, G., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., and Yalamanchili, S., "Simd re-convergence at thread frontiers," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, 2011.

[23] Diamos, G., Kerr, A., Yalamanchili, S., and Clark, N., "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *PACT-19*, 2010.

[24] Ebrahimi, E., Mutlu, O., and Patt, Y. N., "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems.," in *HPCA-15*, (Washington, DC, USA), pp. 7–17, IEEE Computer Society, 2009.

[25] Fung, W. W. L. and Aamodt, T. M., "Thread block compaction for efficient simt control flow," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[26] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M., "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO*, 2007.

[27] Gao, H. and Wilkerson, C., "A dueling segmented lru replacement algorithm with adaptive bypassing," in *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, JWAC-1, 2010.

[28] Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., and Skadron, K., "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, 2011.

[29] Gebhart, M., Keckler, S. W., and Dally, W. J., "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, 2011.

[30] Gebhart, M., Keckler, S. W., Khailany, B., Krashinsky, R., and Dally, W. J., "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, 2012.

[31] Harish, P. and Narayanan, P. J., "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th international conference on High performance computing*, HiPC'07, 2007.

[32] Harish, P., Vineet, V., and Narayanan, P. J., "Large graph algorithms for massively multithreaded architectures," Tech. Rep. IIIT/TR/2009/74, International Institute of Information Technology, Hyderabad, INDIA, 2009.

[33] Hill, M. D. and Smith, A. J., "Experimental evaluation of on-chip microprocessor cache memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, (New York, NY, USA), pp. 158–166, ACM, 1984.

[34] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K., "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, 2011.

[35] Hong, S., Oguntebi, T., and Olukotun, K., "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 78–88, 2011.

[36] Iacobovici, S., Spracklen, L., Kadambi, S., Chou, Y., and Abraham, S. G., "Effective stream-based and execution-based data prefetching.," in *ICS-18*, pp. 1–11, 2004.

[37] Jia, W., Shaw, K., and Martonosi, M., "Mrpb: Memory request prioritization for massively parallel processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 272–283, Feb 2014.

[38] Jia, W., Shaw, K. A., and Martonosi, M., "Characterizing and improving the use of demand-fetched caches in gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, 2012.

[39] J.Nesbit, K. and E.Smith, J., "Data cache prefetching using a global history buffer," in *HPCA-10*, (Washington, DC, USA), pp. 96–105, IEEE Computer Society, 2004.

[40] Johnson, T. L. and mei W. Hwu, W., "Run-time adaptive cache hierarchy management via reference analysis.," in *ISCA-19*, pp. 315–326, 1997.

[41] Joseph, D. and Grunwald, D., "Prefetching using Markov predictors," in *ISCA-19*, (New York, NY, USA), pp. 252–263, ACM, 1997.

[42] Kharbutli, M. and Solihin, Y., "Counter-based cache replacement and bypassing algorithms.," *IEEE Trans. Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[43] Kumar, S., Zhao, H., Shriraman, A., Matthews, E., Dwarkadas, S., and Shannon, L., "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 376–388, IEEE Computer Society, 2012.

[44] Lee, J., Lakshminarayana, N. B., Kim, H., and Vuduc, R., "Many-thread aware prefetching for gpgpus," in *MICRO-43*, 2010.

[45] Li, L., Tong, D., Xie, Z., Lu, J., and Cheng, X., "Optimal bypass monitor for high performance last-level caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 315–324, 2012.

[46] Luby, M., "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, 1985.

[47] Luk, C.-K., "Tolerating memory latency through software-Controlled pre-execution in simultaneous multithreading processors," in *ISCA-23*, (New York, NY, USA), pp. 40–51, ACM, 2001.

[48] Luk, C.-K. and Mowry, T. C., “Compiler-based prefetching for recursive data structures,” in *Proc. of the 7th Int'l. conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 222–233, ACM, 1996.

[49] Luo, L., Wong, M., and Hwu, W.-m., “An effective gpu implementation of breadth-first search,” in *Proceedings of the 47th Design Automation Conference*, DAC '10, 2010.

[50] Meng, J., Tarjan, D., and Skadron, K., “Dynamic warp subdivision for integrated branch and memory divergence tolerance.,” in *ISCA-32*, 2010.

[51] Merrill, D., Garland, M., and Grimshaw, A., “Scalable gpu graph traversal,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, 2012.

[52] Nasre, R., Burtscher, M., and Pingali, K., “Data-driven versus topology-driven irregular computations on gpus,” in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 463–474, May 2013.

[53] Nesbit, K. J., Dhodapkar, A. S., and Smith, J. E., “AC/DC: An adaptive data cache prefetcher,” in *PACT-13*, (Washington, DC, USA), pp. 135–145, IEEE Computer Society, 2004.

[54] NVIDIA, “Fermi: Nvidia's next generation cuda compute architecture.” http://www.nvidia.com/fermi.

[55] NVIDIA, “Nvidia's next generation cuda compute architecture: Kepler gk110.” http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[56] NVIDIA Corporation, *CUDA Programming Guide*.

[57] O'Neil, M., Tamir, D., and Burtscher, M., “A parallel gpu version of the traveling salesman problem,” in *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'11, 2012.

[58] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., hsien Lee, T., Lenharth, A., Manevich, R., Mndez-lojo, M., Prountzos, D., and Sui, X., “The tao of parallelism in algorithms,” in *In PLDI*, pp. 12–25, 2011.

[59] Qureshi, M., Suleman, M., and Patt, Y., “Line distillation: Increasing cache capacity by filtering unused words in cache lines,” in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 250–259, Feb 2007.

[60] Rhu, M. and Erez, M., "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, 2012.

[61] Rhu, M., Sullivan, M., Leng, J., and Erez, M., "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 86–98, 2013.

[62] Roth, A., Moshovos, A., and Sohi, G. S., "Dependence based prefetching for linked data structures," in *Proc. of the 8th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[63] Roth, A. and Sohi, G. S., "Effective jump-pointer prefetching for linked data structures," in *ISCA-21*, (Washington, DC, USA), pp. 111–121, IEEE Computer Society, 1999.

[64] Rothman, J. and Smith, A., "Sector cache design and performance," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pp. 124–133, 2000.

[65] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S.-Z., Stratton, J. A., and mei W. Hwu, W., "Program optimization space pruning for a multithreaded gpu," in *CGO-6*, pp. 195–204, 2008.

[66] Saavedra, R. H. and Park, D., "Improving the effectiveness of software prefetching with adaptive execution," in *Proc. of the 5th Int'l. Conf. on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 68–78, IEEE Computer Society, 1996.

[67] Seznec, A., "Decoupled sectored caches: Conciliating low tag implementation cost," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, (Los Alamitos, CA, USA), pp. 384–393, IEEE Computer Society Press, 1994.

[68] Sim, J., Lee, J., Qureshi, M. K., and Kim, H., "Flexclusion: Balancing cache capacity and on-chip bandwidth via flexible exclusion," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[69] Sinharoy, B., Kalla, R. N., Tendler, J. M., Eickemeyer, R. J., and Joyner, J. B., "Power5 system microarchitecture.," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 505–522, 2005.

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

[71] Tarjan, D., Meng, J., and Skadron, K., "Increasing memory miss tolerance for simd cores.," in *SC*, 2009.

[72] Tendler, J., Dodson, S., Fields, S., Le, H., and Sinharoy, B., "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[73] The IMPACT Research Group, UIUC, "Parboil benchmark suite." http://impact.crhc.illinois.edu/parboil.php.

[74] Tyson, G., Farrens, M., Matthews, J., and Pleszkun, A., "A new approach to cache management," in *MICRO-28*, 1995.

[75] Vasconcelos, C. N., Rosenhahn, B., and Hannover, L. U., "Bipartite graph matching computation on gpu," in *in Proc. Intl. Conference Energy Minimization Methods in Computer Vision and Pattern Recognition*, pp. 42–55, 2009.

[76] Vineet, V., Harish, P., Patidar, S., and Narayanan, P. J., "Fast minimum spanning tree for large graphs on the gpu," in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, 2009.

[77] Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C., "Guided region prefetching: a cooperative hardware/software approach," in *ISCA-25*, (New York, NY, USA), pp. 388–398, ACM, 2003.

[78] Wu, Y., "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation*, (New York, NY, USA), pp. 210–221, ACM, 2002.

[79] Xie, X., Liang, Y., Sun, G., and Chen, D., "An efficient compiler framework for cache bypassing on gpus," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pp. 516–523, Nov 2013.

[80] Yang, Y., Xiang, P., Kong, J., and Zhou, H., "A gpgpu compiler for memory optimization and parallelism management," in *Proc. of the ACM SIGPLAN 2010 Conf. on Programming Language Design and Implementation*, 2010.

[81] Yang, Y., Xiang, P., Mantor, M., and Zhou, H., "Cpu-assisted gpgpu on fused cpu-gpu architectures," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, 2012.

[82] Yoon, D. H., Jeong, M. K., and Erez, M., "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 295–306, ACM, 2011.

[83] Yoon, D. H., Jeong, M. K., Sullivan, M., and Erez, M., "The dynamic granularity memory system," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 548–559, IEEE Computer Society, 2012.

[84] Zhang, M. and Asanovic, K., "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pp. 336–345, 2005.