CPU-Assisted GPGPU on Fused CPU-GPU Architectures

Yi Yang, Ping Xiang, Mike Mantor[*], Huiyang Zhou

Department of Electrical and Computer Engineering   ***Graphics Products Group**
North Carolina State University   *Advanced Micro Devices*
{yyang14, pxiang, hzhou}@ncsu.edu   *Michael.Mantor@amd.com*

Abstract

This paper presents a novel approach to utilize the CPU resource to facilitate the execution of GPGPU programs on fused CPU-GPU architectures. In our model of fused architectures, the GPU and the CPU are integrated on the same die and share the on-chip L3 cache and off-chip memory, similar to the latest Intel Sandy Bridge and AMD accelerated processing unit (APU) platforms. In our proposed CPU-assisted GPGPU, after the CPU launches a GPU program, it executes a pre-execution program, which is generated automatically from the GPU kernel using our proposed compiler algorithms and contains memory access instructions of the GPU kernel for multiple thread-blocks. The CPU pre-execution program runs ahead of GPU threads because (1) the CPU pre-execution thread only contains memory fetch instructions from GPU kernels and not floating-point computations, and (2) the CPU runs at higher frequencies and exploits higher degrees of instruction-level parallelism than GPU scalar cores. We also leverage the prefetcher at the L2-cache on the CPU side to increase the memory traffic from CPU. As a result, the memory accesses of GPU threads hit in the L3 cache and their latency can be drastically reduced. Since our pre-execution is directly controlled by user-level applications, it enjoys both high accuracy and flexibility. Our experiments on a set of benchmarks show that our proposed pre-execution improves the performance by up to 113% and 21.4% on average.

1. Introduction

The integration trend of CMOS devices has led to fused architectures, in which the central processing units (CPUs) and graphics processing units (GPUs) are integrated onto the same chip. Recent examples include Intel's sandy bridge [21], on which CPUs and GPUs are on one chip with a shared on-chip L3 cache, and AMD accelerated processing unit (APU) [26] on which both on-chip CPUs and GPUs share the same off-chip memory [19]. Such heterogeneous architectures provide the opportunity to leverage both the high computational power from GPUs for regular applications and flexible execution from CPUs for irregular workloads. In this paper, we assume that the fused CPU-GPU architecture has a shared L3 cache and shared off-chip memory between CPUs and GPUs and we propose a novel approach to collaboratively utilize the CPU and GPU resources. In our proposed approach, called CPU-assisted GPGPU (general purpose computation on graphics processor units), after the CPU launches a GPU program, it starts a pre-execution program to prefetch the off-chip memory data into the shared L3 cache for GPU threads.

Our proposed CPU-assisted GPGPU works as follows. First, we develop a compiler algorithm to generate the CPU pre-execution program from GPU kernels. It extracts memory access instructions and the associated address computations from GPU kernels and then adds loops to prefetch data for concurrent threads with different thread identifiers (ids). The update of the loop iterators provides a flexible way to select/skip thread ids for prefetching (see Section 4). Second, when the GPU program is launched, the CPU runs the pre-execution program. Such pre-execution is expected to warm up the shared L3 cache for GPU threads since (1) the pre-execution program only contains the memory operations (and address calculations) but not floating point/ALU computations; and (2) the CPU runs at a higher frequency and is more aggressive in exploiting instruction-level parallelism (ILP). To make the proposed pre-execution effective, it is critical to control the timing of the prefetches since they need to be issued early enough to hide memory access latencies while at the same time not so early that the prefetched data might be replaced before being utilized. We propose two mechanisms to achieve this through the loop iterator update code in the pre-execution program. The first is an adaptive approach, which requires a new CPU instruction to inquire the performance counter of L3 cache hits periodically so as to adjust the loop iterator update. The insight is that if there are too many L3 hits experienced by the CPU pre-execution program, it means that CPU is not effective in fetching new data into the L3 cache. As a result, the CPU needs to run further ahead by increasing the amount of loop iterator update. On the

other hand, if there are too few L3 cache hits, meaning that the CPU pre-execution program may skip too many threads, it reduces the loop iterator update so as to select more thread ids for prefetching. The second is a static approach, which determines the best loop iterator update value based on profiling and does not require any new hardware support.

In summary, this paper makes the following contributions: (1) we propose to utilize the otherwise idle CPU to assist GPGPU through pre-execution; (2) we propose compile algorithms to generate the CPU pre-execution program from different types of GPU kernels; (3) we propose simple yet effective approaches to control how far the CPU code runs ahead of GPU threads; and (4) we implemented our proposed schemes by integrating and modifying the MARSS X86 [13] and the GPGPUsim [1] timing simulators and our results show that our proposed CPU pre-execution can improve the performance of GPU programs by up to 113% (126%) and 21.4% (23.1%) on average using adaptive iterator update (fixed iterator update). The cost of achieving such performance gains is nominal: the average instruction overhead of the CPU pre-execution program is 0.74% (0.69%) of the number of instructions executed by GPU using adaptive iterator update (fixed iterator update).

The remainder of the paper is organized as follows. In Section 2, we present a brief background on GPGPU and fused CPU-GPU architectures. In Section 3, we present our modeling of fused CPU-GPU architecture and our experimental methodology. Section 4 discusses our proposed CPU-assisted GPGPU in detail. The experimental results are presented in the Section 5. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2. Background

2.1. General Purpose Computation on Graphics Processor Units (GPGPUs)

State-of-the-art GPUs use many-core architecture to deliver very high computational throughput. The processor cores in a GPU are organized in a hierarchy [15]. In NVIDIA/AMD GPUs, a GPU has a number of streaming multiprocessors (SMs) / SIMD engines and each SM/SIMD engine in turn contains multiple streaming processors (SPs) / thread processors (TP). The on-chip memory resource includes register files, shared memory, and relatively small caches for different memory regions. In GPUs, a large number of threads are supported to run concurrently in order to hide long off-chip memory access latencies. The threads follow the single-program multiple-data (SPMD) program execution model and are also organized in a hierarchy. A number of threads are grouped in a warp (32 threads for NVIDIA GPUs) or a wavefront (64 threads for AMD GPUs), which are executed in the single-instruction multiple-data (SIMD) manner. Multiple warps/wavefronts are assembled together into a thread block/workgroup with the capability to communicate data through on-chip shared memory. Each SM can host one or more thread blocks depending on the resource usage of each thread.

2.2. Fused CPU-GPU architectures

Advances in the CMOS technology make it possible to integrate multi-core CPUs and many-core GPUs on the same chip, as exemplified with the latest AMD's accelerated processing units (APUs) and Intel's Sandy Bridge processors. Figure 1 shows such a fused architecture with a shared L3 cache and shared off-chip memory.
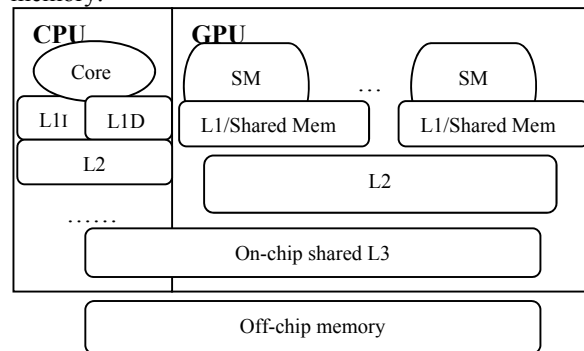

Figure 1: A fused CPU-GPU architecture with a shared on-chip L3 cache and off-chip memory.

Compared to high-end discrete GPUs, the GPUs on current fused architectures have less computation throughput and lower memory access bandwidths. However, with the shared off-chip memory, fused architectures eliminate the costly CPU-GPU data transfers [19] and the existence of the shared on-chip L3 cache opens more opportunities for close collaboration between GPUs and CPUs. Compared to CPUs, the on-chip GPUs still deliver high computational power. For example, on AMD E2-3200 APU, the GPU module (HD6370D) has the throughput of 141.8 GFLOPS while the CPU has a throughput of 38.4 GFLOPS with SSE [25]. In this paper, we propose a new approach to collaboratively utilize both CPU and GPU resources on fused architectures efficiently to achieve high performance GPGPU.

3. Architectural Modeling and Experimental Methodology

In this paper, we model the fused architecture as shown in Figure 1. To build our simulator

infrastructure, we use the full system X86 timing simulator MARSSx86 [13] for the CPU part and the GPGPUSIM [1] for the GPU part. To merge two simulators to model the fused architecture, we consider the CPU simulator as the host, meaning that the GPU simulator is invoked from the CPU simulator. In every few CPU cycles determined by the frequency ratio of CPU over GPU, the CPU simulator invokes a GPU cycle. We also ported the DRAM model in the GPGPUSIM into marssx86.

To enable the full system simulator to run CPU code and GPU code collaboratively, we partition the memory space of the fused CPU-GPU architecture into two parts: the lower address memory space that is used solely by CPU and the upper address memory space that can be accessed by both GPU and the CPU. For example, if we allocate 256MB memory for the whole system, we reserve the upper 128MB memory by passing "-mem=128M" as a parameter to boot the Linux operating system so that the operating system only uses the lower 128MB memory and reserves the upper 128MB memory for GPU. The GPU accesses the DRAM directly with physical addresses starting from 128MB. If a CPU application needs to access the upper memory space, we need to first use the Linux function 'ioremap' to map the upper 128MB memory space into the operating system as a memory device. Then, applications in user space can use the 'mmap' system call to map the memory device into the user virtual memory space. This way, applications in user space and GPU programs can access the same DRAM memory. As a side effect of our simulator infrastructure, the CPU memory accesses have higher overheads than GPU memory accesses since applications in CPU user space need to first perform address mapping to get physical addresses. However, as shown in Section 5, our proposed CPU pre-execution can achieve significant performance improvement even with our pessimistic handling of CPU memory accesses compared to GPU memory accesses.

The parameters in our simulator are set up to model AMD APU E2-3200 [26], except the shared L3 cache. For the CPU part, we model a 4-way-issue out-of-order CPU running at 2.4 GHz with a 128KB L1 cache and a 512KB L2 cache. For the GPU part, we model a GPU with 4 SMs and each SM contains 32 SPs running at 480M Hz. Each SP can deliver up to 2 flops (a fused multiplication and add) every cycle. As a result, the overall GPU computation power in our model is about 122.8GFLOPS. The register file and the shared memory in each SM are 32kB (8k registers) and 16KB, respectively. Each SM can support up to 768 active threads. In our experiments, we also vary the key

parameters to explore the design space of the on-chip GPU. Our dram memory model has 8 memory modules and every module support a bus width of 32 bits. So, the bandwidth of DRAM memory in the simulator is 600M Hz * 32 bits * 8=19.2GB/s. Because the frequency of GPU is 1/5 of the CPU frequency, the simulator executes one GPU cycle every 5 CPU cycles and a DRAM cycle every 4 CPU cycles. We also add the L3 cache to the simulator, which is 4MB and is shared by the CPU and the GPU. The L3 cache hit latency for CPU is 40 CPU cycles while the L3 hit latency for GPU is 20 GPU cycles (i.e., 80 CPU cycles). The difference is to account for the fact the CPU core is located closer to the L3 cache than GPU and GPU is less latency sensitive than CPU.

The benchmarks used in our experiments are listed in Table 1. Among the benchmarks, Blackscholes, Vecadd, and Montecarlo are from NVIDIA SDK [16]. BitonicSort and Convolution are from AMD SDK [3]. Matrix multiplication is from [1]. We implemented and optimized transpose-matrix-vector multiplication and matrix-vector multiplication, which have similar performance to the CUBLAS library. The Fast Fourier Transform implementation is based on the work by Govindaraju et al. [14].

Table 1. Benchmarks used in experiments

Benchmarks	Input sizes	Number of threads	Threads per thread block
Blackscholes (BSc)	1M	1M	128
Vector-Add (VD)	1M	1M	128
Fast Fourier Transform (FFT)	1M	512K	128
Matrix Multiplication (MM)	512x512	256K	16x16
Convolution (Con)	1kx1k & 3x3	1M	16x16
Transpose matrix vector multiplication (TMV)	512x2048	512	256
BitonicSort (BS)	2M	1M	256
MonteCarlo (MC)	256K	128K	256
Matrix-vector multiplication (MV)	16x65k	65K	256

Our proposed compiler algorithms to generate CPU code from GPU kernels (See Section 4) are implemented using a source-to-source compiler infrastructure, Cetus [22].

4. CPU-Assisted GPGPU

In our proposed CPU assisted GPGPU, after the CPU launches the GPU kernel, it starts running a pre-execution program to prefetch data into the shared L3 cache for GPU threads. Although we use the same compiler algorithm to generate CPU pre-execution programs from different GPU kernels, for the purpose

of clarity, we classify GPU kernels into two types, (a) lightweight workload in a single GPU thread (LWST), and (b) heavyweight workload in a single GPU thread (HWST), and discuss the generation of CPU pre-execution programs separately. The difference between the two types is that HWST GPU kernels have one or more loops, which contain global memory accesses while LWST kernels do not.

In this section, we first present our compiler algorithm to generate the CPU pre-execution program from LWST GPU kernels. Next, we discuss the mechanisms to control how far the CPU pre-execution can run ahead of GPU threads. Then, we present the generic compiler algorithm to generate the CPU pre-execution program from LWST/HWST GPU kernels.

4.1. Generating the CPU Pre-Execution Code from LWST GPU Kernels

For LWST GPU kernels, our proposed compiler algorithm to generate the pre-execution program is shown in Figure 2.

1. For a GPU kernel, extract its memory operations and the associated address computations and put them in a CPU function; replace thread id computation with an input parameter.
2. Add a nested loop structure into the CPU code to prefetch data for concurrent threads.
   a. The outer loop traverse through all TBs. The iterator starts from 0 and the loop bound is the number of thread blocks of the GPU program. The iterator update is the number of concurrent TBs, meaning the number of TBs that can run concurrently on the GPU.
   b. The second-level loop traverses through concurrent threads. The loop iterator starts from 0 and the loop bound is the number of concurrent threads (which is the product of TB size and the number of concurrent TBs). The iterator update is set as a product of three parameters, unroll-factor, batch_size, and skip_factor.

Figure 2. The compiler algorithm to generate CPU pre-execution program from LWST GPU kernels.

As shown in Figure 2, the algorithm contains two parts. First, it extracts all the memory access operations and the associated address computations from a GPU kernel and puts them in a CPU function. The store operations are converted to load operations. The thread id is converted to a function parameter. Second, the compiler adds loops so as to prefetch data for concurrent threads. Since GPU threads are organized in thread blocks (TBs), a nested loop is inserted. The outer loop is used to traverse through all TBs. The iterator starts from 0 and is updated with the number of concurrent TBs, which is determined by the resource usage (registers, shared memory, and the TB size) of each TB and the GPU hardware configuration. The

resource usage information is available from the GPU compiler such as nvcc and the hardware information is dependent on the target platform. The second-level loop is used to traverse through all concurrent threads in these concurrent TBs and the iterator is used to compute the thread ids, for which the data will be prefetched. As shown in Figure 2, the iterator update is set as a product of three parameters (unroll_factor * batch_size * skip_factor). The last two are used in our proposed mechanisms to control how far the pre-execution program should run ahead of GPU threads (See Section 4.2). The unroll factor is used to boost the memory requests from the CPU. Before dissecting this parameter, we first illustrate our compiler algorithm using a vector-add GPU kernel, which is an LWST kernel as it does not have any loops containing global memory accesses. Both the GPU code and the CPU pre-execution code are shown in Figure 3.

```
__global__ void VecAdd(const float* A, const float* B,
float* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}                           (a)
```

```
float memory_fetch_for_thread (int n) {
  return (A[n] + B[n] + C[n]);  /* A, B, C are the CPU
                 pointers mapped to the GPU memory */
}
float cpu_prefetching( … ) {
  unroll_factor = 8;
  //added loop to traverse thread blocks
  for (j = 0; j < N_tb; j+= concurrent_tb)  {
   //added loop to traverse concurrent threads
   for (i = 0; i < concurrent_tb*tb_size ;
      i+=skip_factor* batch_size*unroll_factor)  {
    for (k=0; k<batch_size; k++) {
     int thread_id = i + skip_factor* k*unroll_factor
        + j* tb_size;
     // unrolled loop
     float a0 = memory_fetch_for_thread (thread_id+
          skip_factor *0);
     float a1 = memory_fetch_for_thread (thread_id+
          skip_factor *1);
     ……
     sum += a0+a1+a2+a3+a4+a5+a6+a7; /* operation
       inserted to overcome dead code elimination */
    }
   //Updating skip factor (See Section 4.2)
   …
  } } }              (b)
```

Figure 3. A code example for LWST. (a) A vector-add GPU kernel; (b) the (partial) CPU pre-execution program.

As shown in Figure 3, the function 'memory_fetch_for_thread' is a result of extracting the memory accesses from the GPU kernel and can be used to prefetch data for the thread with thread id 'n'. The

memory update operation on 'C[n]' is converted to a load operation. The loaded values are summed together so as to avoid the compiler eliminating this function as dead code when the CPU program is compiled. The outer loop in the function 'cpu_prefetch' is the one that traverse all TBs. The loop bound is 'N_tb' is computed as (N / TB size), i.e., number of TBs. The iterator update 'concurrent_tb' is the number of TBs that can run concurrently on the GPU. The second loop with the iterator 'i' is introduced to prefetch data for concurrent threads. The loop iterator 'i' is used to compute the thread ids, for which the data will be prefetched. Since only those thread ids: (thread_id + skip_factor * 0), (thread_id + skip_factor * 1), ..., (thread_id + skip_factor * (unroll_factor − 1)) will be used for prefetching, the variable 'skip_factor' determines the number of threads to be skipped before a thread id is used for prefetching. We initialize this variable 'skip_factor' to be 'L3 cache line size / the size of float' (16 according to our model) so that we do not waste CPU instructions prefetching the data from the same cache line. Similarly, if the next-line prefetching is enabled, 'skip_factor' is initialized to '2 x L3 cache line size / the size of float'.

In order to make the CPU pre-execution program effective, we need to consider the memory traffic carefully as the GPU in fused GPU-CPU architectures can easily dominate the memory traffic, in which case the CPU prefetching impact will be very limited. The reasons are (1) a GPU has many SPs (128 in our model) and every SP can issue one memory fetch in one GPU cycle. If these fetches miss the GPU caches and cannot be merged, GPU will have a high rate of off-chip memory accesses; (2) GPU is designed to support high degrees of thread-level parallelism (TLP) and independent threads can issue memory requests as long as there are no structural hazards on resources to support outstanding requests such as miss status handling registers (MSHRs) or memory request queues. As a result, although the GPU frequency is slower than the CPU frequency, the number of memory fetches from the GPU can be much larger than those from the CPU. We analyze this effect using the example code in Figure 3 and the results are shown in Figure 4.

In Figure 4, we compare the memory requests generated from the CPU running the pre-execution code with different unroll factors (labeled 'un_N'). We also enable the L2 cache next-line prefetcher from the CPU side to boost the number of memory requests from CPU (labeled 'un_N_pref'). The rate of memory requests generated by the GPU is also included for comparison. From the figure, we can see that (1) increasing the unroll factor increases the number of memory requests significantly. With a unroll factor of

16, the memory requests are about 1.5X of those with a unroll factor of 1. (2) Prefetching is also important to maximize the memory requests from the CPU side. An unroll factor of 8 combined with next line prefetching achieve good utilization of the request queue at the CPU L2 cache, which is the reason why we set the value of unroll factor as 8 in Figure 3. (3) Given the high number of GPU SPs, GPU memory requests are about 3X compared to the CPU memory requests using our default 'un_8_pref' configuration. We also tried with increasing the number of cache lines which are prefetched in L2 to further increase the number of memory requests from CPU, our results with prefetching 2 or 4 cache lines only show less than 0.2% speedup compared to the next-line prefetcher. Therefore, we use the next-line prefetcher in our experiments.
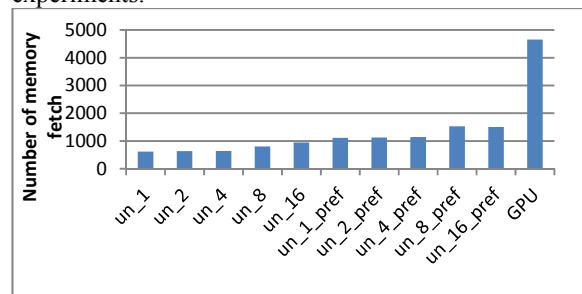

Figure 4. Comparing memory requests generated from different versions of CPU code and from the GPU for every 100,000 cycles. "un_N" means the CPU pre-execution program with the unroll factor of N; "un_N_pref" means the CPU pre-execution program with the unroll factor of N and with the CPU L2 cache next-line prefetch enabled.

4.2. Mechanisms to Control How Far the CPU Code Can Run Ahead of GPU Threads

As shown in Figure 2, the CPU pre-execution code primarily contains the memory operations from the GPU kernel. Considering the fact that CPU runs at a higher frequency and employs out-of-order execution to exploit instruction-level parallelism (ILP), we expect that the CPU pre-execution code runs ahead of the GPU kernels, meaning that when the CPU code selects a thread id for prefetching, the corresponding GPU thread has not yet reached to the corresponding memory access instruction. As discussed in Section 4.1, the parameter 'skip_factor' determines how many thread ids to be skipped before one is used for prefetching. Adjusting this parameter provides a flexible way to control the timeliness of the CPU prefetches. Here, we propose two mechanisms to adjust

this parameter. The first one is an adaptive approach and the second is a fixed value based on profiling.

In our adaptive approach, we design the update algorithm for the variable 'skip_factor' based on the following observations. If the CPU pre-execution program has experienced too many L3 cache hits, it means that the memory requests generated from the CPU are not useful because no new data are brought into the L3 cache and it can be that the GPU threads are running ahead and already brought in the data. Therefore, we need to increase the 'skip_factor' to make the CPU run further ahead. On the other hand, if there are too few L3 cache hits for CPU pre-execution, it means that CPU side is running too far ahead and we can reduce 'skip_factor' to skip fewer thread ids to generate more prefetches. To determine whether there are too many or too few L3 cache hits, we periodically sample the number of L3 cache hits for the CPU and compare the current sample with the last one. If the difference is larger than a threshold, which is set as a fixed value of 10 (our algorithm is not sensitive to this threshold setting as shown in Section 5.4), we need to update the skip_factor. The implementation of this adaptive approach for the code in Figure 3 is shown in Figure 5. Such update code is inside the second-level loop with iterator 'i' in the code shown in Figure 3 and is executed after we process a batch of thread ids. In other words, the variable 'batch_size' determines how often we update the 'skip_factor'. In our implementation, batch size is set to 16, meaning that we update the 'skip_factor' once we process (16 x 8 x skip_factor) thread ids.

```
//Accessing an L3 cache Performance counter
ptlcall_cpu_read_value(PTLCALL_CPU_NAME_CPU_
HIT_L3, &hitnumber);
if (hitnumber-last_hit>threshold) skip_factor += 32;
else  if (back_dis != skip_factor -32) {
   //preventing skip_factor bouncing between two values
      skip_factor -= 32;
       back_dis = skip_factor;
}
last  hit = hitnumber;
```

Figure 5. Adaptive update of the variable 'skip_factor'.

Since our adaptive approach needs the L3 cache hit statistics for CPU, we introduce a new instruction to access this L3 cache performance counter and this new instruction is implemented through a new 'ptlcall_cpu_read_value' function in our simulator. As shown in Figure 5, if the CPU has too many L3 cache hits, we increase the 'skip_factor' by 32. If the CPU has too few L3 cache hits, the variable is decreased by 32. The check 'if(back_dis != skip_factor -32)' prevents 'skip_factor' from bouncing between its current value and (the current value – 32). The variable

'back_dis' is initialized as 0. As discussed in section 4.1 the constant value of 32 is used as '2 x L3 cache line size / the size of float'. In Figure 6, we examine the effectiveness of our proposed adaptive update approach. In this figure, we report the value of the variable 'skip_factor' over time for different benchmarks. We can see that for most benchmarks, the variable quickly converges to a fixed value, implying that the GPU kernel has a stable memory access pattern and the CPU pre-execution code keeps a fixed distance ahead of GPU threads. For the benchmarks BS and MV, the value of this variable changes over time, indicating that their memory access patterns are not stable due to the data dependent nature of sorting algorithms (BS) and the L1/L2 caching effects (MV).
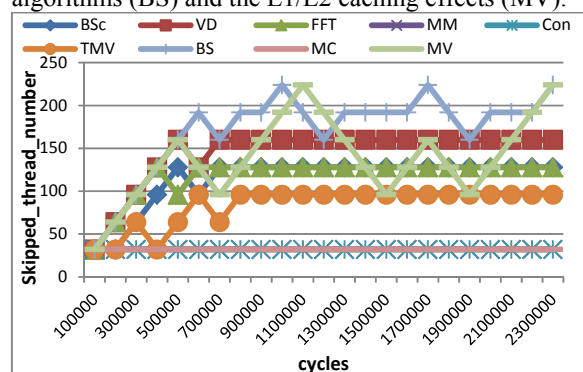

Figure 6. The value of the variable 'skip_factor' over time using our adaptive update approach shown in Figure 5.

Since many GPU workloads have regular memory access patterns, as shown in Figure 6, we also propose to use profiling to simplify the update of the variable 'skip_factor'. In the profiling process, the compiler sets the 'skip_factor' to a fixed value from the set {32, 64, 96, 128, 160, 192, 224} and selects the one with highest performance during test runs. This way, the periodic update of 'skip_factor' can be removed from the CPU pre-execution code and there is also no need for a new instruction to access the L3 cache hits for the GPU. The CPU pre-execution code with a fixed 'skip factor' is shown in Figure 7, from which we can see that the code related to 'batch_size' is also removed.

4.3. Generating the CPU Pre-Execution Code from HWST GPU Kernels

HWST GPU kernels contain one or more loops, which contain global memory access instructions. We refer to such a loop as a kernel loop. Among our benchmarks, MM, Con, TMV, MV, and MC are of the HWST type. To generate the CPU pre-execution program for an HWST GPU Kernel, we process one kernel loop at a time. For each kernel loop, a CPU function is generated, which contains the global

memory access instructions and the address computation operations in the loop body. Both the thread id and the kernel loop iterator are replaced with function input parameters. If the kernel loop is a nested one, the iterators from all loop levels are replaced with function parameters. Our proposed compiler algorithm is shown in Figure 8.

```
float cpu_prefetching( … ) {
  unroll_factor = 8; skip_factor = 160;
  //added loop to traverse thread blocks
  for (j = 0; j < N_tb; j+= concurrent_tb)  {
    //added loop to traverse concurrent threads
    for (i = 0; i < concurrent_tb*tb_size ;
        i+=skip_factor*unroll_factor)  {
      int thread_id=i+skip_factor*unroll_factor+j* tb_size;
      // unrolled loop
      float a0=memory_fetch_for_thread (
          thread_id+ skip_factor *0);
      float a1=memory_fetch_for_thread (
          thread_id+ skip_factor *1);
      ......
      sum += a0+a1+a2+a3+a4+a5+a6+a7; /* operation
          inserted to overcome dead code elimination */
  } } }
```

Figure 7. The CPU pre-execution code for the vector-add GPU kernel with a fixed skip factor of 160.

```
1. For each kernel loop, extract memory operations and
the associated address computations from the loop body
and put them in a CPU function, replace thread id
computation with an input parameter, and replace the
kernel loop iterators with input parameters.
2. Add a nested loop structure into the CPU code to
prefetch data for concurrent threads.
  a. The outer loop traverse through all TBs. The
  iterator starts from 0 and the loop bound is the
  number of thread blocks in the GPU program. The
  iterator update is the number of concurrent TBs,
  meaning the number of TBs that can run concurrently
  on the GPU.
  b. The second-level loop corresponds to the kernel
  loop and we use the same loop bound. The loop
  update is increased to unroll the next level loop. If the
  kernel loop is nested, this second-level loop is also
  nested.
  c. The third-level loop traverses through concurrent
  threads. The loop iterator starts from 0 and the loop
  bound is the number of concurrent threads (which is
  the product of TB size and the number of concurrent
  TBs). The iterator update is set as a product of three
  parameters, unroll-factor, batch_size, and skip_factor.
```

Figure 8. The compiler algorithm to generate CPU pre-execution program from HWST GPU kernels.

As shown in Figure 8, after generating the function to load data for one loop iteration of a kernel loop (i.e., step 1), we insert loops to prefetch data for many concurrent threads. Similar to LWST kernels, the outer loop is used to prefetch data for concurrent TBs. Before going through concurrent threads, however, we insert the second-level loop to account for the kernel loop structure. The third-level loop traverses through all concurrent threads, similar to step 2b in Figure 2 for LWST kernels. We illustrate our algorithm using the simplified version of transpose-matrix-vector multiplication. The GPU kernel and the generated CPU pre-execution program are shown in Figure 9.

As shown in Figure 9, the CPU function 'memory_fetch_for_thread_loop_1' is generated from the loop body of the GPU kernel and the loop iterator and thread id are replaced with function parameters. In CPU function 'cpu_prefetching', the second-level loop (with iterator '*m*') corresponds to the kernel loop. The iterator update is a fixed value 8 rather than 1 so as to unroll the loop body for 8 times. The third level loop (with iterator '*i*') traverses through concurrent threads for prefetching. The reason for such loop organization is that GPU executes many threads in parallel. Therefore, instead of prefetching data of multiple iterations for one thread, we prefetch data of one iteration for many threads before moving on to the next iteration. The 'skip_factor' update part in Figure 9 is the same as discussed in Section 4.2 and both adaptive and profiling approaches can be applied.

From our algorithms shown in Figures 2 and 8, we can see that the granularity of our CPU prefetch function is one loop iteration, with LWST as a special case of HWST. One may suggest finer granularity such as prefetching one memory access a time. In other words, the CPU fetches one datum (e.g., A[n] in Figure 3) for many threads before moving on the next (B[n] in Figure 3) rather than fetching all the data in one iteration for a thread (A[n], B[n], and C[n] in Figure 3). We do not choose this approach since it requires the CPU and GPU to follow the exactly same access order and the GPU compiler is more likely to re-order the accesses within a loop body than to reorder accesses across different loops. Furthermore, using one CPU function call to issue one access incurs too much control overhead for CPU execution.

Note that the algorithms in Figures 2 and 8 assume that the TBs are dispatched to SMs in-order, which is the case based on our experiments on current discrete GPUs. If out-of-order TB dispatch is used, our scheme would require GPU to send the active TB ids to the CPU and the prefetching is done accordingly for those active blocks. In other words, we will replace the implicit block ids in the loop "for (j = 0; j < N_tb; j+= concurrent_tb)" (step 2a in Figure 2 and Figure 8) with explicit ones forwarded from the GPU. Our sequential dispatch assumption eliminates such GPU-to-CPU communication.

```
__global__ void tmv_naive(float* A, float* B,
      float* C, int width, int height) {
  int x = blockIdx.x *blockDim.x+threadIdx.x;
  float sum = 0;
  for (int i=0; i<height; i++) {
    sum += A[i*width+x]*B[i];
  }
  C[x] = sum; }            (a)
```

```
float memory_fetch_for_thread_loop_1 (int n, int m)
{ // n is the thread id and m is the loop iterator
  return (A[m*width+n] + B[m]); /* A, B are the
        CPU pointers mapped to the GPU memory */
}
float cpu_prefetching( ... ) {
  unroll_factor = 8;
  //added loop to traverse thread block
  for (j = 0; j < N_tb; j+= concurrent_tb)  {
    //the loop corresponding to kernel loop
    for (m = 0; m < loop_counter_in_thread; m+=8) {
      //added loop to traverse concurrent threads
      for (i = 0; i < concurrent_tb*tb_size;
          i+=skip_factor*batch_size*unroll_factor)
      for(k = 0; k < batch_size; k++) {
        int thread_id = i + k*skip_factor*unroll_factor
            + j*tb_size;
        // unrolled loop
        float a0 = memory_fetch_for_thread_loop_1 (
            thread_id+ skip_factor*0, m+ 0);
        float a1 = memory_fetch_for_thread_loop_1 (
            thread_id+ skip_factor*1, m+ 0);
        ......
        sum+=a0+a1+a2+a3+a4+a5+a6+a7; /*operation
        inserted to overcome dead code elimination */}
      for (i = 0; i < concurrent_tb*tb_size;
          i+=skip_factor*batch_size*unroll_factor)
      for(k = 0; k < batch_size; k++) {
        int thread_id = i + k*skip_factor*unroll_factor
            + j*concurrent_tb*tb_size;
        float a0 = memory_fetch_for_thread_loop_1 (
            thread_id+ skip_factor *0, m+ 1);
        float a1 = memory_fetch_for_thread_loop_1 (
            thread_id+ skip_factor *1, m+ 1);
        ......
        sum+=a0+a1+a2+a3+a4+a5+a6+a7;  }
      ...
      // Updating skip factor (See Section 4.2)
} } }                    (b)
```

Figure 9. A code example for HWST. (a) A Transpose Matrix Vector Multiplication GPU kernel; (b) the (partial) CPU pre-execution program.

5. Experimental Results

5.1 Performance of CPU-Assisted GPGPU

After the CPU pre-execution program is generated, we let the CPU to execute this program right after the GPU kernel is launched. In our first experiment, we

examine the performance improvements achieved with our proposed CPU-assisted execution. In Figure 10, we report the GPU performance using instruction per cycle (IPC) for each benchmark for three configurations, no CPU pre-execution (labeled 'no-preex'), CPU pre-execution with adaptive update of skip factor (labeled 'adaptive'), CPU pre-execution with a fixed skip factor determined from profiling (labeled 'profiling'). Since our GPU has 4 SMs and each SM has 32 scalar SPs, the peak IPC is 128. We also include the GPU performance results for a perfect L3 cache (labeled 'perfect L3) in Figure 10 for reference.
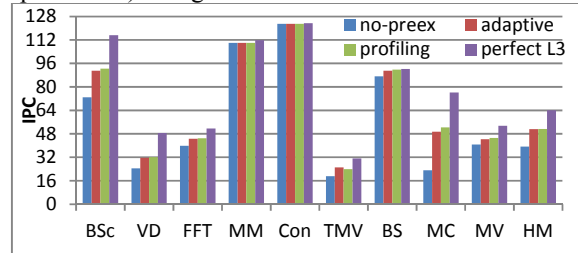

Figure 10. GPU performance comparison among no-pre execution (no-preex), CPU pre execution with adaptive update of 'skip_factor' (adaptive) and CPU pre execution with a fixed 'skip_factor' determined from profiling (profiling)

Form Figure 10, we can see that our proposed CPU pre-execution improves performance significantly, up to 113% (MC) and 21.4% on average with adaptive update of 'skip_factor' and up to 126% and 23.1% on average using a fixed 'skip_factor' determined from profiling. Among these benchmarks, BSc, VD and TMV are memory intensive and we achieve about 30% speedups. The high performance gains from MC are due to the fact that the GPU kernel (without CPU pre-execution) suffers from partition conflicts [9] while our pre-execution exploits the partition-level parallelism of off-chip memory when it prefetches data across multiple TBs. As the GPU requests hit in L3 cache, they do not go to off-chip memory, thereby avoiding the partition conflicts. The speedups for BS, MV, FFT are from 4% to 11% due to their irregular address patterns and cache conflicts. There are no performance benefits for MM and Con because they are highly optimized and have good locality and data reuse in L1 and L2 cache of GPU, which makes the L3 cache not critical. Even with a perfect L3 cache, the performance gains are negligible for these two benchmarks.

Another observation from Figure 10 is that both adaptive update of 'skip_factor' and a fixed 'skip_factor' selected from profiling are effective in improve the GPGPU performance. The profiling approach is slightly better as the adaptive approach

usually quickly converges to the optimal value, as shown in Figure 6.

5.2 The Efficacy of Data Prefetching using CPU Pre-execution

In this experiment, we examine the efficacy of data prefetching using CPU pre-execution. First, we evaluate the coverage of this prefetching scheme by examining the L3 cache hit rate for GPU accesses with and without CPU prefetching. The results are shown in Figure 11. The hit rates for GPU execution without CPU pre-execution is labeled 'no-preex' and GPU execution with CPU pre-execution using the adaptive update of 'skip_factor' is labeled 'adaptive'. The results for CPU pre-execution using fixed 'skip_factor' are very close to adaptive update. From Figure 11, we can see that the L3 cache hit rates are highly improved by CPU pre-execution. On average, it improves from 12.9% to 39.2%. The L3 cache hit rate improvements for MM and Conv do not translate to performance gains as shown in Figure 10. The reason is that TLP and higher level of caches provide sufficient latency hiding for these benchmarks.
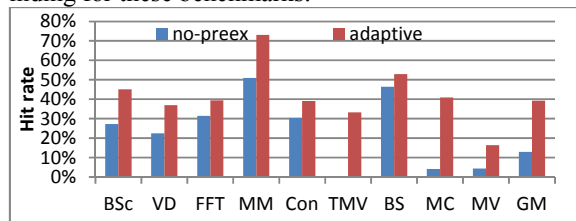

Figure 11. L3 hit rate for GPU execution without execution (no-preex) and with CPU pre execution (adaptive).

Another metric for data prefetching is accuracy and we evaluate it by computing the ratio of how many L3 misses generated from CPU pre-execution are actually accessed by GPU threads and the results are shown in Figure 12. It can be seen from the figure that our proposed CPU pre-execution has very high accuracy. On average, 98.6% data blocks loaded from the memory by CPU are accessed by the GPU threads.
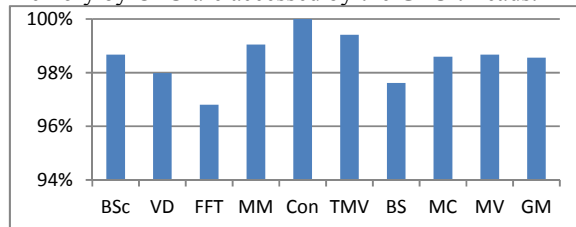

Figure 12. Prefetch accuracy of CPU pre-execution.

Since our proposed CPU pre-execution needs to execute instructions to generate prefetching requests for GPU threads, one way to evaluate the overhead of

our approach is to examine how many instructions the CPU needs to execute in order to achieve the performance gains. In Figure 13, we report the ratio of the number of instructions executed by the CPU over the number of instructions executed by GPU for both adaptive update of 'skip_factor' (labeled 'adaptive') and fixed 'skip_factor' selected using profiling (labeled 'profiling'). From Figure 13, we can see that the performance gains shown in Figure 10 are achieved with little instruction overhead. On average, our CPU assisted GPGPU using adaptive update of 'skip_factor' (fixed value of 'skip_factor') only executes 0.74% (0.69%) extra instructions to deliver the performance gains.
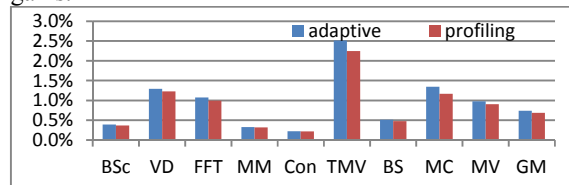

Figure 13. The ratio of (number of instruction executed by CPU / number of instruction executed by GPU).

5.3 Understanding the Impact of GPU Architectures

In this experiment, we vary the following GPU architecture parameters to understand the impact on our CPU-assisted GPGPU, the GPU SP frequency, the off-chip memory frequency, and the number of SPs in an SM. First, we vary the GPU SP frequency from the default 480 MHz to 267 MHz and 800 MHz. The CPU frequency remains at 2.4 GHz and the DRAM bandwidth remains at 19.2GB/s. In Figure 14, we report the speedups that are achieved from CPU assisted execution for each SP frequency compared to no CPU pre-execution (labeled 'sp267_speedup', 'sp480_speedup', and 'sp800_speedup'). All CPU pre-execution uses adaptive update of 'skip_factor' in the experiments in this section and the fixed 'skip factor' has slightly better performance gains. The results labeled 'sp480_speedup' are what reported in Figure 10.
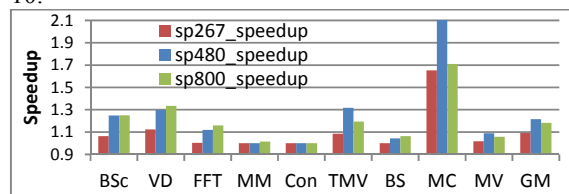

Figure 14. The speedups from CPU pre-execution for GPUs running at different frequencies and the normalized execution time without pre-execution.

From Figure 14, we can see that when SP frequency is reduced, the relative memory latency is also reduced. Therefore, CPU pre-execution provides less performance gains. On the other hand, when we increase SP frequency, these benchmarks show different trends. First, the benchmarks, VD, FFT, BSc, have higher performance gains as the memory latency becomes more significant. Secondly, for the benchmarks, TMV, MV, and MC, the impact is opposite and the reason is that the GPU SPs nearly double the rate of its memory requests, which enforce the CPU to skip more threads and to prefetch less data. For the benchmark BS, its baseline IPC is very high (close to 90) when SP frequency is 480 MHz, thereby limiting the pre-execution impact as shown in Figure 10. Overall, increasing the speed of SPs has less impact than decreasing the speed and CPU pre-execution is still effective for all these different SP speeds.

Next, we vary the off-chip memory frequency from the default 600 MHz to 300 MHz and 1200 MHz and the results are show in Figure 15. From Figure 15, we can see that when memory frequency is increased, the memory latency is reduced. Therefore, CPU pre-execution provides less performance gains. When we reduce memory frequency, these trends of these benchmarks are similar to increasing the SP frequency. For example, for BSc, VD and FFT, the CPU pre-execution shows better speedups when the memory frequency is reduced from 600 MHz to 300 MHz, because the memory latency dominates these three benchmarks. For TMV, MV, and MC, memory latency does not dominate the execution time. Therefore, the speedups of CPU pre-execution are reduced when the memory frequency is reduced from 600 MHz to 300 MHz. Overall, reducing the speed of memory has much less impact than increasing the memory speed and our proposed CPU pre-execution is effective for all these different memory speeds.
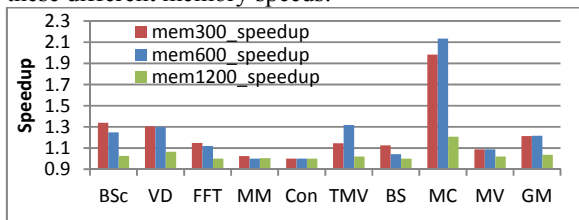

Figure 15. The speedups from CPU pre-execution for off-chip memory running at different frequencies and the normalized execution time without pre-execution.

Then, we vary the number of SPs in an SM and keep the same (4) SMs in our GPU model. In Figure 16, we report the speedups that are achieved from CPU assisted execution for each SM configuration

compared to no CPU pre-execution (labeled 'w16_speedup', 'w32_speedup', and 'w64_speedup'). In our baseline GPU configuration, each SM has 32 SPs. From the figure, we can see that when the number of SPs is reduced in an SM (while keep the same number of SMs in the GPU), the GPU becomes more latency tolerant as each instruction in a warp will take more cycles to finish. Given the same application, reducing the number of SPs is equivalent to increasing TLP, thereby reducing the performance gains achieved from CPU pre-execution. On the other hand, increasing the number of SPs also increases the rate of their memory requests, similar to increasing the SP speed, which can also reduce the effectiveness of CPU pre-execution. Nevertheless, on average, for these three SM configurations, CPU pre-execution achieves 14.7%, 21.4%, and 12.4% performance improvement, respectively.
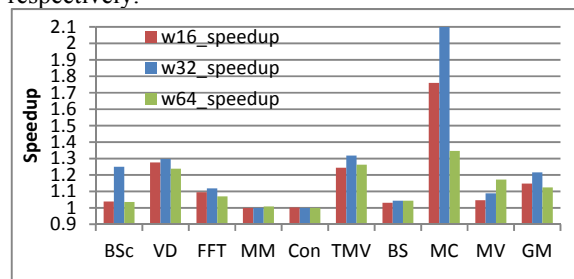

Figure 16. The speedups from CPU pre-execution for different numbers of SPs in an SM.

5.4 Sensitivity of the Parameters in CPU Pre-Execution Program

In this experiment, we study the sensitivity of the two parameters used in our CPU pre-execution program to update the variable 'skip_factor' (see Section 4.2). The first is the 'batch_size', which determines how often the skip_factor is updated. We vary this variable from 8, 16, and 32 and the GPU performance results are shown in Figure 17. As seen from the figure, although the batch size of 16 achieves the best performance, the performance difference for different batch sizes is limited, except for BSc, which prefers a large batch size.
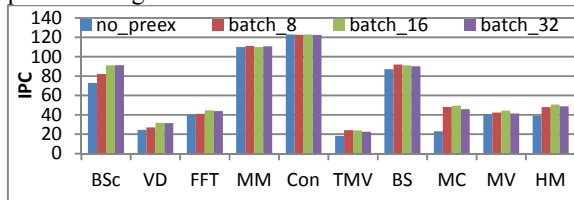

Figure 17. The GPU performance for CPU pre-execution using different batch_sizes.

In another experiment, we also change the threshold used to determine whether there are too many or too few L3 cache hit. We vary the threshold from 10 to 50 and the results are nearly identical, showing that our algorithm is not sensitive to this parameter.

5.5 Using CPU to Execute GPU Threads

In this experiment, we consider the option of using the CPU to directly execute some GPU threads to reduce the GPU workload. On the GPU side, the thread blocks are distributed to SMs based on the order of thread block id from small to large. On the CPU side, the CPU executes the thread blocks from the opposite direction, starting from the one with the largest thread block id. In our simulator, we implemented a special instruction for the CPU to get the largest thread block id issued in the GPU. This way, we ensure that there is no overlap workload between the CPU and the GPU. The speedups of such workload distribution between CPU and GPU over GPU-only execution are shown in Figure 18. From the figure, we can see that the performance gains of most of benchmarks are less than 2%. The main reason is the limited floating-point throughput of the CPU and the high overhead of CPU to access GPU memory partition. Among the workloads, the benchmark, VD, shows the highest (about 5%) speedup since it does not have many ALU operations to expose the ALU bottleneck of the CPU.
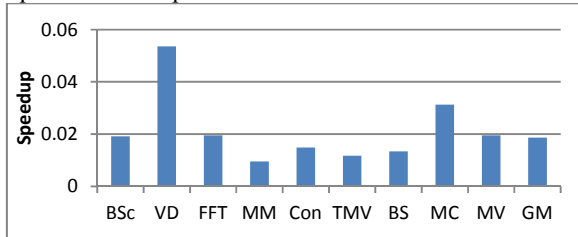

Figure 18. The speedups of workload distribution between GPU and CPU over GPU-only execution.

6. Related work

Although a key design philosophy of GPU is to use TLP to hide long memory access latency, the importance of GPU memory hierarchy has been widely recognized to achieve high performance GPGPU. In [23], software prefetching is used to overlap memory access latency with computations. However, prefetching data into registers or shared memory increases the register pressure and may hurt the performance due to reduced TLP [27]. In [12], Lee et al. proposed many-thread aware GPU prefetching approaches for L1 cache. Besides leveraging the well-known stride access pattern, they revealed the

interesting insight that when a workload is parallelized into many threads, each thread may be short and inter-thread/inter-warp prefetching is more effective than intra-thread/intra-warp prefetching. Compared to this work, our proposed CPU pre-execution does not rely on stride access patterns and provides both intra- and inter-warp prefetching. More importantly, all the previous works on GPU prefetching do not fit well with fused architectures as both demand cache misses and prefetches compete for critical resource, such as L2 cache miss handling status registers (L2 MHSRs), on the GPU side while leaving the CPU side resource idle. Our proposed approach, in contrast, leverages such critical resources on CPU side for prefetches and keeps those on GPU side for demand misses, thereby achieving better resource utilization. We also implemented the per-PC stride prefetcher with enhanced warp id indexing [12] in our simulator, which shows a 5.24% speedup on average.

To take advantage of fused architectures, it is proposed in [7] that the GPUs run prefetching algorithms to prefetch data for CPU programs. In comparison, our goal is to accelerate GPU programs and we believe it is a better fit to fused architectures since both GPU and CPU are used to do what they are good at: GPU for ALU/floating-point computations and CPU for flexible and accurate data prefetching.

Our proposed CPU-assisted GPGPU is also inspired from many works on CPU-based pre-execution [2][4][6][8][10][11][17][18][20][24], in which a pre-execution thread is used to provide data prefetching and/or accurate control flow to the main thread. The novelty of our work is to use a single CPU thread to prefetch data for many concurrent GPU threads and a simple yet effective way to control how far the pre-execution thread runs ahead.

7. Conclusion

In this paper, we propose to collaboratively utilize CPU and GPU resources for GPGPU applications. In our scheme, the CPU runs ahead of GPU threads to prefetch the data into the shared L3 cache for the GPU. Novel compiler algorithms are developed to automatically generate CPU pre-execution programs from GPU kernels. We also provide flexible mechanisms to control how far the CPU runs ahead of GPU threads. Our experimental results show that our proposed CPU pre-execution has very high prefetching accuracy and achieves significant performance gains at the cost of minor instruction overhead from the CPU side. Furthermore, our results show that the proposed scheme remains effective for different GPU configurations.

Acknowledgements

References

[1] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator. IEEE International Symposium on Performance Analysis of Systems and Software, April 2009.

[2] A. Roth and G. Sohi, Speculative data driven multithreading. IEEE International Symposium on High Performance Computer Architecture, 2001.

[3] AMD Accelerated Parallel Processing (APP) SDK V2.3, http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx, 2011

[4] C. K. Luk, Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. International Symposium on Computer Architecture, 2001.

[5] C. Luk, S. Hong, H. Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. IEEE/ACM International Symposium on Microarchitecture, 2009.

[6] C. Zilles and G. Sohi, Execution-based prediction using speculative slices. International Symposium on Computer Architecture, 2001.

[7] D. H. Woo, H. S. Lee, COMPASS: a programmable data prefetcher using idle GPU shaders. Proceedings of the Architectural support for programming languages and operating systems, March 13-17, 2010.

[8] D. Kim and D. Yeung, Design and evaluation of compiler algorithms for pre-execution. Proceedings of the Architectural support for programming languages and operating systems, 2002.

[9] G. Ruetsch and P. Micikevicius, Optimize matrix transpose in CUDA. NVIDIA, 2009.

[10] H. Zhou, Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Sept. 2005

[11] J. D. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, Speculative precomputation: long range prefetching of delinquent loads. International Symposium on Computer Architecture, 2001.

[12] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, Many-thread aware prefetching mechanisms for gpgpu applications. IEEE/ACM International Symposium on Microarchitecture, 2010.

[13] MARSSx86, http://marss86.org/~marss86/index.php/Home

[14] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, High performance discrete Fourier transforms on graphics processors. Proceedings of Supercomputing, 2008.

[15] NVIDIA CUDA C Programming Guide 3.1, 2010.

[16] NVIDIA GPU Computing SDK 3.1, http://developer.nvidia.com/gpu-computing-sdk, 2011.

[17] O. Mutlu, J. Stark, C. Wilkerson and Y. N. Patt, Run ahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. IEEE International Symposium on High Performance Computer Architecture, February 2003.

[18] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, Dynamically allocating processor resources between nearby and distant ILP. International Symposium on Computer Architecture, 2001.

[19] P. Boudier, Memory System on Fusion APUs - The Benefits of Zero Copy. AMD fusion developer summit, 2011.

[20] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation. IEEE International Symposium on High Performance Computer Architecture, 2002.

[21] Sandy Bridge, http://en.wikipedia.org/wiki/Sandy_Bridge.

[22] S. I. Lee, T. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In Proc. Workshops on Languages and Compilers for Parallel Computing, 2003

[23] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu, Optimization space pruning for a multi-threaded GPU. International Symposium on Code Generation and Optimization, 2008.

[24] Y. Solihin, J. Lee and J. Torrellas, Using a user-level memory thread for correlation prefetching, ISCA 2002

[25] Streaming SIMD Extensions, http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

[26] The AMD Fusion Family of APUs, http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.

[27] Y. Yang, P. Xiang, J. Kong and H. Zhou, A GPGPU Compiler for Memory Optimization and Parallelism Management. ACM SIGPLAN conference on Programming Language Design and Implementation, 2010.