Adaptive GPU Cache Bypassing

Yingying Tian[‡][*] Sooraj Puthoor[†] Joseph L. Greathouse[†]
Bradford M. Beckmann[§] Daniel A. Jiménez[‡]

[‡]Dept. of Computer Science and Engineering
Texas A&M University
College Station, TX, USA
{tian, djimenez}@cse.tamu.edu

[†§]AMD Research
Advanced Micro Devices, Inc.
[†]Austin, TX, USA [§]Bellevue, WA, USA
{Sooraj.Puthoor, Joseph.Greathouse, Brad.Beckmann}@amd.com

ABSTRACT

Modern graphics processing units (GPUs) include hardware-controlled caches to reduce bandwidth requirements and energy consumption. However, current GPU cache hierarchies are inefficient for general purpose GPU (GPGPU) computing. GPGPU workloads tend to include data structures that would not fit in any reasonably sized caches, leading to very low cache hit rates. This problem is exacerbated by the design of current GPUs, which share small caches between many threads. Caching these streaming data structures needlessly burns power while evicting data that may otherwise fit into the cache.

We propose a GPU cache management technique to improve the efficiency of small GPU caches while further reducing their power consumption. It adaptively bypasses the GPU cache for blocks that are unlikely to be referenced again before being evicted. This technique saves energy by avoiding needless insertions and evictions while avoiding cache pollution, resulting in better performance. We show that, with a 16KB L1 data cache, dynamic bypassing achieves similar performance to a double-sized L1 cache while reducing energy consumption by 25% and power by 18%.

The technique is especially interesting for programs that do not use programmer-managed scratchpad memories. We give a case study to demonstrate the inefficiency of current GPU caches compared to programmer-managed scratchpad memories and show the extent to which cache bypassing can make up for the potential performance loss where the effort to program scratchpad memories is impractical.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Cache memories

General Terms

Microarchitecture, GPU, cache management

[*]Work performed while interning at AMD Research.

GPGPU'15 , February 2, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3407-5/15/02 ...$15.00.

Keywords

Bypassing, graphics processing unit cache, prediction.

1. INTRODUCTION

By densely packing many parallel arithmetic logic units together and clocking them at a moderate rate, graphics processing units (GPUs) have a much higher throughput than traditional CPUs of similar size and power envelope [38,44]. The last decade has seen growth in general purpose GPU (GPGPU) programming, where these graphics processors are used to perform highly parallel computations on traditional computational problems. Because of the large performance increases attainable with these processors, GPGPU programming has evolved into a popular way to accelerate highly parallel and computationally intensive algorithms [19]. As part of this move towards more general-purpose architectures, recent GPU designs have included deep hardware-controlled cache hierarchies to ease the burden of writing efficient GPGPU algorithms [1,39,40].

Replicating CPU cache management policies in GPU caches leads to performance and power inefficiencies. Unlike CPUs, GPUs run thousands of concurrent threads, greatly reducing the per-thread cache capacity. Moreover, many GPGPU workloads include large data structures that do not fit into any reasonably sized caches. These streaming accesses replace many other useful values, such that even frequently accessed data may be evicted before being referenced again. Placing this streaming data into a traditional cache hierarchy needlessly costs energy and yields no performance benefit.

A naïve solution is to add more storage to the cache hierarchy, which is inefficient for GPUs, as the die area spent on these caches could instead be dedicated to more parallel computation resources, increasing peak throughput. A good GPU cache management technique should thus strive to make small caches highly efficient for GPGPU workloads. They should yield a high hit rate for reused values while avoiding the energy used to store values that are not reused.

This paper presents dynamic hardware mechanisms that reduce the need for explicitly caching all data in GPU caches. We propose a GPU cache management technique that enhances the L1 data caches in a modern GPU by improving cache efficiency and reducing energy consumption. Our technique uses low-overhead dynamic bypass prediction to prevent streaming one-time-use values from being needlessly cached. If it predicts that a block will be reused, the data are placed into the cache hierarchy as normal. If a block is unlikely to be reused, it is sent directly to the compute units without being placed into the cache. Bypassing saves energy by avoiding storing values into the cache, only to later evict them after never accessing them again. Moreover, by inserting fewer useless blocks, the bypass mechanism allows useful data to reside in the cache longer, increasing the cache hit rate and improving performance.

We show that, over 13 GPGPU benchmarks, a 16KB L1 cache that uses our bypass predictor increases performance by up to 13% and slightly outperforms a 32KB L1 cache that does not bypass. Furthermore, our bypass predictor reduces L1 cache energy consumption by 25%, while requiring less than 256 bytes of extra storage in each private L1 cache and 0.5KB of extra storage in the 256KB shared L2 cache. Rather than doubling the size of the caches to improve hit rate, our technique keeps the caches small, allowing the saved area to be used for additional compute units.

This paper makes the following contributions:

- We propose a simple but effective GPU cache management technique. It prevents streaming one-time-use values from being needlessly inserted into the cache with high accuracy and minimal area overhead.
- We demonstrate performance gains and energy savings when using our bypass predictor for a GPU L1 data cache.
- We study the limitations of current GPU cache designs and the effects of a bypass predictor as they relate to using scratchpad memories. In particular, we compare an application that uses scratchpad memories to a rewritten version of the same application that does not require the complexity of manual memory layout in the context of our optimization.

The organization of this paper is as follows: Section 2 introduces the background of GPU computing and motivates the proposed technique. Section 3 describes the bypass predictor in detail, and Section 4 discusses the experiments used to evaluate our design. We explore experimental results in Section 5 and discuss related work in Section 6. Finally, Section 7 concludes and discusses future work.

2. BACKGROUND AND MOTIVATION

A GPU is a highly parallel processor consisting of hundreds to thousands of concurrently operating logic units. Though they were originally hard-coded circuits meant only to accelerate 3D graphics computations, modern GPUs are now fully programmable general-purpose processors. General purpose GPU computing (GPGPU) uses GPUs to accelerate applications in domains such as science, engineering, physics, media, and statistics [19].

2.1 GPUs and GPGPU Computing

Because GPUs were originally fixed-function circuits, programming them to yield useful general-purpose results was a laborious process that involved mapping the computational kernel onto the graphical equations that the GPU could perform [5, 11, 12]. As GPUs became more programmable, languages such as OpenCL^{TM} [17] and CUDA [41] have emerged to allow C-like programming of these accelerators [17, 37]. Among many microarchitectural details that programmers must contend with to attain high GPU performance, this paper focuses on the GPU memory system.

GPUs hide long memory access latencies through a high degree of thread-level parallelism. If one group of threads is stalled on a long latency memory request, many others can take that opportunity to execute. This is acceptable for most graphics workloads, but some GPGPU workloads can cause the whole pipeline to stall by causing all available thread groups to wait on memory. In addition, both graphics and general-purpose applications can heavily tax the memory bandwidth of a GPU. As such, GPUs traditionally used small read-only texture caches and scratchpad memories in order to increase available bandwidth to their computational pipelines. However, these resources are difficult to use for GPGPU workloads because they require either the programmer or compiler to decide whether particular memory accesses should go through these subsystems.

Modern GPU architectures have adopted hardware-controlled cache hierarchies between globally accessible DRAM and the compute units to aid programs that are unable to use the GPU's shared memory [39]. For example, AMD's Graphics Core Next (GCN) architecture has a 16KB private L1 cache for each compute unit (CU) and 64-128KB of shared L2 cache per memory channel [1]. Nvidia's Fermi architecture has a 16KB/48KB configurable private L1 cache for each streaming multiprocessor and up to 768KB of shared L2 cache [39]. The Heterogeneous System Architecture (HSA) Foundation has announced a roadmap that includes fully coherent cache memories across CPUs and GPUs [29].

Hardware-managed GPU caches are used for two main purposes: 1) to cache data with immediate spatial and temporal locality, and 2) as write-combining buffers to reduce the memory bandwidth and energy requirements of the system. Although caches are effective write-combining buffers for GPGPU workloads, they are less useful at exploiting locality [24]. The underlying reason for this is the streaming nature of GPGPU memory accesses resulting in good spatial locality but very low temporal locality.

2.2 GPGPU Memory Characteristics

Traditional graphics workloads traverse large scenes of 3D vertices while calculating shading values, performing mathematical transformations, and laying textures on surfaces. These algorithms stream large amounts of data from memory, consuming hundreds of megabytes to render a single frame. Because such large working sets are completely impractical to hold in on-chip caches, GPUs have traditionally had copious memory bandwidth and enough parallelism to keep these long latency accesses from stalling.

This bandwidth and latency hiding has subsequently affected the kinds of general-purpose applications that are commonly ported to run on GPUs. GPGPU applications often look like graphics workloads: highly parallel, regular, and with large storage and bandwidth needs. Although these workloads may exhibit good data reuse, the distance between repeated accesses to the same value is such that most of the reusable data are evicted from the cache before it can be touched again.

Figure 1 demonstrates this idea across a series of benchmarks from the Rodinia suite [6] and a selection of AMD APP SDK programs. The *zero-reuse* bars represent the percent of cache blocks that are evicted from a 16KB L1 cache before they are touched again. The data show that an average of 46% (and a maximum of 84%) of cache blocks are evicted by the pseudo-LRU replacement algorithm without

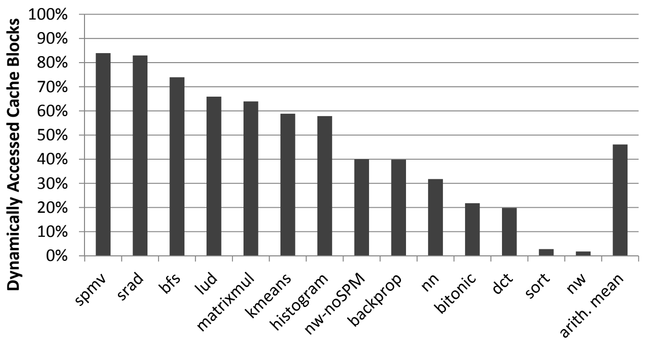

Figure 1: Zero-reuse blocks in the L1 data cache.

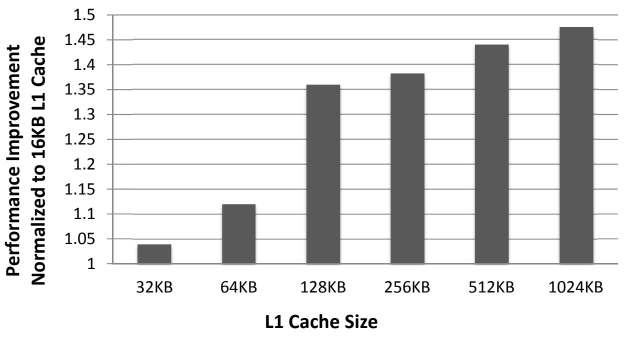

Figure 2: Performance improvement across different L2 cache sizes normalized to a 16KB L1 cache.

being touched again. Inserting the data into the cache costs energy, but only results in pollution and the potential eviction of other useful blocks.

Streaming data accesses in these programs, coupled with large data sets, are the primary reasons for these long reuse distances. For graphics applications, GPUs traditionally used different memory subsystems for data that would cache well (such as textures), allowing other data to bypass these specialized caches. Similarly, scratchpad memories (called Local Data Stores on AMD GPUs [1] and Shared Memory on Nvidia GPUs [39, 40]) can be used to manually store reusable data while skipping streaming values. Some GPUs now include compiler hints to say that particular static loads are streaming and so should not be cached [3, 25, 28].

As GPGPUs extend further into non-traditional domains, more programmers whose expertise lies outside GPU architectures are using these devices. Such explicitly managed memory systems are known to be more difficult to use than hardware-controlled caches [33], so requiring such structures limits the market for GPUs to only expert programmers. Moreover, scratchpad memories are not always portable across devices or generations of designs. Because scratchpad sizes and layouts change over time, further increasing the programmer's burden. With these issues in mind, this paper focuses on hardware mechanisms that can improve existing GPU caches and be transparent to software and programmers.

2.3 Improving GPU Caches

We previously identified two major problems with GPU caches: 1) They are not effective at exploiting temporal locality due to noise from streaming data; and 2) insertions and evictions of useless data consume energy without increasing performance.

Figure 2 shows the average performance improvement of different L1 data cache sizes normalized to a 16KB baseline over a series of GPGPU benchmarks described in Section 4.2. This demonstrates that more powerful caching systems have the capability to increase the GPU's performance. However, L1 caches larger than 16-64KB are impractical for current GPU designs.

As described in Section 2.1, current AMD GPUs have 16KB of L1 data cache per CU. The Fermi generation of Nvidia chips had a dynamically configurable 16KB or 48KB L1D. The Kepler and Maxwell generations of Nvidia GPUs can configure their L1 data caches to be 16, 32, or 48KB [40]. However, this L1 cache is only used to store local data, such as register spills, and is always bypassed when accessing global data (i.e., there is essentially no hardware-controlled R/W L1 data cache [42, 43]).

These cache sizes are unlikely to increase significantly as the general performance benefit from adding extra cache space does not outweigh the extra area taken up by these caches. That area could instead be dedicated to more computational resources, which would directly increase performance in traditional graphics and many GPGPU applications. Unfortunately, at these sizes, the large structures and streaming data used by GPGPU applications cause unnecessary cache evictions, reducing reuse and wasting energy.

Only useful data would be installed if these zero-reuse blocks were not inserted into the cache. Useful data would also be more likely to remain in the cache and be reused before eviction. Therefore, a bypass decision mechanism could increase the efficiency of the cache without requiring effort on the programmer's part or a large amount of area.

The remainder of this paper investigates adaptive *GPU cache bypassing* mechanisms that avoid inserting zero-reuse blocks into the L1 data cache of the GPU.

3. ADAPTIVE GPU CACHE BYPASSING

We propose a dynamic GPU cache bypassing technique that prevents zero-reuse blocks from being placed in the L1 data cache of the GPU compute units that access them. If a block is unlikely to be accessed again before it is evicted from the cache, the mechanism instead sends the data directly to the compute unit, bypassing the cache. This technique saves energy by avoiding needless insertions followed by later evictions and improves performance by reducing cache pollution.

The most important question for such a technique is: how can the hardware decide whether a block is zero-reuse when it fetches data during a cache miss? Previous CPU cache bypassing techniques proposed to make decisions using mechanisms such as frequency of accesses [23,27], temporal locality information [16], or reuse distance [21]. Using information related to memory addresses is impractical in GPU caches due to the large number of data accesses. Single Instruction Multiple Data (SIMD) units used in GPUs simultaneously perform the same task on different items of data, resulting in a high degree of data parallelism and large numbers of memory addresses. Using memory address-related information to make bypass decisions would require a large amount of storage, which is not amenable to GPUs. Figure 3 shows the number of 64B memory blocks accessed in our set of benchmarks. Hundreds of thousands of memory blocks are accessed during the execution of these small kernels.

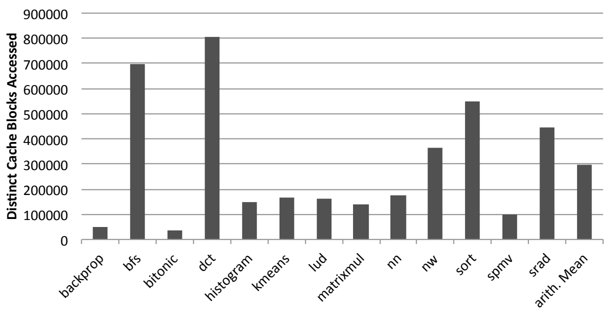

Figure 3: Number of distinct blocks accessed during the execution of each benchmark.

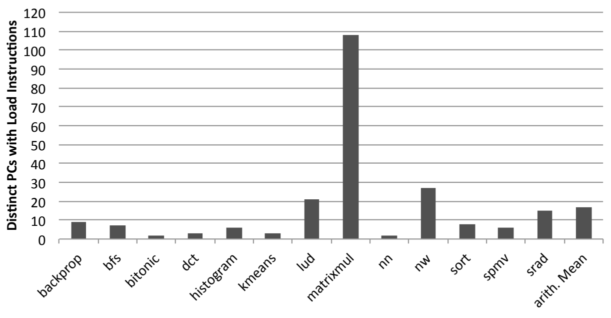

Figure 4: Number of distinct program counter values in each benchmark that execute loads.

The number of distinct memory instructions in a GPGPU workload is much smaller than the large amount of data accessed. Program behavior is dominated by a few small kernels with a high degree of thread-level parallelism.

Figure 4 shows that there are far fewer distinct load instructions executed in each benchmark. Rather than hundreds of thousands of data addresses, there are instead only tens to hundreds of distinct program counters (PCs) of memory instructions. Thus, a predictor indexed using PCs of memory instructions is more practical than one indexed with accessed addresses. There are fewer distinct entries, which requires far less on-chip storage, and there are fewer distinct values concurrently generated, which reduces the port count of the predictor. Beyond the capacity concern, a PC-based predictor can be more accurate because it learns to generalize the behavior of a single instruction to multiple data blocks.

Previous CPU dead block prediction techniques leverage the fact that sequences of memory instruction PCs tend to lead to the same behavior for different memory blocks [30, 34]. Khan *et al.* showed that in last level caches (LLCs), the PC of the last memory instruction to touch a particular block is highly correlated with whether or not the block will be used again, leading to a compact and highly accurate predictor [26]. Wu *et al.* used this observation to classify LLC blocks in terms of their likely reuse distances [48].

We extend this intuition to predict zero-reuse blocks in GPGPU workloads. Although both our technique and the sampling dead block prediction (SDBP) [26] use PCs to make a prediction, the intuition behind them is different. SDBP is designed for LLCs, where much of the temporal locality has been filtered by higher level caches. Thus, using the PC of the last memory instruction rather than a trace of PCs (as in previous work [30, 34]) achieves higher accuracy in LLCs. By contrast, our technique is designed for GPU L1 caches, where temporal information is complete. However, we propose to use the PC of the last memory instruction, rather than sequences of memory instructions, because of the observation of characteristics of GPGPU memory accesses as shown in Figure 4.

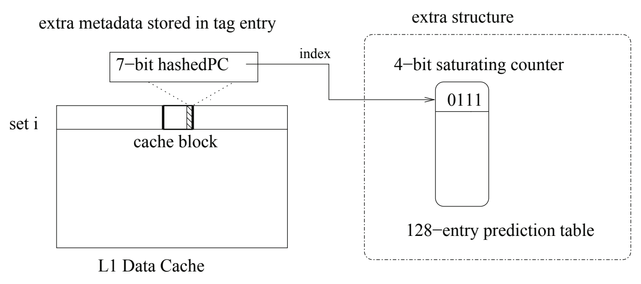

Figure 5: Structure of a PC-based bypass predictor in a GPU's L1 cache.

Another option for PC-based bypassing would be to use a trace of PC values, rather than a single PC. However, because GPU kernels are small and frequently launched, the interleaving changes frequently. This interleaving has a negative impact on warm-up time for the predictor when using PC traces rather than the last PC.

3.1 PC-Based Bypass Predictor

Figure 5 shows the structure of our PC-based bypass predictor in a GPU L1 cache. The predictor keeps a 128-entry prediction table aside the L1 cache, where each entry contains a 4-bit saturating counter. This table is indexed by a hashed PC and consumes 64 bytes of storage in each L1 cache. The number of entries of the prediction table is very small, taking advantage of the characteristics of GPU programs that there are only few distinct PCs. Each access to the prediction table yields a confidence compared with a threshold; if the threshold is met, then the corresponding block accessed by that PC is predicted as zero-reuse. Beyond the prediction table, each tag entry stores one more item of metadata: a hashed PC value (7 bits) that records the last memory instruction that referenced the current block.

No matter how high the prediction accuracy is, a bypass misprediction in this design is irreversible. That is, when a bypass decision related to a PC is made, no blocks accessed by that PC will be placed into the L1 cache. If the prediction is wrong, all subsequent blocks accessed by this PC will miss in the L1 cache, causing additional penalties for accessing lower cache levels. To correct potential mispredictions, each L2 cache block keeps an extra bit, called the *bypassBit*, to help verify predictions. When a block is selected to be bypassed on a L1 cache miss, the prediction is sent to the L2 cache with the memory request. The L2 cache stores this information in the corresponding L2 entry (set $bypassBit = 1$). If the block is referenced again before being evicted from the L2 cache, this information is sent back to the L1 cache with the requested data, indicating that the previous bypass prediction might be incorrect. The requested block will not be bypassed this time. Instead, it is placed into the L1 cache for potential verification.

3.2 Prediction Algorithm Details

Listing 1 gives the pseudocode of our PC-based bypass predictor. We use the least-recently-used (LRU) replacement policy in this example. On each L1 access, the L1 cache is searched for the tag of the requested block. If there is a tag match, then the last PC that accessed this block led to a reused block. A prediction table entry indexed by the hashed PC stored in the cache entry is decremented to indicate a potentially reused block. The current PC is hashed and stored in the cache entry, with the corresponding replacement status updated.

If it is a cache miss, the bypass prediction of the requested block is made and sent to lower level caches with the memory request. If the predictor decides not to bypass this block, the LRU block is replaced with the incoming block. The prediction entry indexed by the hashed PC stored in the LRU block entry is updated, indicating this PC likely leads to zero-reuse blocks. On receiving the requested block, the corresponding metadata is updated.

If the prediction is to bypass, the requested block will not be placed into the cache. However, there is a chance that the prediction is incorrect. If the bypassBit sent from the L2 cache is set, it is possible that this block would be reused (since it is hit in the L2 cache). In this case, instead of being bypassed again, this block is placed into the L1 cache for potential re-references and misprediction correction. The misprediction correction does not distinguish if the bypassBit set by a previous bypass prediction is from a different compute unit. The intuition is that different compute units behave similarly in GPUs. Thus, using prediction information from other compute units will not interfere with one another; by contrast, it helps correct potential mispredictions with limited information.

Note that previous warp scheduling proposals such as Cache-Conscious Wavefront Scheduling (CCWS) [46] were also designed for increasing GPU cache efficiency. Our work is orthogonal to warp scheduling techniques and can be used along with them for better performance. To fairly evaluate our work as a GPU cache management technique, we conservatively use the "Oldest-First" scheduling technique which minimizes cache thrashing caused by warp interference.

3.3 Comparison to Counter-Based Prediction

Counter-based bypass prediction [27] is a CPU LLC bypassing technique. That work proposed to use an event counter in each cache block to record an event of interest such as cache accesses. When the counter reaches a threshold, the block observes no more reuse. This information is stored in a prediction table indexed by hashed block addresses and PCs. To bypass zero-reuse blocks, the block addresses and PCs of bypass candidates are used to index to the prediction table for bypass prediction. Compared to PC-based bypass prediction which tracks repetitive program patterns, counter-based prediction tracks block access patterns. GPU programming features a small number of distinct PCs addressing a large amount of distinct data. To record block-level reuse patterns, counter-based prediction keeps extra information per block and a large prediction table. Due to the limited capacity of the GPU L1 caches, counter-based prediction consumes too much on-chip area to be practical in GPU cache designs.

Counter-based bypass prediction achieves worse performance on average and much higher storage overhead compared to PC-based bypass technique. Based on our experiments, on average, in each 16KB L1 cache, counter-based prediction takes more than 10.5KB of storage overhead, while PC-based prediction takes less than 256 bytes of overhead in each L1 cache, and a total 0.5KB of storage overhead in a shared 256KB L2 cache. In addition, PC-based bypass prediction outperforms counter-based prediction by 2.3%. We give a detailed evaluation in Section 5.

```
On each L1 access (address, PC):
if (the access is a hit) {
   /* corresponding prediction is updated to
       indicate a reused block */
   predictionTable[block[address].hashedPC)]--;
   /* PC information is stored in the cache
       entry for future verification */
   block[address].hashedPC = hash(PC);
   /* update LRU replacement status */
   block[address].LRU_stack = 0;
}
else {
   /* get bypass prediction */
   bool isBypassed = predictionTable[hash(PC)]
       >= threshold ? true: false;
   /* send memory request to L2, along with the
       prediction */
   SendMemReq (address, isBypassed);

   if (!isBypassed) {
      /* if the prediction is to not bypass a
          victim block, VictimAddr has to be
          replaced. corresponding prediction
          entry is updated to indicate a zero-
          reuse block */
      predictionTable[block[VictimAddr].
          hashedPC]++;

      /* bypassBit stored in L2 cache is sent
          back with requested data */
      bypassBit = L2Block[address].bypassBit;
      L2Block[address].bypassBit = false;
      Data = RecvMemPkt(address,  L2Block[
          address].data, bypassBit);
      /* cache installation */
      block[address].data = data;
      block[address].hashedPC = hash(PC);
      block[address].LRU_stack = 0;
   }
   else {
      /* if the prediction is to bypass, use
          the bypassBit to confirm */
      bypassBit = L2Block[address].bypassBit;
      L2Block[address].bypassBit = false;
      Data = RecvMemPkt(address,  L2Block[
          address].data, bypassBit);
      if(bypassBit) {
         /* if the bypssBit indicates a
             previous misprediction, do not
             bypass */
         isBypassed = false;
         block[address].data = data;
         block[address].hashedPC = hash(PC);
         block[address].LRU_stack = 0;
      }
      else {
         /* bypass L1 cache */
      }
   }
}
```

Listing 1: Pseudocode of our PC-based bypassing prediction technique.

Table 1: Experimental system configuration.

GPU Clock	1GHz
Compute Units (CUs)	8
CU SIMD Width	64 scalar units within 4 SIMDs
GPU L1-I/D Cache	8-way 16KB, 64B, 1 cycle of tag access, 4 cycles of data access
GPU Shared L2 Cache	16-way 256KB, 64B, 4 cycles of tag access, 16 cycles of data access
L3 Memory-side Cache	16-way 4MB, 15 cycles of tag access, 30 cycles of data access

4. EXPERIMENTAL METHODOLOGY

This section outlines our experimental methodology.

4.1 Simulation Environment

We use an in-house APU simulator that extends gem5 [4]. The simulator runs with a microarchitectural timing model of a GPU that directly executes the HSA Intermediate Language (HSAIL) [18] and produces detailed statistics including execution cycles, cache miss rate, and traffic. Table 1 shows the configuration of the GPU side of the evaluated system, which is similar to the AMD Graphics Core Next architecture [1]. The warp scheduling policy is oldest-first, which attempts to minimize cache thrashing caused by wavefront interference. All caches use a default Pseudo-LRU replacement policy.

Compared to the baseline system, each L1 bypass predictor requires a 128-entry prediction table of 4 bit counters and additional metadata of 7-bit in each tag entry, costing 224 bytes in total of storage overhead in each L1 cache. To help verify prediction accuracy, each L2 tag entry contains one extra bit of bypassBit, taking 0.5KB in total.

We also evaluate counter-based bypass prediction. For a 16KB L1 cache, our counter-based bypass predictor contains a two-dimensional prediction table with 128×128 entries, each containing five bits of prediction information. Each tag entry contains 20 bits of extra information for the hashed PC, counters, and the prediction. The storage overhead of this counter-based bypass predictor is 10.626KB.

4.2 Benchmarks

We evaluate 13 benchmarks from Rodinia [6], the AMD APP SDK, OpenDwarfs [10], and one custom microbenchmark implementing a four-byte radix sort with high data reuse. These workloads represents the OpenCL^{TM} benchmarks we have that can be compiled and run in our simulator. Table 2 lists the characteristics of the evaluated benchmarks. The benchmarks are sorted by *memory intensity* (MI, calculated as the global memory accesses per 1000 instructions) [50]. Among all the benchmarks, *matrixmul*, *spmv*, *bfs* are memory-intensive workloads, and *dct*, *sort*, *histogram*, *nw* and *lud* are compute-intensive workloads. We use medium to large inputs for each benchmark.

5. EVALUATION

This section shows our analysis of the bypass predictor, regarding energy, performance, and prediction accuracy.

5.1 Energy Saving

In this section, we evaluate the energy savings of the bypass predictor. Insertion of zero-reuse blocks wastes energy without performance improvement and may even cause

Table 2: Benchmark workloads and their inputs.

Program	Input	MI	Description
matrixmul	512 × 512	395.6	matrix multiplication
spmv	256 × 256	215.8	sparse matrix-vector multiplication
bfs	1M	202.7	breath-first search
nn	342080	130.4	k-nearest neighbor
kmeans	16384	121.8	kmeans clustering
bitonic	131072	114.3	bitonic sort
srad	512 × 512	102.2	speckle reducing anisotropic diffusion
backprop	8192 × 16	89.7	back propagation
dct	2048 × 2048	76.2	discrete cosine transform
sort	65536	76.2	radix sort
histogram	1024	43.1	histogram
nw	512 × 512	30.4	needleman-wunsch
lud	1024 × 1024	14.2	LU decomposition

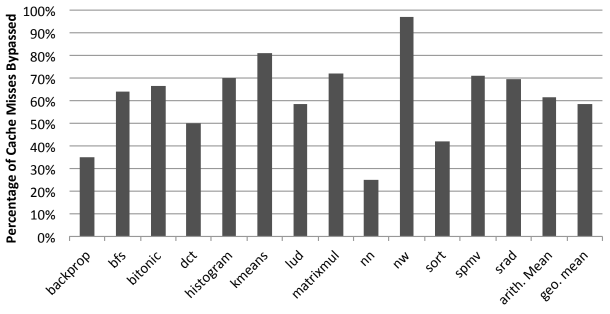

Figure 6: Ratio of bypasses to cache misses.

cache pollution. Cache bypassing significantly reduces the energy consumption by preventing unnecessary filling of data into caches. A large amount of streaming data is bypassed, reducing the energy cost and potential cache pollution.

In a conventional L1 cache, both the tag and data arrays are accessed in parallel on each L1 cache access in order to reduce latency. On a cache miss, both the tag and data arrays will be accessed again to fill the selected cache block with data from lower level of the memory hierarchy. With cache bypassing, on each L1 cache access, the tag and data arrays are accessed in parallel together with a direct access to a very small prediction table. On a cache miss predicted to bypass, the data are sent directly to the compute unit without accessing the cache structure again. As shown in Figure 6, an average of 58% of cache fills are prevented through bypassing.

The reduction of unnecessary cache fills significantly reduces the energy consumption compared to the baseline. Table 3 shows the results of CACTI 6.5 simulations [36] to determine the energy reduction by adding a PC-based bypass predictor compared to the 16KB baseline. The extra structure of the prediction table is modeled as a tag array (with four-bit tags) of a direct-mapped cache with 128 sets. Each tag entry in the L1 cache with bypassing has eight more bits,[1] and the data array remains unchanged. Figure 7 gives the reduction in energy with PC-based bypassing compared

[1]We add seven bits in each tag entry for prediction. To use CACTI correctly, we evaluated it as eight bits.

Table 3: Power cost.

Energy (nJ)	16KB baseline	bypassing
per tag access	0.00134096	0.0017867
per data access	0.106434	0.106434
per prediction table access	N/A	0.000126232
Dynamic Power (mW)	44.2935	36.1491
Static Power (mW)	7.538627	7.72904

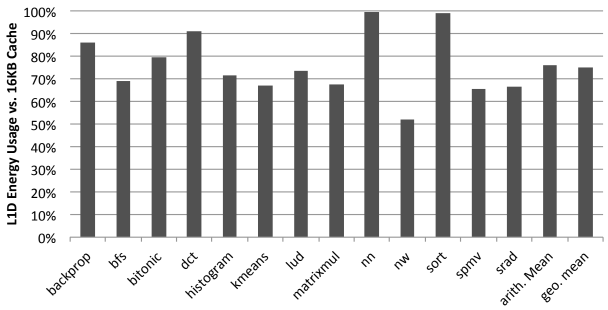

Figure 7: Energy Usage of a 16KB cache with bypassing (relative to a baseline that does not bypass).

to the 16KB baseline. The energy cost of the 16KB baseline is reduced by up to 49%, and on average by 25% with bypassing. Table 3 also shows the quantified power cost. On average, PC-based bypassing reduces dynamic power by 18% compared to the 16KB baseline and increases the leakage power by only 2.5%.

5.2 Performance

Bypassing improves the cache efficiency by preventing unnecessary filling of data into caches to cause cache pollution. Therefore data stored in caches are likely to be useful. In another word, bypassing improves cache efficiency and overall performance.

In this section, we evaluate cache miss reduction and performance improvement over a 16KB L1 cache baseline for both PC-based bypass prediction and counter-based bypass prediction. We compare both to a larger 32KB L1 cache. For brevity, we use Baseline, PC-based predictor, counter-based predictor, and 32KB Cache as abbreviations, respectively.

Figure 8 shows L1 misses normalized to the baseline system for each benchmark with different techniques and Figure 9 shows the speedup (i.e. the execution time of benchmarks on the baseline system divided by the execution time on the evaluated system). To help analyze the results, Figure 10 shows the hit rate in the L1 cache of each benchmark in the baseline system.

PC-based bypass prediction offers a significant performance improvement in the benchmarks *matrixmul*, *bfs*, and *spmv*. These benchmarks observe intermediate or low L1 hit rate in the baseline (as shown in Figure 10) because most of the data that should be reused are replaced due to cache pollution. As shown in Figure 1, these benchmarks have a high percentage of zero-reuse blocks. With PC-based bypass prediction, streaming data are bypassed and previously doomed useful blocks are kept in the L1 cache. Cache efficiency is improved for these benchmarks. Among these three benchmarks, *bfs* produces a speedup of 13% over the baseline, *spmv* yields a speedup of 9% and *matrixmul* generates a speedup of 6%. Compared to PC-based bypassing, the counter-based bypass predictor provides much less speedup for benchmarks *bfs* and *spmv* but yields a better performance for benchmark *matrixmul*. In comparison, the 32KB Cache provides less performance improvement for all three benchmarks.

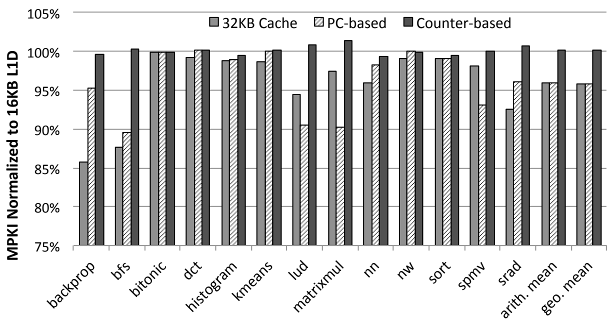

Figure 8: Reduction in L1 misses.

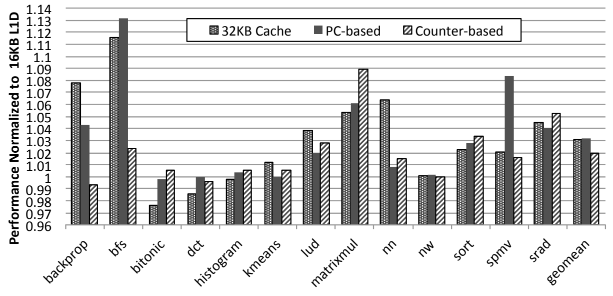

Figure 9: Speedup over the baseline..

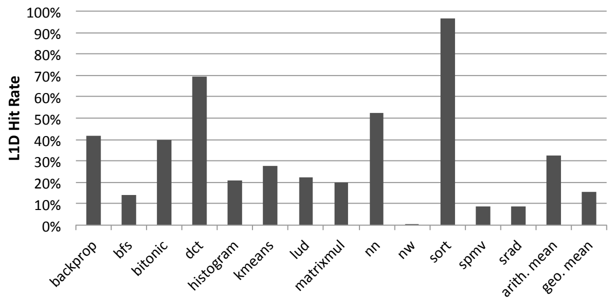

Figure 10: L1 cache hit rate in the 16KB cache.

The benchmarks *backprop* and *srad* have intermediate-to-low L1 hit rates as well as low reuse rates. The performance of *backprop* with a PC-based predictor is improved by 4.3% and *srad* reaches a speedup of 4% over the baseline.

The benchmarks *sort*, *dct*, and *lud* are compute-bound benchmarks [7]. Increasing cache size does not significantly improve performance for these benchmarks. Their overall performance mainly depends on the compute ability of SIMD processors. All three evaluated techniques yield an average speedup of about 3%.

Some benchmarks observe little performance improvement with all evaluated techniques. The benchmarks *kmeans* and *histogram* invoke many kernel launches and frequently share data between the CPU and the GPU. Their performance is thus dominated by pulling data from CPU side, resulting in no significant performance improvement with any of the techniques. The benchmark *bitonic* contains frequent barrier synchronizations [13], causing the program to execute in lock-step with no observed performance improvement with any techniques. Larger cache sizes degrade the performance due to the cache walk required when kernels complete. The benchmark *nw* puts all reused data into the scratchpad memory for computation and writes data to global memory when the computation is finished. As shown in Figure 6, with PC-based bypassing, *nw* has more than 95% of cache insertions prevented. Therefore, for *nw*, there is little performance improvement while around 50% of energy reduction with PC-based cache bypassing.

Storage is a key issue in GPU cache design. On average, the PC-based bypassing prediction in a 16KB cache outperforms both the counter-based prediction and the 32KB cache system while using far less overhead, which means almost half of the chip area dedicated for private caches is saved without performance degradation. The tension between number of compute units and the size of caches makes it infeasible to increase the cache size naïvely. For example, an AMD Radeon$^{\text{TM}}$ HD 7970 [2] GPU contains an AMD GCN processor with 32 parallel CUs. To double the cache size of its 16KB L1 caches without increasing the chip area, we estimate that up to 4 CUs would need to be removed, leading to a theoretical maximum throughput degradation of 12.5% [8, 9, 32][2].

5.3 Prediction Accuracy and Coverage

In this section, we evaluate prediction accuracy and coverage of PC-based bypassing.

There are two groups of mispredictions: false positives and false negatives. False positives are more harmful because they wrongly bypass reused blocks. Further re-references cause extra misses. The coverage of the bypass predictor is the ratio of bypass prediction to all prediction made on cache misses. Higher coverage means more opportunity for the optimization. Figure 11 shows the coverage and false positive rates of the PC-based bypass predictor. On average, the coverage rate is 58.6%, and the false positive is 12%.

Note that the reason why the false positive rate is higher than previous work is because we include incorrectly bypassed or replaced blocks as false positives. Sampling-based dead block prediction [26] calculated false positive as (number of accesses to predicted dead blocks / number of dead predictions), so only re-referenced blocks predicted dead are categorized as false positives. Using the same computation as sampling-based dead block prediction gives a false positive rate of 1% for the GPU cache bypassing.

A Case Study of the Benchmark nw.

GPU L1 caches can be treated as hardware-controlled scratchpad memories. Both of them store reused data shared within a compute unit. Programmers use scratchpad memories to bypass streaming-like data by explicitly storing only

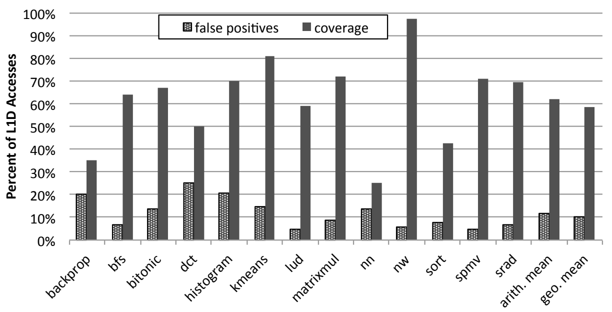

Figure 11: False positive and coverage rate of our PC-based bypassing predictor.

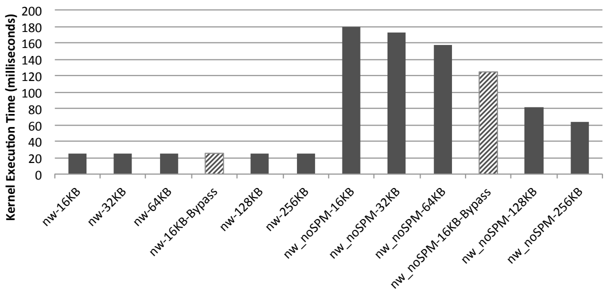

Figure 12: Execution time of nw with different cache configurations.

reused data into the scratchpad memories. A GPU L1 cache with bypassing stores reused data by adaptively bypassing streaming-like data without programmer intervention. We quantify the extent to which dynamic L1 cache bypassing can make up for the potential performance lost in production environments where the effort to program scratchpad memories is impractical.

To explore the effectiveness and limitation of adaptive L1 cache bypassing, we take a Rodinia benchmark *Needleman-Wunsch* for a case study. *Needleman-Wunsch (nw)* uses a global optimization algorithm for DNA sequence alignment in bioinformatics [6]. It dynamically loads the northern and western edges of a 2-D matrix into the scratchpad memory and processes the data in the scratchpad memory. After computation, results are written through to the main memory. Most of the kernel is spent doing partial computation in the scratchpad memory. There is very little reuse observed in L1 caches because the scratchpad filters reused data. We re-wrote the source code of *nw* to remove the use of the scratchpad memory (benchmark *nw-noSPM*). Note that we did not simply replace the *_local_* functions into *_global_* functions (which will cause significant degradation of performance); rather, we re-wrote the source code by understanding the original algorithm resulting in a best-effort program without the use of scratchpad memories.

Figure 12 shows the execution time of *nw* and *nw-noSPM* with different configurations. As shown in the left of Figure 12, performance is slightly changed with different cache configurations due to the highly reuse in the scratchpad memory. Without using scratchpad memories, *nw-noSPM* takes 7 times longer than the original program. With the help of cache bypassing, the gap is reduced by 30%, which outperforms a 64KB L1 cache. Note that cache bypassing is running with 16KB L1 caches.

[2]Based on estimates derived from die images and expert teardowns [8, 9], the total chip area is $352mm^2$ and 32 CUs take up approximately $176mm^2$. The computational logic in each CU is estimated to be approximately $3.7mm^2$, and a 16KB cache structure takes up to $1.8mm^2$. If doubling the size of each cache to 32KB leads to an area increase of $0.8mm^2$, a chip of roughly the same area ($176mm^2$) would require removing 4 CUs to fit the extra cache storage.

This limited study shows that, while the technique currently cannot replace scratchpad memories programmed by expert programmers, it can improve performance in production environments where such programming effort is impractical, as well as programmability. We believe mechanisms such as our predictor bring GPU programming closer to general-purpose programming in terms of programmability while retaining the performance advantage of GPUs.

6. RELATED WORK

6.1 Scratchpad Management Techniques

Compiler-controlled scratchpad memories were proposed to improve the efficiency of scratchpad memories [3]. Knight *et al.* proposed an optimizing compiler for architectures with software-managed memory hierarchies to explicitly manage scratchpad memories [28]. Kandemir *et al.* proposed a compiler-controlled dynamic on-chip scratchpad memory management technique for real-time embedded systems [25].

6.2 GPU Cache Related Work

Jia *et al.* proposed a memory request prioritization buffer (MRPB) to improve GPU performance, which also employs cache bypassing to mitigate intra-warp contention [22]. Instead of distinguishing reused blocks from significant amount of zero-reuse blocks, MRPB blindly and aggressively bypasses memory requests when there are resource limits, which can cause performance degradation. Compared to MRPB, our adaptive cache bypassing does not cause any performance degradation. To evaluate MRPB in terms of programmability, Jia *et al.* created an "unshared" version of some Rodinia benchmarks that used scratchpad memory by simply using global memory instead. Simply replacing *_local_* functions with *_global_* ones will cause significant degradation of performance and lead to biased comparison. In our case study, we rewrote the source code by understanding the original algorithm, resulting in a best-effort program.

Rogers *et al.* proposed cache-conscious wavefront scheduling (CCWS) to improve GPU cache efficiency by avoiding the data thrashing that causes cache pollution [46]. CCWS restricts the number of wavefronts that are able to access the caches by changing the hardware to schedule a limited number of wavefronts, which adversely affects the ability of hiding high memory access latency of GPUs. Our technique bypasses the unused blocks without starving the SIMD pipeline by artificially limiting the wavefront availability to reduce cache thrashing.

Lee and Kim proposed a thread-level-parallelism-aware cache management policy to improve performance of the shared LLC in heterogeneous multi-core architecture [31]. They focus on shared LLCs that are dynamically partitioned between CPUs and GPUs. Mekkat *et al.* proposed a similar idea for heterogeneous LLC management to better partition LLC for GPUs and CPUs in a heterogeneous system [35].

6.3 CPU Cache Bypassing

Much previous research focuses on CPU cache management techniques [14,16,20,21,23,45,47,49]. We only discuss papers that have explored bypassing in CPU caches.

Tyson *et al.* proposed bypassing based on the hit rate of the memory access instructions [47], while Johnson *et al.* propose to use the access frequency of the cache blocks to predict bypassing [45]. Kharbutli and Solihin propose using counters of events such as number of references and access intervals to make bypass predictions in the CPU LLC [27]. All of these techniques use memory address-related information to make the prediction, requiring significant storage that would be impractical for GPU caches.

PC trace-based dead block prediction leveraged the fact that sequences of memory instruction PCs tend to lead to the same behavior for different memory blocks [30]. This dead block prediction scheme is useful for making bypass predictions in CPUs. We show that GPU kernels are small with few distinct memory instructions. Using only the PC of the last memory instruction to access a block is sufficient for a compact GPU bypassing predictor.

Khan *et al.* proposed a sampling-based predictor to make CPU LLC dead block predictions [26]. Their technique, which uses set-sampling to reduce storage and power overhead, is significantly more complex than our bypassing predictor. In addition, their techniques requires storing a large amount of metadata in the cache that is unnecessary in our bypass predictor. That is, each block in the cache must be associated with a prediction bit to drive the replacement policy, while our technique simply discards blocks predicted as bypass candidates so no such prediction bit is needed. The sampling-based predictor uses an extra data structure called the sampler to keep less state and require fewer prediction table updates compared to previous dead block prediction techniques. To increase the prediction accuracy, it uses a complex and large prediction table to reduce hash collision. Compared to this work, our bypass predictor has far less storage and energy overhead and similar accuracy using a much smaller and simpler prediction table, based on the observation that GPUs have many accesses from a small number of instructions. We also provide a simple and efficient misprediction correction mechanism, which is irreversible in previous CPU cache bypassing work.

Li *et al.* proposed using a global tracking of incoming victim block pairs to make bypass prediction designed for CPU LLC. Cache Bursts [34] is another dead block prediction technique that exploits bursts of accesses hitting the MRU position to improve predictor efficiency. For GPU workloads that use scratchpad memories, the majority of re-references have been filtered. Gaur *et al.* [15] proposed bypass and insertion algorithms for exclusive LLCs to adaptively avoid filling them with unmodified dead blocks.

7. CONCLUSION AND FUTURE WORK

Current GPU cache hierarchies are inefficient in the face of streaming data. This paper proposes a simple but effective cache bypassing technique to improve GPU L1 cache efficiency and reduce energy overhead without requiring additional effort on the programmer's part. Based on our evaluation, this technique yields significant cache energy reduction while outperforming a cache of twice the baseline size.

Our initial study into scratchpad replacement was limited to a single program, as appropriately removing scratchpad memory usage from an application is a time-consuming process. We plan on studying more of these applications in the future. Nonetheless, from our initial results, we show that, while our technique gives positive and promising results, we cannot currently reach the performance attained by an expert using scratchpad memory. We believe that there are further hardware-assisted mechanisms that can help bridge this gap and plan to explore such techniques in future work.

8. ACKNOWLEDGEMENTS

Daniel A. Jiménez and Yingying Tian are supported by National Science Foundation grants CCF-1216604 and CCF-1012127.

AMD, the AMD Arrow logo, Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple, Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

9. REFERENCES

[1] Advanced Micro Devices, Inc. AMD Graphics Cores Next (GCN) Architecture. `http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, Jun. 2012.

[2] Advanced Micro Devices, Inc. AMD Radeon^{TM} HD 7900 Series Graphics Cards: 7970, 7970 GHz, 7950. `http://www.amd.com/en-us/products/graphics/desktop/7000/7900`, Jan. 2015.

[3] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *Proc. of the Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2004.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[5] C.-A. Bohn. Kohonen Feature Mapping through Graphics Hardware. In *Proc. of the Int'l Conf. on Computational Intelligence and Neurosciences*, 1998.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[8] Chipworks, Inc. Inside the ASUS AMD 7970 graphics card - TSMC 28nm! `http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-asus-amd-7970-graphics-card-tsmc-28-nm/`, Feb. 2012.

[9] Chipworks, Inc. A Look at Sony's Playstation 4 Core Processor. `http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/a-look-at-sonys-playstation-4-core-processor`, Nov. 2013.

[10] W. Feng, H. Lin, T. Scogland, and J. Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proc. of the Int'l Conf. on Performance Engineering (ICPE)*, 2012.

[11] J. Fung and S. Mann. OpenVIDIA: Parallel GPU Computer Vision. In *Proc. of the Int'l Conf. on Multimedia*, 2005.

[12] J. Fung, F. Tang, and S. Mann. Mediated Reality Using Computer Graphics Hardware for Computer Vision. In *Proc. of the Int'l Symp. on Wearable Computers*, 2002.

[13] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2007.

[14] R. V. Garde, S. Subramaniam, and G. H. Loh. Deconstructing the Inefficacy of Global Cache Replacement Policies. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2008.

[15] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2011.

[16] A. González, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proc. of the Int'l Conf. on Supercomputing (SC)*, 1995.

[17] L. Howes and A. Munshi. The OpenCL Specification Version 2.0, 2014. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[18] HSA Foundation. HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG). `http://www.hsafoundation.com/?ddownload=4945`, Jun. 2014.

[19] W. W. Hwu, editor. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[20] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2010.

[21] J. Jalminger and P. Stenstrom. A Novel Approach to Cache Block Reuse Predictions. In *Proc. of the Int'l Conf. on Parallel Processing (ICPP)*, 2003.

[22] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2014.

[23] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-time Cache Bypassing. *IEEE Trans. on Computers*, 48(12):1338–1354, 1999.

[24] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt. GPUDet: a Deterministic GPU Architecture. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[25] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A Compiler-Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):243–260, Feb. 2004.

[26] S. M. Khan, Y. Tian, and D. A. Jiménez. Sampling Dead Block Prediction for Last-Level Caches. In *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2010.

[27] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Trans.*

on Computers, 57(4):433–447, April 2008.

[28] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for Explicitly Managed Memory Hierarchies. In *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[29] G. Kyriazis. Heterogeneous System Architecture: A Technical Review. Technical report, HSA Foundation, 2012.

[30] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2001.

[31] J. Lee and H. Kim. TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2012.

[32] Leonidas. AMD R1000/Tahiti Die-Shot. `http://www.3dcenter.org/abbildung/amd-r1000tahiti-die-shot-markiert`, Sep. 2012.

[33] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2007.

[34] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2008.

[35] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[36] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, HP Laboratories, Apr. 2009.

[37] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[38] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010.

[39] Nvidia Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[40] Nvidia Corp. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. 2012.

[41] Nvidia Corp. CUDA C Programming Guide Version 6.5. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, Aug. 2014.

[42] Nvidia Corp. Tuning CUDA Applications for Kepler. `http://docs.nvidia.com/cuda/kepler-tuning-guide/`, Aug. 2014.

[43] Nvidia Corp. Tuning CUDA Applications for Maxwell. `http://docs.nvidia.com/cuda/maxwell-tuning-guide/`, Aug. 2014.

[44] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proc. of the IEEE*, 96(5):879–899, 2008.

[45] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing Reuse Information in Data Cache Management. In *Proc. of the Int'l Conf. on Supercomputing (SC)*, 1998.

[46] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2012.

[47] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 1995.

[48] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2011.

[49] M. Zahran. Cache Replacement Policy Revisited. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2007.

[50] J. Zhao, G. Sun, G. H. Loh, and Y. Xie. Energy-efficient GPU Design with Reconfigurable In-package Graphics Memory. In *Proc. of the Int'l Symp. on Low Power Electronics and Design (ISPLED)*, 2012.