A Case for a Flexible Scalar Unit in SIMT Architecture

Yi Yang[#], Ping Xiang[&], Mike Mantor*, Norman Rubin*, Lisa Hsu*, Qunfeng Dong[$], Huiyang Zhou[&]

[#] Dept. of CSA, NEC Labs, Princeton, NJ, USA, yyang@nec-labs.com

*AMD Inc., Orlando, FL, USA, {Michael.Mantor, norman.rubin, Lisa.Hsu}@amd.com

[$] Dept. of Central Hardware, Huawei, Shanghai, China, qunfeng.dong@ieee.org

[&]Dept. of ECE, NCSU, Raleigh, NC,USA, {pxiang, hzhou}@ncsu.edu

Abstract—**The wide availability and the Single-Instruction Multiple-Thread (SIMT)-style programming model have made graphics processing units (GPUs) a promising choice for high performance computing. However, because of the SIMT style processing, an instruction will be executed in every thread even if the operands are identical for all the threads. To overcome this inefficiency, the AMD's latest Graphics Core Next (GCN) architecture integrates a scalar unit into a SIMT unit. In GCN, both the SIMT unit and the scalar unit share a single SIMT-style instruction stream. Depending on its type, an instruction is issued to either a scalar or a SIMT unit. In this paper, we propose to extend the scalar unit so that it can either share the instruction stream with the SIMT unit or execute a separate instruction stream. The program to be executed by the scalar unit is referred to as a scalar program and its purpose is to assist SIMT-unit execution. The scalar programs are either generated from SIMT programs automatically by the compiler or manually developed by expert developers. We make a case for our proposed flexible scalar unit through three collaborative execution paradigms: data prefetching, control divergence elimination, and scalar-workload extraction. Our experimental results show that significant performance gains can be achieved using our proposed approaches compared to the state-of-art SIMT style processing.**

Keywords-GPGPU; scalar unit; SIMT; Vector unit

I. INTRODUCTION

With the availability and the single-instruction multiple-thread (SIMT) style programming models such as CUDA and OpenCL, GPGPU (General purpose computation on GPUs) becomes widely adopted for high performance computing. Although SIMT programming is easier than vector programming for developers to exploit data-level parallelism (DLP) in their applications, one significant deficiency of SIMT processing is that every thread needs to execute the instructions even if they have the same operand values across all the threads. In other words, scalar operations in nature are forced to be executed in all threads due to the SIMT-style processing. To address this issue, the AMD's Graphics Core Next (GCN) architecture adds a scalar unit to a SIMT unit. Although the SIMT unit is referred to as a vector unit in GCN literature, it uses the typical SIMT architectural design pattern [26]. A single instruction stream is shared by the scalar unit and the SIMT unit. Depending on its type, an instruction is issued to either unit to be executed. The compiler is responsible to generate instructions with a proper type.

In this paper, we propose to extend the capability of the scalar unit so that it can either share an instruction stream with a SIMT unit or execute its own instruction stream, referred to as a scalar program. The purpose of scalar programs is to assist SIMT execution in a flexible way. We present three collaborative execution paradigms between the scalar unit and the SIMT unit and propose the compiler algorithms to generate the scalar code from SIMT programs. First, we adapt a recently proposed algorithm [40] to generate scalar programs for a scalar unit to prefetch data for SIMT units. Second, we propose a software approach to eliminate/reduce control divergence in SIMT units. In this approach, the SIMT code is slightly altered to communicate control flow information of each thread to a scalar unit. The scalar program reorganizes/remaps the threads so as to minimize control divergence. Third, we extract scalar workload from SIMT programs to generate scalar programs. Our experimental results show that that significant performance gains can be achieved using our proposed approaches compared to current SIMT-style processing.

The remainder of the paper is organized as follows. In Section II, we present a brief background on SIMT and GCN architecture. We discuss our proposed architecture and the collaborative execution programming model in Section III and Section IV, respectively. Section V describes the experimental methodology. Section VI presents the execution paradigm of using a scalar unit to prefetch data for SIMT units. Section VII presents control-flow divergence elimination using a scalar unit. Section VIII presents scalar-operation extraction. Related work is discussed in Section IX. Section X concludes the paper.

II. BACKGROUND

Modern GPUs feature many-core architecture. In AMD/NVIDIA GPUs, a GPU has a number of compute units (CUs)/streaming multiprocessors (SMs). Each SM/CU in turn has multiple streaming processors (SPs)/processing elements (PEs). Each SM/CU has its register files, shared memory, and an L1 data cache.

Modern GPUs use the single-instruction multiple-thread (SIMT) programming model. A GPGPU program is referred to as a kernel. All threads for the same kernel follow the single-program multiple-data (SPMD) execution model and the thread identifier (id) is used to determine the workload for each thread. These threads are organized in a hierarchy. On AMD/NVIDIA GPUs, 64/32 threads are grouped in a wavefront/warp. The threads in the same warp/wavefront are executed in the single-instruction multiple-data (SIMD) manner, which means that these threads share the same program counter (PC) and execute instructions in locksteps. Multiple warps/wavefronts in turn form a thread block/workgroup. The threads in the same thread block/workgroup can communicate with each other through shared memory. Depending on the resource usage of a thread block, an SM can execute one or more thread blocks concurrently. A high number of concurrent threads can effectively hide instruction execution latencies and reduce the adverse impact of long-latency off-chip memory accesses.

The SIMT programming model is a key reason for the popularity of GPGPU as it enables programmers to manage data-level parallelism (DLP) in a similar manner to task-level parallelism. In a kernel, a programmer just needs to specify the operations for a single thread. The GPU hardware assembles threads into warps/wavefronts, which are essentially vectors. Such transparent vectorization hides significant complexity of classical vector programming. One deficiency of the SIMT-style instruction processing, however, is that the GPU hardware vectorizes all the instructions even if some are scalar operations in nature. This issue can be illustrated using a vector-add example shown in Fig. 1.

As shown in Fig. 1a, using the vector programming model, vector instructions are used only on vector operands. In contrast, there is no distinction between vector and scalar operations in the SIMT code, as shown in Fig. 1b. All instructions in the kernel are vectorized. In other words, although the computation '*blockIdx.x*blockDim.x*' is the same for all the threads in a thread block, the GPU hardware will carry out this computation for all the threads.

We experimentally verified this observation on the latest NVIDIA Kepler GPUs. A thread-id independent kernel, which performs some computations based on its thread block id, consumed the same amount of power (also the same performance) as a thread-id dependent kernel, which carries out the same computations based on its thread id.

To address this limitation, in AMD's GCN architecture, a scalar unit is introduced into each CU. When the compiler generates machine code, it specifies whether an operation is a scalar or vector one. When an instruction is fetched for a warp/wavefront, the warp scheduler issues it to either the scalar unit or the multiple SPs (i.e., the SIMT/vector unit) depending on its type. The scalar unit has its own register file and the communication between the scalar and the SIMT/vector unit is done explicitly through register-move instructions.

Although GCN improves the efficiency of SIMT processing, several critical bottlenecks remain. First, long off-chip memory access latencies require extremely high degrees of thread-level parallelism (TLP). The recent trend of GPU development, which has been integrating more SPs into an SM as evidenced in the NVIDIA Kepler architecture, further increases the demand for TLP. Second, as both SIMT and vector processing rely on SIMD execution to achieve energy efficiency, control divergence in a warp/wavefront/vector will result in idle SPs (or idle execution lanes), severely affecting the efficiency of SIMD execution. Furthermore, GCN has not completely eliminated vectorization on scalar operations. The reason is that although the operation such as '*blockIdx.x*blockDim.x*' becomes scalar in GCN for each warp/wavefront, different warps/wavefronts in the same thread block/workgroup still need to perform the same computation redundantly. In the next section, we propose our solutions to these bottlenecks.

```
for(i = 0; i < n; i += vlen) //scalar operations
    C[i:i+vlen] = A[i:i+vlen] + B[i:i+vlen]; //vector operations
```

(a) Vector-add in the vector programming model

```
int base = blockIdx.x* blockDim.x;
int idx = base+threadIdx.x;
C[idx] = A[idx]+B[idx];
```

(b) Vector-add in the SIMT programming model

Figure 1. C-like pseudo code for vector-add using the vector and the SIMT programming models.

III. A Flexible Scalar Core in a SIMT Unit

Similar to the GCN architecture, our proposed architecture contains both a scalar unit and a SIMT/vector unit, as shown in Fig. 2. The key difference from the GCN architecture is that the scalar unit can fetch and execute its own instruction stream. Both the scalar unit and the SIMT unit support multithreaded execution. The scalar unit can run multiple scalar threads as long as there are sufficient scalar registers. The SIMT units can run multiple warps and the (vector) register file and the shared-memory usage determines the number of concurrent warps in each SIMT unit. In other words, both the scalar unit and the SIMT unit support the SIMT-style instruction processing. Similar to the warp scheduler in the SIMT unit, which manages concurrent multiple warps and issues (vector) instructions, we introduce a scalar-thread scheduler to manage multiple concurrent scalar threads and issue scalar instructions to the scalar pipeline. The instruction multiplexer (I-MUX) in Fig. 2 can also be configured such that scalar instructions are issued by the warp scheduler in the SIMT unit. With this configuration, our proposed architecture can operate in the same way as the GCN architecture, where the scalar operations are embedded in the SIMT/vector instruction stream and encoded with a scalar operation type.

The SIMT unit and the scalar unit can use the same RISC-style instruction set architecture to reduce the complexity for hardware design and the compiler support. Both scalar unit and the SIMT unit employ the in-order pipeline organization. As a result, the scalar unit looks very much like an SP/a vector lane in the SIMT unit except that it has its own decoder and instruction execution control logic.

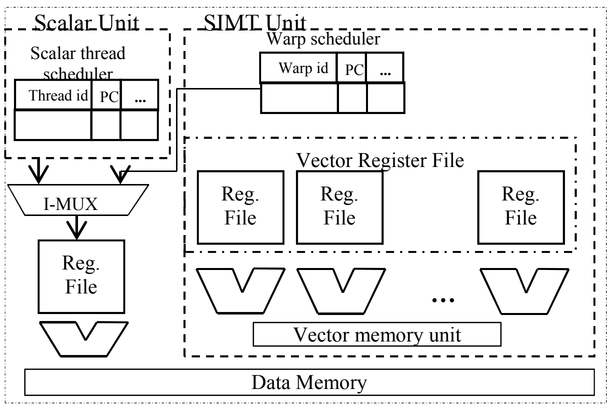

Figure 2. Proposed architecture containing a scalar and a SIMT unit.

The scalar unit shares the memory hierarchy with the SIMT unit and they can communicate with each other through shared memory or direct register move instructions. Shared memory also provides a synchronization primitive between the scalar unit and the SIMT unit, similar to the one used for synchronizing the warps in a thread block. The scalar unit also has its own load/store unit.

The thread block dispatcher is extended such that when a thread block to be dispatched, the scalar register is also checked to see whether there are sufficient scalar registers in the scalar register file. If not, the thread block will not be dispatched to the SM. In other words, in our proposed architecture, a thread block contains a thread block of SIMT threads (or vector threads) and a scalar thread. The scalar thread and its associated SIMT thread block do not need to wait for each other for completion although they start at the same time because of resource allocation.

Note that our proposed architecture is different from the fusion architectures [2] where GPUs and CPUs are integrated on the same die. In our design, a scalar unit is introduced in each SM/CU and is closely coupled to the SIMT unit whereas the CPUs and GPUs in fusion architectures communicate through shared memory space.

IV. A Collaborative Programming Model

In our proposed programming model, each program has two kernels: one SIMT kernel for the SIMT unit and one scalar kernel for the scalar unit. The purpose of the SIMT kernel is to provide high-throughput energy-efficient computation, the same as current SIMT/GPGPU kernels. The purpose of the scalar kernel is to execute scalar workloads and/or to facilitate the SIMT kernel as a helper thread. With this programing model, expert developers can program the scalar unit to leverage their knowledge of the applications. It also enables new compiler techniques to automatically generate helper threads or extract scalar workloads from SIMT kernels.

When a CPU launches a GPU program, it invokes both the SIMT kernel and the scalar kernel and lets both kernels to share the same input and output parameters. The thread grid and thread block dimensions remain the same as if there is no scalar kernel. In our programming model and our architecture design, we choose to have one scalar thread generated for each thread block of SIMT threads. Another possible choice is to have one scalar thread generated for one warp of SIMT threads. Our choice of one scalar thread per one thread block of SIMT threads makes it unlikely for the scalar unit to become the resource bottleneck to limit the number of thread blocks to be dispatched to an SM. Another way to view this is that a thread block in our programming model contains both a SIMT thread block, which contains multiple warps of SIMT threads, and a scalar thread block, which contains only one scalar thread.

To synchronize the scalar kernel and the SIMT kernel, we extend the existing *__syncthreads()* function, which is used to synchronize all the threads in a thread block. We add one parameter, '*k*', to the *__syncthreads(k)* function. The parameter is used to differentiate the synchronization among a thread block of SIMT threads from the synchronization among the scalar thread and all the SIMT threads in a thread block. If the parameter is 0, *__syncthreads(0)* is used to synchronize the SIMT threads in a thread block. Otherwise the synchronization happens among all the SIMT threads in a thread block and the corresponding scalar thread.

Next, we use the reduction primitive as an example to illustrate our proposed programming model. The scalar kernel, the SIMT kernel, and the CPU code to launch the GPU reduction program are shown in Fig. 3.

As shown in Fig. 3, the GPU reduction program contains two kernels: the scalar kernel '*reduction_scalar*' and the SIMT kernel '*reduction_SIMT*'. Both kernels have the same input and output parameters. Either has its own memory space. If the two kernels need to communicate through shared memory, the shared-memory variable should be declared as '*extern*', such as the variable '*sdata*' shown in Fig. 3. The size of the '*extern*' shared memory variable is declared in the third parameter '*smSize*' of the kernel invocation configuration. To ensure that both kernels accessing the same shared memory location, the same shared memory base address is assigned to the '*extern*' shared memory variable. In the main function, before the SIMT reduction kernel is invoked, we first set its scalar kernel as '*reduction_scalar*'. Then, the thread grid dimension and the thread block dimension are set up in the same way as for a typical GPGPU kernel. As shown in Fig. 3c, each thread block is configured to have 128 threads. The number of scalar threads per thread block is fixed as 1. This way, each thread block of the SIMT kernel shown in Fig. 3b computes the sum of 128 elements in the input array. It first loads data into shared memory, and then accumulates them into 4 elements in shared memory. Because the last few summation operations will only be performed by the first few threads in a thread block, we choose to use the scalar unit to take over such computation. The instruction *__syncthreads(1)*

```
__global__ void reduction_scalar(float* in, float* out) {
  extern __shared__ float sdata[];
  __syncthreads(1);
  float sum = sdata[0]+ sdata[1]+ sdata[2]+ sdata[3];
  out[blockIdx.x] = sum;}
```

(a) The scalar kernel, which performs the last two steps of the reduction operation.

```
__global__ void reduction_SIMT(float *in, float *out) {
  extern __shared__ float sdata[];
  unsigned int tid = threadIdx.x;
  sdata[tid] = in[blockIdx.x*BLOCK_SIZE + tid];
  for(int s=blockDim.x/2; s>2; s>>=1) {
    __syncthreads(0);
    int i = 2 * s * tid;
    if (i < blockDim.x) sdata[i] += sdata[i + s]; }
  __syncthreads(1); }
```

(b) The SIMT kernel, which performs all the reduction steps except the last two.

```
int main( int argc, char** argv) {
……
reduction_SIMT.set_scalar(reduction_scalar);
int grid = 1024; int threads = 128; int smSize = 128;
reduction_SIMT <<< grid, threads, smSize>>>(d_A, d_B);
……}
```

(c) The CPU code to launch the SIMT reduction kernels

Figure 3. The reduction example to illustrate our proposed programming model.

guarantees the four results are in shared memory before the scalar kernel shown in Fig. 3a finishes the rest work.

In our proposed programming model, a scalar thread can run independently to SIMT-vector threads. This feature enables a scalar thread to perform more tasks than just taking over scalar operations. In Sections VI, VII, and VIII, we present three collaborative execution paradigms for our proposed architecture: data prefetching, control divergence elimination, and scalar workload extraction. Before that, we first lay out our experimental methodology.

V. Experimental Methodology

We model our proposed architecture using GPGPUSim V3.1 [3] for performance evaluation. We also modify GPU-PowerSim [14] and McPAT [28] to derive the power consumption results of our proposed architecture. Our baseline SIMT architecture is modeled based on the NVIDIA GT 640 with GDDR5 memory [16]. The GT 640 GPU has 2 SMX (next generation SM), and each SMX has 192 SPs running at 0.95 GHz. The off-chip memory frequency is 2.5GHz and, the memory bus width is 128 bits. The memory bandwidth is 80GB/s due to the double rate of GDDR5. Each SMX has 4 warp schedulers and each scheduler can issue maximum 2 instructions every cycle. The pipeline width and the warp size are 32. To model our proposed architecture, we add one scalar unit into each SIMT pipeline, which contains 32 SPs. Therefore, there are six scalar units in each SMX. Shared memory of each SMX is configured as 48K bytes. While the overall size of L1 cache and shared memory in an SMX is 64K bytes, the Kepler provides additional 48K read-only data cache for each SMX. Considering most of input data of our applications are read-only, we set the L1 cache is set to 48K bytes. The L2 cache has 4 banks and the total size is 512K bytes. The register file in each SMX has 64K 32-bit registers. Each SMX can support up to 16 thread blocks and 2048 threads. The area overhead of the scalar unit, including the decoder, the fetch unit, the scalar register file (1K 32-bit registers), and the ALU, the load/store unit, is estimated as 5.4% for each SMX using GPU-PowerSim. The reason is that the major area is spent on caches, the vector register file, the vector memory unit, and the vector ALUs.

We generate scalar kernels and modify SIMT kernels using a source-to-source compiler [25] based on our proposed compiler algorithms. In the CPU code, we assign different streams generated by '*cudaStreamCreate*' to the scalar kernel and the SIMT kernel so that two kernels can start asynchronously. In this dual kernel execution model, the timing simulation of GPGPUsim waits for both kernels and then checks the availability of the resources on both the scalar and SIMT unit. If the resources can satisfy both a SIMT thread block and a single scalar thread, the scheduler will issue one thread block. The synchronization between threads from the SIMT kernel and the scalar kernel is supported by adding additional check based on the existing implementation of thread block synchronization. Because we do not change the NVIDIA compiler, we manually set value of 'id' of the instruction __syncthreads(id) in the PTX code, which GPGPUsim uses for functional and timing simulation.

TABLE I. Benchmarks used in our experiments

benchmark	Input size	Threads per TB	TB per SMX	Scalar register
N queen (NQU) [3]	12	96	3	5
CUDASEG (SEG) [8]	50*80*80	32*4	12	3
Laplace 3D (LPS) [3]	50*80*80	32*4	16	4
Ray tracing (RT) [3]	50*80	16*8	13	4
Neural Network (NN) [3]	28	1	10	8
Stream Cluster(SS) [5]	256*64k	256	16	6
Fast Fourier Trans. (FFT) [14]	1k*256	256	7	8 (Sec. VIII) 6 (Sec. VI)
Vector-add(VD) [31]	1M	128	16	2 (Sec .VIII) 6 (Sec. VI)
Reduction (RD) [31]	1M	128	16	6
Memory Copy (MC) [36]	1M	128	16	4
Transpose matrix vector multiply (TMV) [40]	64k*16	128	16	8
Scalar product (SP) [31]	256*4096	256	8	15
Histogram (HIST) [1]	1M	64	5	9
BitonicSort (BS) [1]	1M	64	16	4
Mandelbrot (MB) [31]	50*80	256	8	3
Discrete Cosine Transform (DCT) [31]	512*512	8*8	16	4

We choose a variety of benchmarks from NVIDIA SDK [31], Rodinia [5], GPGPUSim [3], AMD SDK [1] and open sources projects. The details of benchmarks are listed in Table 1. The benchmarks in Table 1 have different characteristics. Therefore, we use different subsets of them to evaluate our collaborative execution paradigms. In Section VI, we use the memory-bound benchmarks including FFT, VD, RD, MC, TMV, SP, DCT, and HIST to analyze the performance impact of data prefetching using the scalar unit. In Section VII, the benchmarks, NQU, SEG, LPS, MB, and RT, are used for control-divergence elimination as for these benchmarks, less than 24 threads per warp are active during more than 40% of execution cycles. Although the benchmarks, NN and SS, also have control divergence, they fit better with scalar workload extraction since there is only one thread active most of the time in a thread block. The three benchmarks, VD, BS and FFT, are also used for scalar-workload extraction as a comparison to the AMD GCN architecture. Therefore in Section VIII, scalar workloads are extracted from the benchmarks, NN, SS, FFT, BS, and VD, for more efficient execution.

In Table 1, we also include the register usage information of the scalar kernel generated for each benchmark. We can see that the scalar kernel only takes a small number of registers. As long as each SMX has a scalar register file with 128 scalar registers, the numbers of thread blocks per SMX for all the benchmarks are only limited by the SIMT kernels. Another potential resource bottleneck that may limit the number of concurrent threads on each SMX is shared memory. The execution paradigms for control divergence elimination (Section VII) and scalar workload extraction (Section VIII) introduce additional shared memory variables for communication between scalar thread and SIMT threads. To reduce the shared-memory usage, we choose to use 4 bytes of shared memory for each thread for data communication. Threads can exchange 4 bytes each time and such an exchange can be repeated multiple times to finish all the data that need to be transferred. This way, even an SMX hosts 2k concurrent threads, the additional shared memory usage is limited to 8kB. As each SMX has 48kB shared memory, such additional usage does not present as a resource bottleneck to our applications.

VI. Collaborative Execution Paradigm I: Data Prefetching

In this execution paradigm, we adapt a recent work on using CPU to prefetch data for GPU [40] to generate the scalar kernel. This collaborative execution paradigm is used when a performance profiler shows that a SIMT kernel under-utilizes the GPU memory bandwidth and suffers from stall cycles due to memory accesses. The key differences from the previous work [40] are two folds. First, in our proposed architecture, the scalar unit and the SIMT unit share the L1 cache and the L2 cache, which simplifies the control mechanism for prefetch timing. Second, one scalar thread prefetches for one SIMT thread block whereas in the previous work, a CPU thread prefetches data for all the thread blocks on the GPU.

Similar to [40], we classify GPU programs into two categories: the first category has no loops containing memory accesses; the second category has loops containing memory accesses. For the first category, since the scalar thread and the SIMT thread block start at the same time, there is little room for the scalar thread to run ahead to prefetch data in advance. Therefore, we choose to let the scalar thread prefetch the data for a future thread block of SIMT kernel. For the second category, the scalar thread will fetch data for the current thread block and it runs ahead of the SIMT threads by skipping loop iterations. Our algorithm is presented in Fig. 4.

As shown in step 1 in Fig. 4, if a SIMT kernel has no loops containing memory accesses, the memory instructions of the SIMT kernel are exacted and we introduce a loop to prefetch data for all threads in a future SIMT thread block, whose thread block id is computed as the sum of the current thread block id and the number of concurrent thread blocks that can run on the GPU. One such example is shown in Fig. 5, which includes the scalar and the SIMT kernel of the memory-copy benchmark. The SIMT kernel '*memcpy_SIMT*' does not have a loop containing memory accesses. In the scalar kernel, the variable '*k*' is updated with

```
1. If a SIMT kernel has no loop, extract its memory operations and add
   a loop to prefetch data for all the threads in a future SIMT thread
   block. The future thread block id is generated as a sum of the
   current thread block id and the number of concurrent thread blocks
   that can run on the GPU.
2. If a kernel contains a loop, which has memory accesses, create the
   same loop in the scalar kernel. In the loop body, extract its memory
   operations and add a loop to prefetch data for all the threads in the
   SIMT thread block. Insert the parameter to skip some iterations of
   the loop. To enable adaptive prefetching, insert code to use the
   clock function to check the number of cycles spent on memory
   fetch. If the number of cycles is less than the off-chip memory
   latency, it means the memory fetch hits in the cache, and we
   increase the number of skipped iteration to make the scalar unit run
   further ahead.
```

Figure 4. The compiler algorithm for generating a scalar kernel to prefetch data for a SIMT kernel

```
__global__ void memcpy_scalar(float* in, float* out) {
   int k = blockIdx.x + NUM_SM*TB_PER_SM;k = BLOCK_SIZE*k;
   if (k>== BLOCK_SIZE*gridDim.x) return;
   //loop over the threads in a thread block
   for(int i = 0; i < BLOCK_SIZE; i += 64)
         sum += in[k + i]; //prefetch instructions
}
__global__ void memcpy_SIMT(float *in, float *out) {
   unsigned int tid = threadIdx.x;
   int k = blockIdx.x*BLOCK_SIZE + tid;
   out[k] = in[k]; }
```

Figure 5. The scalar and SIMT kernels of memcpy. The scalar kernel prefetches data for a future SIMT thread block.

the product '*NUM_SM*TB_PER_SM*', which computes the number of concurrent thread blocks that can run on the GPU, to generate the future thread block id to be assisted by the scalar thread. The loop '*for(i = 0, i < BLOCK_SIZE; i += 64)*' is introduced to prefetch data for all the threads in the future SIMT thread block. The constant 64 is used as each cache line has 256 bytes and the type of data item to be accessed is '*float*' (i.e., 4 bytes). Note that memory copy is a well-known bandwidth limited workload. When running on our GT640 GPU model, it underutilizes the memory bandwidth due to the thread number limit (2048) for each SMX. The prefetches from the scalar thread improve the bandwidth utilization as well as the performance (see our discussion on Fig. 7).

For SIMT kernels with one or more loops containing memory access instructions, we generate a scalar thread to prefetch data for its SIMT thread block and let it bypass some loop iterations so that it can run ahead of the SIMT threads, as shown in step 2 of the algorithm in Fig. 4. In Fig. 6, we use a reduction kernel to illustrate the process. The kernel '*reduction_SIMT*' is the regular GPGPU program, in which each thread first accumulates multiple data from global memory into shared memory and then all threads in the same thread block perform the reduction on the shared-memory data. The function '*reduction_scalar*' is the scalar kernel performing data prefetching. It keeps the '*while*' loop as in the SIMT kernel. For memory accesses in the while loop, it adds another loop '*for(int i = 0; i < BLOCK_SIZE; i+= 64)*' to prefetch data for all the threads in its SIMT thread block. The variable '*SKIP*' is used to decide how far the scalar kernel can run ahead of the SIMT kernel. If the value of '*SKIP*' is increased, the scalar kernel will skip more iterations of the while loop. We use two clock functions to infer the latency of memory accesses. If it is less than the off-chip memory access latency, we consider that the prefetches hit in the cache. Therefore, we need to increase '*SKIP*' to let the scalar thread run faster. Here, we do not decrease the '*SKIP*' variable, because it only affects the current thread block. After a new thread block is dispatched, the '*SKIP*' variable will be initialized to the default value and updated independently in each thread block.

We evaluate the performance of this collaborative execution paradigm using a subset of the benchmarks in Table 1, which have relatively high cache miss rates. Note that high cache miss rates do not necessarily mean that memory accesses are the bottleneck. The reason is that a high degree of TLP can hide memory access latencies. The

```
__global__ void reduction_scalar(float* in, float* out, int n) {
   extern __shared__ float sdata[];
   int k = blockIdx.x*BLOCK_SIZE + 0;
   int SKIP = 2;   float sum = 0;
   while (k < n) {
      clock_t c0 = clock();
      //loop over the threads in a thread block
      for(int i = 0; i < BLOCK_SIZE; i+= 64)
             sum += in[k+i];   //prefetch instructions
      clock_t c1 = clock();
      if (c1-c0<400) SKIP++;
      k+= SKIP*BLOCK_SIZE*gridDim.x;
   }}
__global__ void reduction_SIMT(float *in, float *out, int n) {
  extern __shared__ float sdata[];
  unsigned int tid = threadIdx.x;
  float sum = 0;
  int k = blockIdx.x*BLOCK_SIZE + tid;
  while (k < n) {  sum += in[k];   k+=BLOCK_SIZE*gridDim.x; }
  sdata[tid] = sum;
  ........ // reduction on shared memory
           // and store in global memory
}
```

Figure 6. The scalar and SIMT kernels of reduction. The scalar kernel prefetches data for its SIMT thread block.

tradeoff is that the hardware needs to have enough resource to hold the thread contexts, which lead to a large register file and large shared memory. As we will show next, our prefetching scheme can significantly reduce cache miss rates, which in turn can reduce the high requirement on TLP and the associated hardware resources.

First, in Fig. 7, we evaluate the performance gains that are achieved using the scalar kernel to prefetch data for SIMT kernels. We can see that most of applications have more than 19% performance gains except vector-add (VD) and FFT. The reason it that these two benchmarks have high memory-level parallelism (MLP) resulting from both TLP and instruction-level parallelism (ILP). The benchmark VD has two independent memory loads in each SIMT thread. The benchmark FFT has four independent memory requests in each thread and each memory request is of the type 'float2'. Therefore, they utilize the off-chip memory efficiently and prefetching has little performance impact. Compared to VD and FFT, MC (memory copy) has no intra thread MLP. Since each SMX of Kelper can support only 2048 threads, considering the off-chip memory latency being 300 cycles, each SMX can only fetch 2048*4B =8 KB every 300 cycles. With the SMX frequency as 0.95 GHz, the bandwidth utilization per SMX is 8kB/(300*1ns/0.95), which is 25.9GB/s. As a result, the 80GB/s memory bandwidth of GT 640 cannot be fully utilized by two SMXes. The prefetches from the scalar unit improve the utilization of the bandwidth, leading to 19.3% performance gains. On average using the geometric mean (GM), our scalar unit-based prefetching mechanism achieves 19.3% performance improvement over our baseline GT 640 GPU by introducing only 0.73% instructions from the scalar unit.

We collect the energy consumption, including the static and dynamic energy consumption, of our proposed architecture with the baseline GT640 GPU. Our scalar unit introduces slightly more dynamic energy consumption and significantly reduces the static energy consumption due to

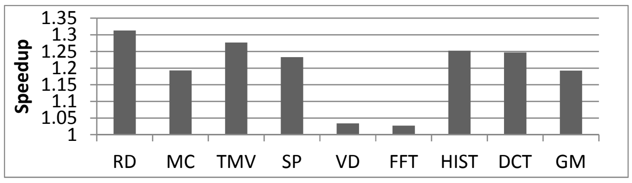

Figure 7. The speedups of using the scalar unit to prefetch data over the baseline without prefetching.

the reduced execution time. Overall, our approach shows up to 14.7% and an average 8.0% energy savings.

In Fig. 8, we report the cache miss rates for both L1 and L2 caches of the baseline (labeled 'L1 miss rate' and 'L2 miss rate') and our scalar unit-based prefetching scheme (labeled 'prf L1 miss rate' and 'prf L2 miss rate'). We can see that our prefetching mechanism significantly reduces the L1 and L2 cache miss rates. On average, the L1 cache miss rate is reduced from 65.2% to 55.4%, and the L2 cache miss rate is reduced from 47.0% to 9.6%.

Because the scalar unit and SIMT unit have independent pipelines, the frequency of scalar unit can be different from the SIMT unit. Therefore, we can reduce the frequency of scalar unit to reduce its dynamic power consumption. In Fig. 9, we show the speedups achieved with the scalar unit running at different frequencies. The first set of results in Fig. 9 (labeled '1') is the same speedups shown in Fig. 7, when the frequency of scalar unit is the same (0.95 GHz) as the SIMT unit. The next three sets of results are those with the scalar unit running at 0.475 GHz (labeled as '1/2'), 0.237 GHz (labeled as '1/4') and 0.118 GHz (labeled as '1/8'), respectively. From Fig. 9, we can see that the speedups are reduced with reduced frequencies. The reason is the reduced memory throughput of scalar unit and the increased pipeline latency of the scalar unit. Nevertheless, we can see that even when the frequency of scalar unit is reduced to 0.475 GHz, our prefetch mechanism delivers 16.0% speedup on average.

VII. COLLABORATIVE EXECUTION PARADIGM II: CONTROL DIVERGENCE ELIMINATION

Control divergence presents a significant challenge for data parallel machines. Different solutions have been proposed at the software [41] and hardware levels [12][13][35]. Here, we propose to use the scalar unit to eliminate control divergence at run time. In Fig. 10, we show a common code example that leads to control divergence. The number of iterations of the while loop is dependent on thread id and can only be determined at the time when the loop body is executed for each thread. Due to SIMT processing, all the threads in a warp will have to go through

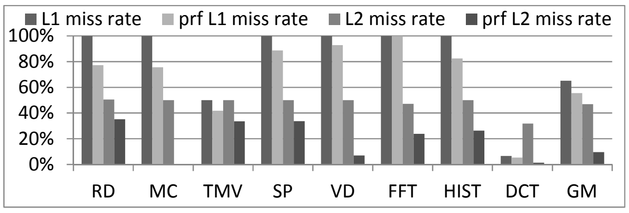

Figure 8. Cache miss rate comparison between our approach and the baseline without prefetching.

the maximum number of loop iterations. The code example in Fig. 10 can be extended to other control divergence cases. For example, we can initialize the variable ‘*status*’ using ‘*workload(tid)*’ and replace ‘*while*’ with ‘*if*’, which means the maximum number of the loop body is only one.

To reduce the adverse performance impact of control divergence, we propose the following compiler transformations. First, the SIMT kernel itself is changed such that it will write the ‘status’ flag of each thread as a status mask into shared memory. Second, the scalar kernel will process the status mask from shared memory and compress the mask by remapping the thread id information. This process is illustrated in Fig. 11, where the warp size is assumed as four for simplicity. As shown in the figure, after the first iteration, SIMT threads 0, 1, 4, and 7 have their status flag set. After reading such status information from shared memory, the scalar unit will remap the workloads of threads 0, 1, 2, and 3 to take over the workload of threads 0, 1, 4, and 7, respectively, and return such mapping information to the SIMT threads. Then, the SIMT threads will continue to execute the second iteration. At this time, the first four threads are active. With a warp size of 4, there is no control divergence. If without such remapping, the second loop iteration needs to be executed by two warps as threads 0 and 1 are in the first warp, and threads 4 and 7 are in the second warp.

The implementation of our proposed compiler transformation is presented using CUDA-like pseudo code as shown in Fig. 12. As shown in Fig. 12a, our proposed implementation involves several code changes in the SIMT kernel. First, we introduce the shared-memory variables ‘*active_count*’ and ‘*actives*’. The variable ‘*active_count*’ records the number of active SIMT threads in the thread block and the ‘*actives*’ array keep the ids of these living threads. Both variables are in shared memory and are to be accessed by both the SIMT threads and the scalar thread. Second, a local variable ‘*next_tid*’ is introduced in the SIMT kernel to keep the remapped thread id, which is then used to identify the workload. When ‘*next_tid*’ is -1, it means that the corresponding thread has finished all its workload. Third, the while loop condition is changed so as to check the variable ‘*active_count*’. If ‘*active_count*’ is not zero, it means there are still active threads and the while loop is entered. Fourth, the code is inserted for computing the variables ‘*active_count*’ and ‘*actives*’ and then the synchronization primitive ‘*__syncthreads(1)*’ is inserted to inform the scalar thread that both ‘*active_count*’ and ‘*actives*’ have been updated and to be processed. With another ‘*__syncthreads(1)*’, the SIMT threads know the scalar thread has finished the processing and resume their execution. If there are live-ins and live-outs of the ‘*workload*’ function using local variables, the thread remapping also need to move these variables using shared memory. Since our threads remapping is implemented through data movement in shared memory, it does not introduce bank conflicts for register accesses.

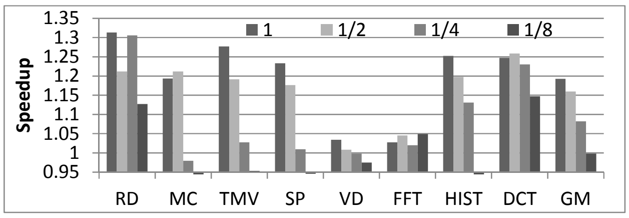

Figure 9. The performance impact of reducing the frequency of the scalar unit.

```
status = true;
while (status) {  status  = workload(tid); }
```

Figure 10. An example of control divergence, where the ‘while’ loop condition is dependent on thread id.

The scalar kernel shown in Fig. 12b compresses the ‘*actives*’ array by moving the active thread ids into the ‘*actives*’ array one by one. Then, it updates ‘*active_count*’. Two synchronizations are also used to guarantee the correct execution order between SIMT threads and the scalar thread.

Some optimizations can be applied to our implementation in Fig. 12 for better performance. First, if ‘*active_count*’ is not larger than the warp size, the remapping can be skipped. Second, we can promote the computation of ‘*status*’ in the SIMT kernel so that the remapping calculation in scalar unit can start as soon as possible. This way, when the scalar unit computes the remapping, the SIMT unit can still do some useful work which is independent on the result of ‘*status*’.

We evaluate the performance benefits of our proposed approach on five benchmarks, LPS, SEG, RT, NQU and MB, and they achieve 15.0%, 10.6%, 4.2%, 1.3%, and 4.0% speedups, respectively. The detailed performance results are shown in Fig. 13. In the figure, we report the execution time of the SIMT unit (labeled ‘_SIMT’) and the scalar unit (labeled ‘_scalar’) normalized to the baseline (labeled ‘_bl’). We classify an execution cycle into the following categories: idle cycles (labeled ‘IDLE’), cycles with 1 to 8 active threads in each warp (labeled ‘W1-W8’), cycles with 9 to 16 active threads in each warp (labeled ‘W9-W16’), cycles with 17 to 24 active threads in each warp (labeled ‘W17-W24’) and cycles with 25 to 32 active threads in each warp (labeled as ‘W25-W32’). Serialized execution between the scalar unit and the SIMT unit is accounted for in the overall execution time as idle cycles in the SIMT unit since it waits for __syncthreads(1) from the scalar unit. Taking the benchmark RT as an example, in the baseline, during 37.5% of the overall cycles, 1 to 8 threads are active in each warp; during 22.9% of the overall cycles, 25 to 32 threads are active in each warp; and 18.0% of the overall cycles are idle cycles. With our collaborative execution paradigm, during 26.2% of the overall cycles, 1 to 8 threads are active in each warp;

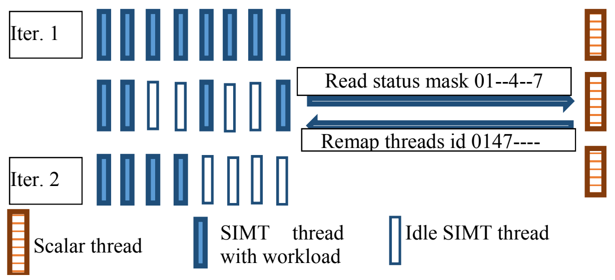
Iter. 1
Read status mask 01--4--7
Remap threads id 0147----
Iter. 2
Scalar thread
SIMT thread with workload
Idle SIMT thread

Figure 11. The process of using a scalar thread to eliminate control divergence in SIMT threads, assuming the warp size of 4.

```
active_count = blockDim.x*blockDim.y;
next_tid = status?tid:-1; actives[tid]=next_tid;
  __syncthreads(1);
while (active_count) {
  if (next_tid!= -1) status = workload(next_tid);
  if (next_tid!= -1 && status) actives[tid] = next_tid;
  else actives[tid] = -1;
  __syncthreads(1); // waiting for the result from scalar unit
  __syncthreads(1);
  next_tid = actives[tid]; /*load next workload*/}
```

(a)The SIMT kernel after divergence elimination

```
__syncthreads(1);
while (active_count) {
  __syncthreads(1);
  int cur = 0;
  for (int i=0; i<BLOCK_SIZE; i++) {
   if (actives[i]!=-1) {
     actives[cur] = actives[i]; // map the living thread id
     if (i!=cur) actives[i] = -1;
     cur++; }}
  active_count  = cur;
  __syncthreads(1); }
```

(b)The scalar kernel

Figure 12. Pseudo-code implementation of control divergence elimination using scalar threads.

during 25.9% of the overall cycles, 25-32 threads are active in each warp, and 26.2% of overall cycles are idle. This means that after we combine partially active warps, it becomes more often to have highly populated warps. The benchmarks, LPS and MB, show the similar behavior to RT. For the benchmark SEG, the cycles in the W17-24 category of the baseline are distributed to the W25-32 and W1-W8 category after we merge the partially active warps. The reason is that as the threads are remapped, it cannot be guaranteed that the number of active threads is a multiple of the warp size (32). The reason why NQU does not get much performance improvement is that NQU has only two active warps. When two warps are merged into one using our approach, the performance is limited by the latency of pipeline and memory accesses, which result in a high amount of idle cycles. For all the benchmarks, the scalar unit is busy only for a negligible amount of time, as shown in Fig. 13.

We also collect the number of instructions being executed and our results show that our proposed solution requires 2.1% instructions on average to be executed to eliminate control divergence.

While our approach reduces the number of active warps, the number of active threads is not changed. We assume that inactive thread will not consume any power, which is the case for both baseline and our proposed architectures. Therefore the energy consumption results show that the only power savings from our approach are from reduced execution time and the scalar unit adds limited dynamic power overhead as it executes a very small number of instructions. Overall, our approach shows up to 7.8% and an average 1.0% energy savings.

VIII. Collaborative Execution Paradigm III: Scalar Workload Extraction

A key reason for the scalar unit in AMD GCN architecture is that many ALU operations in GPU programs are actually scalar operations in nature. Therefore, instead of letting all threads repeat such redundant computations, the compiler of GCN architecture identifies them and generates scalar instructions. At run time, the scalar instructions are only executed once for each wavefront/warp. Although our proposed architecture can be morphed to the AMD GCN architecture, we show that our proposed architecture offers additional solutions with similar or better performance.

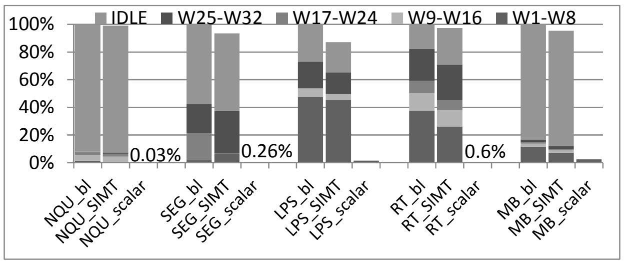

Figure 13. The cycle distribution of the baseline ('_bl'), the SIMT unit ('_SIMT') and the scalar unit ('_scalar'). The scalar unit eliminates control divergence in SIMT units.

In our scalar workload extraction, we target at two scenarios. The first is the instructions that are not dependent on thread id. These operations are the scalar operations in nature and are also referred to as uniform vector operations [6]. The second is a special case of control divergence, where the SIMT kernel has the following '*if*' statement: '*if (threadIdx.x == K) {...}*' where *K* is a constant. In other words, during execution, in each SIMT thread block, there is just one thread performing the workload within the if-statement. In Fig. 14, we present our compiler transformation for SIMT kernels and scalar kernels. As shown in the figure, after detecting the scalar workload in an SIMT kernel, the SIMT kernel is changed so that it copies live-ins to shared memory and invokes the scalar thread through a '*__syncthreads(1)*' statement. With another '*__syncthreads(1)*' statement, the SIMT kernel is informed that the computation has been finished and the live-outs are ready in shared memory. The scalar kernel is generated in a complementary manner: after the first '*__syncthreads(1)*', it reads the live-ins, carries out the computation and copies the live-outs (i.e., computation results) to shared memory. Then, it wakes up the SIMT thread with the second '*__syncthreads(1)*'.

Fig. 15 shows the performance results for five benchmarks using the scalar unit for the scalar workloads in the SIMT kernels. The performance results are reported as normalized execution time and the execution cycles are classified into different categories, the same as in Fig. 13.

Among the benchmarks, NN has many SIMT kernel functions, for which the thread block dimension is actually 1, meaning that there is just one thread in a thread block. Therefore, we can use the scalar units to execute half of the thread blocks to improve the throughput. Therefore, after we distribute the workload to the scalar unit, half of the cycles when the vector instruction has one active lane (W1-8 in the figure) are moved to scalar execution. As there is no synchronization between the scalar and the SIMT unit, scalar execution overlaps with vector execution, rather than being serialized. Due to its significant performance gains, there is a

```
1. Procedure scalar_workload_extraction(SIMT Kernel kernel) {
2.   for each If-Statement, ifs, or a group of uniform vector
     operations, uvo, in kernel {
3.     if (the condition of ifs is scalar check) or there is a uvo {
4.       generate an empty statement {} called nbb
5.        insert statements to declare shared memory variables in nbb
6.        insert statements in nbb to copy live-ins of ifs or uvo to shared
     memory
7.        insert __syncthreads(1) in nbb
8.        insert __syncthreads(1) in nbb
9.        insert statements in nbb to read live-outs of ifs or uvo from
     shared memory
10.       replace ifs or uvo with nbb
11.   } } }
```

(a) Code transformation for SIMT kernels

```
1.  Procedure scalar_thread_gen (SIMT Kernel kernel) {
2.  for each If-Statement, ifs, or a group of uniform vector
    operations, uvo, in kernel {
3.    if (the condition of ifs is scalar check) or there is a uvo {
4.      generate an empty statement {} called nbb
5.      insert statements to declare shared memory variables in nbb
6.      insert __syncthreads(1) in nbb
7.      insert statements in nbb to read live-ins of ifs or uvo from
    shared memory
8.      copy the operations in ifs or uvo in nbb
9.      insert statements in nbb to copy live-outs of ifs or uvo to
    shared memory
10.     insert __syncthreads(1) in nbb } } }
```

(b) Code transformation to generate a scalar kernel

Figure 14. The compiler algorithm for scalar workload extraction.

26.6% overall energy reduction, which is mainly from static energy and the scalar unit incurs 12.1% more dynamic energy consumption.

For the benchmark, FFT, its kernel requires the '*sin*' and '*cos*' calculation at the angles '*(PI/4*(threadIdx.x%4))*'. Because these angles are different for different threads in the same warp/wavefront, the AMD GCN compiler cannot utilize the scalar unit to handle such workload for a whole wavefront. However, in our approach, we can use the scalar unit to compute the sin and cos functions of (PI/4*0), (PI/4*1), (PI/4*2), and (PI/4*3) and let the SIMT threads read the results from shared memory. Although the performance benefits are small, the actual savings are in the fact that we only use one scalar unit to finish the computation for a SIMT thread block. From the figure, we can see that the number of the cycles, when SIMT instructions have 25 to 32 active lanes, is reduced from 38.3% to 35.8% and the number of the cycles, when vector instructions have 17 to 24 active lanes, is reduced from 2.8% to 0.9%. We only introduce a small amount of scalar execution cycles (1.1%) to achieve such savings. The improved performance leads to 2.4% static energy savings.

For the benchmark, SS, the workload under the control flow '*if(threadIdx.x==0)*' is distributed to the scalar unit. The workload contains many off-chip memory accesses, thereby generating many stall cycles. After we move the workload to the scalar threads, we also optimize it by moving '*__syncthreads(1)*' as early as possible so that the scalar execution can overlap with SIMT computations. This optimization significantly reduces the idle cycles in vector execution, resulting in a 26.8% performance improvement and 18.7% static energy savings.

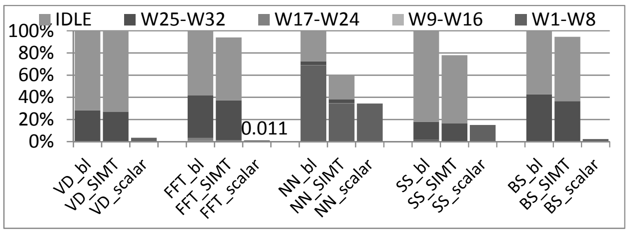

Figure 15. The cycle distribution of the baseline ('_bl'), the SIMT unit ('_SIMT') and the scalar unit ('_scalar'). The scalar unit executes the scalar workload of SIMT units.

For the benchmark VD and BS, the performance bottleneck is the off-chip memory bandwidth. Our approach eliminates some SIMT instructions similar to the product '*blockIdx.x*blockDim.x*' shown in Fig. 1. From the cycle distribution, we can see that with our scalar workload extraction, we reduce 1.5%, 6.0% of the cycles spending on SIMT execution and introduce 3.5%, 2.4% of the cycles of scalar execution for VD, BS, respectively. Due to synchronization, during the cycles spent on scalar execution results, the SIMT units are idle. As the scalar computations replace the vector operations in all the warps in a thread block, it generates a slight performance improvement of 0.4% for VD and 2.7% for BS. The removal of uniform vector operations leads to 3.9% and 14.3% dynamic energy reductions for VD and BS, respectively.

IX. Related Work

GPGPU exploits the SIMT architecture, which is evolved from vector processors [9][10][11][21][22][23][33][37]. Compared with previous studies on vector processors, the scalar unit and SIMT unit in our architecture are decoupled and have their own instruction streams. Such decoupled execution provides flexible ways to improve the performance as shown in our collaborative execution paradigms. The vector-thread architecture [10][26] can also let the scalar unit and the SIMT unit to follow their own instruction streams. It can offer the benefit of decoupled access-computation [10], independent scalar workload computation and other helper thread functions. The fundamental difference is that it is built upon vector processing and programming for vector processors is significantly more complicated than the SIMT programming model [26]. In a sense, our proposed architecture integrates the advantages of VT architecture and the easiness of SIMT programming.

The problems targeted by our collaborative execution paradigms have been well recognized. First, execution driven data prefetching [7][20][29][34] is a classic technique to hide the long latency of off-chip memory. Lee et al. [24] proposed many thread aware prefetching in GPGPUs. Jog et al. [18] improved the GPGPU prefetching by modifying warp scheduling policy. They also proposed memory-side opportunistic prefetching to utilize open DRAM rows [19]. Woo et al. [38] proposed to use the GPU to prefetch data for the CPU. Yang et al. [40] proposed to use the CPU to prefetch the data for GPU applications on fused architecture. Second, both software and hardware solutions have been proposed for the control divergence issue on vector processor and GPUs [4][12][13][17][30][32][39][41]. However, these hardware solutions typically incur significant hardware changes. Compared to [13][32], our threads remapping is implemented through data movement instead of remapping software thread ids and hardware thread ids. Therefore the issue of register bank conflicts does not exist. On the other hand, software solutions [41] have significant performance overhead in using the CPU to eliminate control divergence. In our solution, the scalar unit and SIMT unit are in the same SMX and the communication has very low overhead. Third, we also show that the scalar unit can be used to handle scalar workloads in GPU programs. The most related solution is the AMD GCN architecture, in which the scalar unit and the SIMT unit share the same instruction stream and the compiler is responsible to identify the scalar operations. Although compiler algorithms [27] have been proposed to find more scalar operations, the capability of the scalar unit is still limited because of the shared instruction stream with the SIMT unit. Furthermore, our proposed architecture also has the flexibility to be configured as a GCN processor.

X. Conclusions

In this paper, we propose to extend the scalar unit in the recent GCN architecture such that it can either share the instruction stream with the SIMT unit or have its own instruction stream. The programming model of our proposed architecture is extended from current GPGPU programming model. We present three collaborative execution paradigms to show the benefits of our proposed architecture and programming model. First, we use scalar units to prefetch data for SIMT units. Second, we use scalar units to eliminate control divergence at runtime. Third, the scalar workloads of SIMT/vector programs are exacted and distributed to scalar units. Our experimental results show that our proposed solutions achieve significant performance gains at the cost of low instruction execution overhead from scalar units. We also demonstrate that the scalar kernel can be automatically generated using our proposed compiler algorithms, thereby not increasing the programming complexity.

Acknowledgements

We thank the anonymous reviewers for their insightful comments to improve our paper. This work is supported by an NSF grant CCF-1216569 and a NSF CAREER award CCF-0968667.

References

[1] AMD Accelerated Parallel Processing SDK V2.3, 2011

[2] AMD Accelerated Processing Units, 2011.

[3] A. Bakhoda, et al., Analyzing CUDA workloads using a detailed GPU simulator, ISPASS, 2009.

[4] R. Balasubramonian, et al., Dynamically allocating processor resources between nearby and distant ILP, ISCA, 2001.

[5] S. Che, et al., Rodinia: A Benchmark Suite for Heterogeneous Computing, IISWC, 2009.

[6] S. Collange, et al., Dynamic detection of uniform and affine vectors in GPGPU computations, Euro-Par, 2009

[7] J. D. Collins, et al., Speculative precomputation: long range prefetching of delinquent loads, ISCA, 2001

[8] CUDSEG, http://code.google.com/p/cudaseg/

[9] R. Espasa, et al., Tarantula: a vector extension to the alpha architecture, ISCA, 2002.

[10] R. Espasa , et al., Decoupled vector architectures, HPCA, 1996.

[11] R. Espasa, et al., Out-of-order vector architectures, MICRO, 1997.

[12] W. Fung, et al., Thread block compaction for efficient SIMT control flow, HPCA 2011.

[13] W. Fung, et al., Dynamic warp formation and scheduling for efficient GPU control flow, MICRO, 2007.

[14] N. Goswami, A. Verma, and T. Li, GPU-PowerSim, 2012.

[15] N. Govindaraju, et al., High performance discrete Fourier transforms on graphics processors, SC, 2008.

[16] GT 640, http://www.geforce.com/hardware/desktop-gpus/geforce-gt-640-oem/specifications

[17] S. Hong, et al., Accelerating CUDA graph algorithms at maximum warp, PPoPP, 2011.

[18] A. Jog, et al., Orchestrated Scheduling and Prefetching for GPGPUs, ISCA, 2013.

[19] A. Jog, et al., OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance, ASPLOS, 2013.

[20] D. Kim and D. Yeung, Design and evaluation of compiler algorithms for pre-execution, ASPLOS, 2002.

[21] C. Kozyrakis and D. Patterson, Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks, ISCA, 2002.

[22] C. Kozyrakis and D. Patterson, Overcoming the limitations of conventional vector processors, ISCA, 2003.

[23] R. Krashinsky, et al., The Vector-Thread Architecture, ISCA, 2004.

[24] J. Lee, et al., Many-thread aware prefetching mechanisms for gpgpu applications, MICRO, 2010.

[25] S. I. Lee, et al., Cetus – an extensible compiler infrastructure for source-to-source transformation, LCPC, 2003

[26] Y. Lee, et al., Exploring the tradeoffs between programmability and efficiency in data parallel accelerators, ISCA 2011

[27] Y. Lee, et al. Convergence and Scalarization for Data-Parallel Architectures. CGO 2013.

[28] S. Li, et al., McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures, MICRO, 2009.

[29] C. K. Luk, Tolerating memory latency through soft-ware-controlled pre-execution in simultaneous multithreading processors, ISCA,2001.

[30] J. Meng, et al., Dynamic warp subdivision for integrated branch and memory divergence tolerance, ISCA, 2010.

[31] NVIDIA GPU Computing SDK 3.1 2011.

[32] M. Rhu, et al., Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation, ISCA, 2013.

[33] J. E. Smith, et al., Vector Instruction Set Support for Conditional Operations, ISCA, 2000

[34] Y. Solihin, et al., Using a user-level memory thread for correlation prefetching, ISCA, 2002

[35] M. Steffen, et al., Dynamic Thread Creation for Improving Processor Utilization on SIMT Streaming Processor Architectures, MICRO, 2010.

[36] Visual Molecular Dynamics, http://www.ks.uiuc.edu/Research/vmd/

[37] J. Wawrzynek, et al., Spert-II: A vector Microprocessor System, IEEE Computer, 1996.

[38] D. H. Woo, et al., COMPASS: a programmable data prefetcher using idle GPU shaders, ASPLOS, 2010.

[39] P. Xiang, et al., Warp-Level Divergence in GPUs: Characterization, Impact and Mitigation, HPCA, 2014.

[40] Y. Yang, et al., CPU-Assisted GPGPU on Fused CPU-GPU Architectures, HPCA, 2012.

[41] E. Z. Zhang, et al., Streamlining GPU Applications On the Fly, ICS, 2010.