Approximating Warps with Intra-warp Operand Value Similarity

Daniel Wong
University of California, Riverside
dwong@ece.ucr.edu

Nam Sung Kim
University of Illinois, Urbana-Champaign
nskim@illinois.edu

Murali Annavaram
University of Southern California
annavara@usc.edu

ABSTRACT

Value locality, the recurrence of a previously-seen value, has been the enabler of myriad optimization techniques in traditional processors. Value similarity relaxes the constraint of value locality by allowing values to differ in the lowest significant bits where values are micro-architecturally near. With the end of Dennard Scaling and the turn towards massively parallel accelerators, we revisit value similarity in the context of GPUs. We identify a form of value similarity called *intra-warp operand value similarity*, which is abundant in GPUs. We present *Warp Approximation*, which leverages intra-warp operand value similarity to trade off accuracy for energy. Warp Approximation dynamically identifies intra-warp operand value similarity in hardware, and executes a single representative thread on behalf of all the active threads in a warp, thereby producing a representative value with approximate value locality. This representative value can then be stored compactly in the register file as a value similar scalar, reducing the read and write energy when dealing with approximate data. With Warp Approximation, we can reduce execution unit energy by 37%, register file energy by 28%, and improve overall GPGPU energy efficiency by 26% with minimal quality degradation.

1. INTRODUCTION

The similarity of values produced and consumed by applications in computer systems are well-known concepts that has been leveraged to improve all aspects of computers. It has been recognized that some values occur frequently during an application execution. This phenomenon of *value locality*, the likelihood of the recurrence of a previously-seen value within a storage location, has been used as the basis for many prior works [1–8].

The authors in [9] introduced the broader concept of *partial value locality*, where multiple values are identical in a subset of consecutive bits. A special case of partial value locality is called *value similarity*, where the values differ only in the least significant bits in their binary representation. For instance, the values 124 (01111100_2) and 127 (01111111_2) are microarchitecturally close, because they differ only in a subset of the least significant bits, and hence are categorized are having value similarity. Conversely, the values 127 (01111111_2) and 128 (10000000_2) are arithmetically close, but are not microarchitecturally close since the LSBs are different, and thus, is not categorized as having value similarity.

More specifically, two 32-bit values are called (32-d)-similar if they differ in the d least significant bits and are equal in the remaining (32-d) high-order bits. To simplify the notation, (32-d)-similar will be referred to as d-similar in the remainder of the paper. For example, 4-similar will indicate values which differs in the 4 least significant bits and are equal in the remaining 28 high-order bits. For example, the values 113 (1110001_2) and 127 (1111111_2) are 4-similar since they only differ in the 4 least significant bits.

Partial value locality has been used for energy efficient and compact register organization [9] and for fuzzy memoization [10]. More recently, Load Value Approximation [11] leverages *approximate value locality*, where values are arithmetically close, to generate approximation of load values. Value similarity is a subset of approximate value locality where the values are both arithmetically and microarchitecturally close (only differing in LSBs).

Until now, value similarity has remained largely unexplored in GPUs. GPUs are massively parallel accelerators following the SIMT (Single Instruction Multiple Thread) execution model, where the same instruction can execute on different data operands concurrently. In GPUs, a group of 32 threads, called a *warp*, is executed in a lockstep manner. Due to threads running in lockstep, the opportunity for identifying and exploiting value similarity across different SIMT lanes naturally arises. Prior work [12] has explored value locality in GPUs by identifying opportunities to scalarize warps where all threads within a warp operate on the *same* data. We identified that significant energy saving opportunity remains with value similarity where all threads within a warp operates on *similar* data. Our work is the first to identify and exploit the broader notion of value similarity to enable GPU energy efficiency by leveraging the level of d-similarity as a knob to trade off application quality for energy savings.

In this paper, we make the following contributions:

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

Value Similarity in GPGPUs (Section 2): We first explore the prevalence of value locality in GPUs. We identify a type of value similarity specific to GPUs, which we refer to as *intra-warp operand value similarity*, where all threads within a warp operate on micro-architecturally similar operand values.

Warp Approximation (Section 3): To leverage intra-warp operand value similarity for energy savings, we propose Warp Approximation, an architecture-level approximate computation technique. A key insight of our approximation technique is that it enable programmers or compilers to identify a coarse-grain region of code where approximation can be tolerated, and then opportunistically approximate when the hardware dynamically detects value similarity. Warp Approximation consists of the following two components.

Representative Value Storage (Section 3.1): When intra-warp operand value similarity is detected dynamically in hardware, rather than storing the full vector entry into register file, we can store the value similar data as a *value similar scalar* in the register file. With Representative Value Storage, we can reduce the access energy for reading and writing representative values to the register file.

Representative Thread Execution (Section 3.2): When intra-warp operand value similarity is detected dynamically in hardware, rather than executing all lanes in a warp precisely, we can execute a single *representative thread*. This representative thread will then precisely generate a *representative value*. For the representative thread, and other threads with the same input operand, the representative value will be correct. On the other hand, for other threads with value similar input operands, the representative value would exhibit approximate value locality. By executing a single representative thread we can reduce dynamic energy. By introducing idleness to the inactive lanes, we introduce opportunity for power gating to save static energy.

CUDA API/ISA extensions (Section 4): We introduced mechanisms for programmers/compilers to inform the hardware of safe-to-approximate regions of code, as well as mechanisms to control the degree of approximation by changing the level of *d*-similarity. We present a case study on how we annotate safe approximate regions in CUDA.

We present out evaluation in section 5. In section 6 we discuss matters related to error bounding in Warp Approximation. Section 7 discusses related work and we conclude in section 8.

2. VALUE SIMILARITY IN GPUS

In this paper, we use an Nvidia Fermi GTX480-like GPU architecture. All 32 threads in a warp follow the single instruction multiple thread (SIMT) execution model, where they share a single program counter (PC) but accesses different data operands.

We identified a form of GPU-specific data value similarity called *intra-warp operand value similarity*, where the input operands of all threads within a warp exhibit value similarity. For instructions with more than one operand, intra-warp operand value similarity requires that *each* operand across all threads within a warp must exhibit value similarity. For instance, if an ADD instruction has two input operands, R1 and R2, and only R1 values are similar but R2 values differ, then we do not consider that warp instruction to exhibit value similarity.

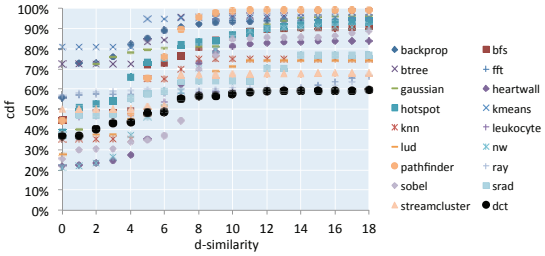


Figure 1: Intra-warp operand value similarity is prevalent in GPU applications

To explore the prevalence of intra-warp value similarity in GPUs, we ran GPGPU-Sim v3.2.1 [13] with a large variety of CUDA applications from the Rodinia benchmark suite [14] (detailed experimental setup is discussed in Section 5). We sweep a range of *d*-similarity values to characterize the amount of value similarity that exists.

Figure 1 shows the intra-warp value similarity that exists. The x-axis represents the level of *d*-similarity, the y-axis represents the percentage of all issued warp instructions that has operands with that *d*-similarity or lower. A *d*-similarity of 0 indicates warps that have intra-warp operand value locality, that is, all operand values are identical across all threads in a warp. The percentage of warps with intra-warp value locality is varied but they are extremely prevalent in most workloads. We found that workloads experience anywhere between 20% to 80% of warp instructions with intra-warp operand value locality. On average, we observe 49.7% of warp instructions exhibit intra-warp operand value locality.

We observe that as the level of *d*-similarity increases beyond zero, the percentage of instructions exhibiting intra-warp operand value similarity increases, but plateaus quickly. In general, most workloads plateau at 7-similarity. Even with 4-similarity, we observe that on average 56.5% of warp instructions exhibit intra-warp operand value similarity. Since we only require a relatively small *d*-similarity to observe a high percentage of value similarity, it bodes well for a variety of usage scenarios that can exploit value similarity. If value similarity can be observed only at a high value of *d*-similarity, then such similarities would be difficult to exploit.

3. WARP APPROXIMATION

In this section, we present an application of intra-warp operand value similarity to enable approximate computation in GPUs. We propose *Warp Approximation*, an architecture-level approximate computation technique which leverages intra-warp operand value similarity and the existing precise hardware in GPUs. A key insight of our approximation technique is that it enable programmers or compilers to identify a coarse-grain region of code where approximation can be tolerated, and then opportunistically approximate when the hardware dynamically detects value similarity. The degree of approximation can be controlled by the level of *d*-similarity. A higher level of *d*-similarity would result in more instances of value similarity, thus, enabling more opportunity for Warp Approximation and more errors. Con-

versely, a lower level of *d*-similarity would result in less opportunity for Warp Approximation and less errors. Warp Approximation also supports approximation of divergent warps with 1 level of divergence.

Warp Approximation consists of Representative Value Storage and Representative Thread Execution. Intra-warp operand value similarity is detected dynamically in hardware. Rather than storing the value for all 32 threads of the warp in the register file, we can store the value similar data as a *value similar scalar* in the register file. With Representative Value Storage, we can reduce the access energy for reading and writing representative values to the register file.

In Representative Thread Execution when value similarity is detected, rather than executing all lanes in a warp precisely, we pick a single *representative thread* to execute. This representative thread will then precisely generate a *representative value*. For the representative thread, and other threads with the same input operand, the representative value will be correct. On the other hand, for other threads with value similar input operands, the representative value would exhibit approximate value locality. By executing a single representative thread we can reduce dynamic energy. By introducing idleness to the inactive lanes, we introduce opportunity for power gating to save static energy.

3.1 Representative Value Storage

We will now describe how Representative Value Storage can be used to detect and store a representative value as a value similar scalar in the register file in order to save read and write access energy.

3.1.1 Detecting intra-warp operand value similarity

We detect value similarity by placing a comparison stage before the writeback stage to the register file. Whenever values are written into the register file, the values are compared for value similarity, thus enabling detection of all value similar data. The value similarity information is then stored as a metadata bit, the `SimilarBit`, within each register entry. GPU register files are logically organized as very wide registers (typically 1024bit-wide entries). Thus, one `SimilarBit` is added to each 1024bit-wide register entry.

The value similarity check is carried out even if the warp is not in an approximate region to ensure that the `SimilarBit` in the registers are not stale. If we stop checking for value similarity outside of approximate regions, we can miss opportunity to detect value similarity. Then when the warp enters an approximate region, we would have value similarity register entries that are not marked, leading to missed opportunity for Representative Thread Execution. Therefore, we choose to enable the comparison stage at all times to maximize opportunity for Warp Approximation.

Comparison Stage: To detect value similarity we add a comparison stage before the write-back stage. In figure 2a we show the hardware implementation of the comparison stages. In order to detect similarity, even when there is a divergent warp, we first need a comparison selection stage to identify an active thread lane whose output operand value will be used to compare against all the other active threads. For this, we use a priority encoder to encode the active mask of the warp to select the output operand value from an active lane for comparison and place it into a *compare* register.

In the comparison stage, every lane will xor the values of their output operand with the value of the compare register, ignoring the last *d* LSBs as specified by the level of *d*-similarity desired. The result of the xor is then filtered through a mux to only select the active lanes and compared using an all zero detector (essentially a simple nor operation) to set the `SimilarBit`. In our current hardware description we very conservatively assume that comparison of operand outputs add one stage delay, and the thread selection logic also requires a separate pipeline stage, requiring a worst-case 2-stage delay for Warp Approximation.

The comparison logic can be bypassed when an instruction operates on input operands whose `SimilarBit`s are all already set. In this case the output operand's `SimilarBit` is set to 1 without using the comparison logic, so long as the output value has the same d-similarity as the input values. For instance, we may have two operands that are 3-similar (10111000 and 10111111) and if we add 1 to both, the results are no longer 3-similar (1011001 and 11111110). To check for this case, we can simply xor the 32-d higher order bits of one input operand and the output value to ensure that the output has the same level of d-similarity, before directly setting the output's `SimilarBit`.

3.1.2 Writing Representative Value to Register File

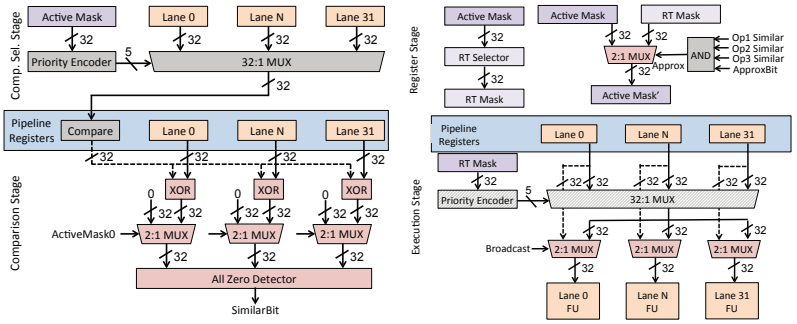
We will use figure 3 as an illustrative example to facilitate our discussion. This figure assumes a warp with 4 threads and a vector register file entry that can store 4 32-bit values.

We will first illustrate the base case for Representative Value Storage with no divergence. If a value is detected as similar, then instead of storing all 32 values into the vector register entry, we can simply store only the representative value as a *value similar scalar* in the register file. For example, in Ⓐ, if the original vector register file entry contains <7,6,4,7> and we set the level of *d*-similarity to *2*-similar, then this vector is value similar. To store this vector as a value similar scalar, we set the `SimilarBit`, and only store the value in the Representative Thread's lane using a *RT Mask*.

Selecting Representative Thread: To select a representative thread, we add a representative thread selector (RT Selector) in the register stage as shown in figure 2b. Given an active mask and a representative thread selection priority, it will generate a representative thread mask (RT Mask). The RT mask has a single 1 in the representative thread lane and 0 in all other lanes. We explored several representative thread selection priorities and found it has negligible impact on our results. Hence, in the rest of this paper we use the lowest order active lane as the representative thread lane for comparison. By simply using the lowest active lane policy for representative thread selection, the RT Selector circuit block is simplified into only a priority encoder and decoder.

3.1.3 Handling Divergence

To keep hardware complexity and overhead low, Warp Approximation only supports a single level of divergence. In the example program in figure 3, the program initially branches into basic block B and basic block F. In the SIMT model, execution of divergent warps are serialized. That is, basic block B will not be executed until basic block F has completed. This ensures that the active threads of block B



(a) Operand Similarity Detector (b) Representative Thread Exec.

Figure 2: Warp Approximation Architectural support

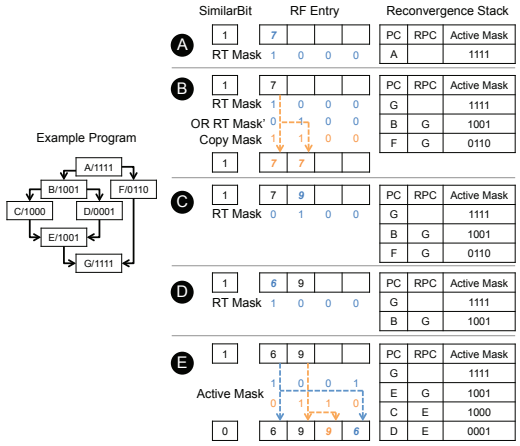


Figure 3: Representative Value Storage

and block F are mutually exclusive. We can leverage this property to minimize the amount of book keeping required to handle divergence.

Ⓑ shows how Representative Value Storage handles branch divergence. On a divergence, both basic block B and F's PC are pushed onto the reconvergence stack, along with the active mask associated with each basic block. Here we see that block B has an active mask of 1001 and block F has an active mask of 0110. Let's assume that block B executes first and its result is also value similar. If block B writes the result back, it will overwrite 7, which block F still requires, therefore the `SimilarBit` will no longer be valid. Conversely, if we execute block F first, block F can only access data values within it's active mask. In this case, block F cannot access the value similar scalar it requires.

In order to solve this dilemma, on a branch divergence, we need to make a copy of the value similar scalar so that it is accessible to an active thread in block F. In order to do so, we simply OR the RT mask of the two divergent paths to create a *copy mask*, and copy the value similar scalar to the active threads in the copy mask. To support this operation, the arbiter can introduce a dummy MOV instruction [15] where we read from the register file using the RT mask, and write to the register file using the copy mask. The hardware modification to support broadcasting of values in the execution unit pipeline is shown in figure 2b. At the beginning of the execution stage we use a circuit similar to the comparison selection logic to select the value similar scalar read from the register file using the RT mask. The dummy MOV instruction will then write-back to the register file, where the copy mask will be used to write to the desired thread location. In [15], it was observed that the introduction of dummy MOV instructions presented negligible performance overheads, which we confirmed in our experiments. We modeled the dummy MOV instructions and included the performance and energy overhead of the MOV instructions in our evaluation.

Now let's say basic block F will execute first (Ⓒ). Let's assume that the instruction will add 2 to the current register's value. Block F has an active mask of 0110, therefore, the RT mask will be 0100. We will first read the value similar scalar of 7 from the register, which was made accessible by the previous step. Representative Thread Execution will then execute 7+2 only in the representative thread's lane. The resultant representative value of 9 is the value similar scalar for the active threads in the active mask 0110. On write-back, we write the value similar scalar of 9 to the register file entry of the active thread in the RT mask.

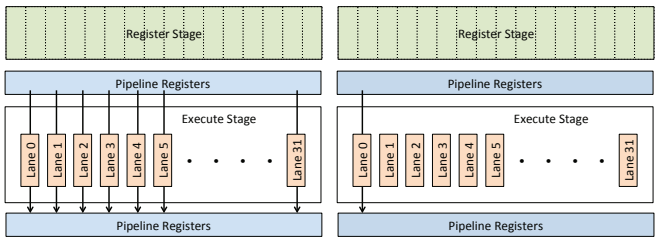
In block B, let's now subtract 1 from the register value (Ⓓ). At this point the register values for block B remains untouched, therefore the value of 7 is still a valid value similar scalar for the active threads in block B. Note that 7 and 9 are not *2*-similar. The `SimilarBit` is still valid because *2*-similarity is still preserved independently within the respective active threads of block F and block B. Block B will follow the same steps as in Ⓒ.

At any point during execution during divergence, if another divergence occurs, a reconvergence occur, or the value written to the register file is no longer value similar, we need to invalidate the `SimilarBit` and expand the value similar scalar in the vector register entry. This scenario is shown in Ⓔ where another divergence occurred. Here, the `SimilarBit` is cleared, and to ensure that the active threads of all future instruction will access the correct values, we need to copy the value similar scalar to all active threads in their respective blocks. The operation here is the same as in Ⓑ, but instead of copying to a copy mask, we copy to the active masks using one dummy MOV instruction per active mask.

To support nested branches, we would need to augment each register file entry to keep track of active mask, and to keep track of which branch each entry belongs to. This would require additional hardware support, as well as less opportunity to save access energy since the number of lanes utilized in the vector register file entry grows with the branch depth. Therefore, we believe that supporting a single level of divergence provides a good tradeoff between energy savings and hardware overhead.

3.2 Representative Thread Execution

Figure 4 shows the general overview of SIMT execution with and without Warp Approximation. In the baseline precise SIMT model (figure 4a), each thread would operate on its own operands in its own lane. Figure 4b shows the op-



(a) Exact SIMT Execution (b) Approximate Execution

Figure 4: When *similar* operands are detected, Warp Approximation executes only a single *representative thread* out of the warp to approximate the execution of the other threads.

eration under Warp Approximation. When the hardware detects that the operands within the warp are all similar, then it can approximate the current warp instruction as shown in the figure. When an instruction reads multiple input operand registers, and if each input operand register's `SimilarBit` is already set, then the entire instruction is guaranteed to operate on operands that satisfy value similarity.

Instead of computing on each operand precisely in its respective lane, the SIMT hardware instead only takes one thread from the warp, called the representative thread, and computes precisely using the representative thread's operands and the existing precise hardware. In the figure, thread 0 is assumed to be the representative thread. This thread's result, the representative value, will then be representative of all the other thread's computation within that warp. The representative value can then be stored as a value similar scalar in the register file.

Warp Approximation can improve energy-efficiency of GPUs by potentially saving both dynamic and leakage energy. Let's assume that a warp with all 32 threads active is being approximated. By executing only a single representative thread to approximate the computation of all the other threads (as shown in figure 4b), we can potentially cut the dynamic execution unit power to 1/32. Furthermore, this approximation introduces idleness to the higher-order lanes, increasing opportunity for static energy savings through lane-level power gating [16].

Supporting per Warp Approximate Region Annotation: First, the hardware should recognize when a warp is in a region that can be approximated. We will discuss how sections of code can be annotated in section 4. Due to different warp scheduling policies, warps within an SM may be running different sections of code; each warp may independently enter and exit approximate code regions. Therefore, it is imperative that each individual warp keeps track of its own approximate region status. To track whether a warp is within an approximate region, a single bit `ApproxBit` is maintained for each warp.

Executing Representative Threads Figure 2b shows the required hardware support for selecting and routing data from the representative thread. If all operands of the instruction are value similar and the warp is in an approximate region, then we will use the RT mask as the active mask for the next stage. If all operands are not similar, then we simply use the existing active mask. If some operands are similar, and others are not, then we need to broadcast the similar operand re-using the broadcast logic that's already in place for the dummy MOV instruction. Then we can execute using the existing active mask.

The RT mask acts as the effective active mask and activates Warp Approximation by forcing execution in only a single lane. Furthermore, this also enables both approximate and precise warps to operate in the same pipeline at the same time as there are no changes required to the functional unit pipelines. At the end of the execute stage, if the warp instruction can bypass the comparison stage, then the RT mask will be used to write-back into the register file. Otherwise, the original active mask of the warp is used.

4. IDENTIFYING APPROXIMATE REGIONS

Warp Approximation can dynamically identify intra-warp operand value similarity at runtime, and then opportunistically approximate. Given this property, it allows Warp Approximation to enable *value-guided approximation*. Since data is identified dynamically and approximated opportunistically, Warp Approximation does not require programmer to specifically identify variables/instructions/loops/functions that *will* be approximated. Instead, Warp Approximation requires at minimum, course-grain annotation of an approximate region where code *can* be approximated, and then the runtime data values will guide when to approximate.

The concept of approximate regions is similar to relax blocks [17], where regions of code that may experience hardware faults are marked. The main difference between relax blocks and our approximate regions is that in relax blocks errors manifest non-deterministically from hardware faults, while in our approximate regions, errors manifest due to deterministic Warp Approximation. In Warp Approximation, results are reproducible given the same application and inputs. Therefore, Warp Approximation can be used as the hardware substrate for profiling-based approximation frameworks, such as Rumba [18], Sage [19], and QPP [20], to guarantee quality levels. Furthermore, with techniques such as Sage, we can eliminate the requirement for programmer annotation as Sage can automatically prune out *d*-similarity values and approximate regions which can lead to excess errors, leaving only *d*-similarity values and approximate regions which can meet a quality of result level.

Warp Approximation can also support a more fine-grain annotation approach, such as EnerJ [21] which declares individual data as approximate. Approximate data annotation can be passed to the hardware by having the compiler mark each instruction with the approximate variable to set the `ApproxBit` at an instruction granularity.

The goal of this work is to identify the opportunity and challenges of leveraging intra-warp operand value similarity for approximate computation in GPGPU hardware. Integration of higher-level approximation frameworks to guarantee quality of results with Warp Approximation are outside the scope of this work. To achieve our goal of exploring the implications of Warp Approximation, we adopt course-grain annotation of approximate region. For completeness, in section 6, we also describe a hardware-based approach to provide statistical guarantees on meeting target quality levels.

4.1 Annotating Approximate Regions

In this work, we will provide CUDA API and ISA access to hardware knobs that control the degree of approximation, and to annotate safe regions of code that are amenable to approximation. While we manually annotated the code and selected the level of *d*-similarity, we envision that this can be offloaded to compilers or profiling-based frameworks. We present an example case study by annotating a CUDA benchmark, `hostpot`, as shown below.

```
1  // Hotspot benchmark main kernel
2  __global__ void calculate_temp(int iteration , ..., float time_elapsed){
3      __shared__ float temp_on_cuda[BLOCK_SIZE][BLOCK_SIZE];
4      __shared__ float power_on_cuda[BLOCK_SIZE][BLOCK_SIZE];
5      __shared__ float temp_t[BLOCK_SIZE][BLOCK_SIZE];
6      ... // more variables
7      int bx = blockIdx.x;
8      int by = blockIdx.y;
9      int tx=threadIdx.x;
10     int ty=threadIdx.y;
11     ... // calculating block size, boundaries, and indexes
12     int index = grid_cols*loadYidx+loadXidx;
13     ...
14     cudaApproxLevel(6);
15     for (int i=0; i< iteration ; i++){
16     computed = false;
17         if( IN_RANGE(tx, i+1, BLOCK_SIZE-i-2) && ... ) {
18             cudaStartApproximate();
19             computed = true;
20             temp_t[ty][tx] = temp_on_cuda[ty][tx] +
21                     step_div_Cap * (power_on_cuda[ty][tx] +
22                     (temp_on_cuda[S][tx] + temp_on_cuda[N][tx] -
23                     2.0*temp_on_cuda[ty][tx]) * Ry_1 +
24                     (temp_on_cuda[ty][E] + temp_on_cuda[ty][W] -
25                     2.0*temp_on_cuda[ty][tx]) * Rx_1 +
26                     (amb_temp - temp_on_cuda[ty][tx]) * Rz_1);
27             cudaEndApproximate();
28         }
29         __syncthreads();
30 ...
31     }
32
33     if (computed){
34         temp_dst[index]= temp_t[ty][tx];
35     }
36 }
```

Listing 1: Hotspot annotation example

4.1.1 ISA/API Support for Warp Approximation

We propose CUDA API and ISA extensions to enable annotation of coarse-grain regions of code that can be safely approximated. We introduce two function calls: cudaStartApproximate() and cudaEndApproximate(), which denotes the beginning and end of a region of code that can be approximated. In order to notify the hardware of the beginning and end of an approximate region, we require the addition of two instructions extending the ISA. In a production environment, we propose the addition of two instructions: approxBegin and approxEnd and envision that the compiler will translate our proposed API calls into the respective ISA instruction. In addition, we also require an extension to specify the level of approximation that can be tolerated, acting as a knob for quality control. To specify the level of approximation tolerated, we add a cudaApproxLevel() function call to CUDA and an approxLevel instruction to the ISA. As shown in listing 1, we specify our level of approximation as 6 (referring to *6*-similar) on line 14, and we enclose our annotated regions using our introduced APIs (underlined) on line 18 and 27.

4.1.2 Identifying approximate regions

In order to annotate regions of code amenable to approximation, we follow the same established guidelines and principles of many other approximate computing work [11, 17, 22–24].

First, the programmer must ensure that annotated code region does not include calculations for pointers or indexes into arrays. Specific to CUDA, this also includes initializing variables based on thread and block ids, which are typically used to index into the data the kernel will process. For example, `hotspot` is a 2D transient thermal simulation kernel which iteratively solves a series of differential equation for block temperatures. The CUDA code first begins by initializing and calculating indexes and block boundaries. As shown in listing 1, we avoid any code that deals with calculation and initialization of thread id, block id, etc. (variables `bx`, `by`, `tx`, `ty`), as well as index calculations.

Second, we conservatively avoid approximation of control flow. Since Warp Approximation uses precise execution units, it is safe for certain control statements to be approximated if intra-warp operand value locality exist (that is, all control statements across all threads in the warp have the same value). For simplicity, we avoid all approximation of control flow.

Third, the approximate region should cover hot code. Essentially, the approximate regions should have high code coverage to maximize Warp Approximation opportunity. Many prior works has shown that it is not difficult to identify approximate code [11, 17, 21, 23, 25, 26].

5. EVALUATION

We evaluate Warp Approximation by annotating several benchmarks representing diverse application domains. We identified and annotated regions that can be safely approximated to evaluate application output quality, performance, and energy efficiency.

5.1 Evaluation Methodology

We evaluated our proposed techniques using GPGPU-Sim v3.2.1 [13]. We used a Nvidia Fermi GTX480-like configuration. The baseline machine contains 15 SMs running at a core clock of 700MHz. Each SM consists of 2 shader processors (SP), each containing 32 CUDA cores to run integer and floating point operations, 16 LDST units for memory operations and 4 SFUs. In our work we assume each SP unit has 32 CUDA cores running at the core clock frequency. GPU register files are logically organized as 1024 bit wide entries where each entry is capable of feeding 32 bits to each of the 32 threads within a warp [30].

We use GPUWattch [31] and the integrated McPAT [32] for energy and area estimations of the GPU. To estimate the energy overhead of our proposed hardware additions, we synthesized the needed logic blocks to obtain the power consumption, and collected the usage statistics of these additions during runtime.

Warp Schedulers: In our evaluation we used three warp schedulers; Loose Round-Robin (LRR), Two-Level (TL), and Greedy-then-Oldest (GTO), in order to demonstrate the potential of Warp Approximation independent of scheduling policy. In LRR, warps are scheduled in a round robin fash-

Benchmarks	Description	Domain	Input Parameters	d-similarity	Error Metric	Error
blackscholes [27]	Black-Scholes Option Pricing	Financial Analysis	40000 options	23	Average Relative Error	0.09%
dct [27]	Discrete Cosine Transform	Image Compression	512x512 image	5	Image Diff	1.6%
fft [28]	Cooley-Tukey Fast Fourier Transform	Signal Processing	5MB random input	9	% Incorrect Elements	1.2%
hotspot [14]	Transient thermal differential equation solver	Physics Simulation	512x512 grid	6	Average Relative Error	0.006%
knn [14]	k-Nearest Neighbors	Data mining	42764 neighbors	4	% Incorrect Elements	5.5%
ray [13]	Raytrace	Graphics/Gaming	512x512 image	1	Image Diff	3.0%
sobel [29]	Sobel Edge Detector	Image Processing	512x512 image	4	Image Diff	0.9%

Table 1: Evaluated benchmarks including description, input parameters, and application error details

ion. If a warp is not ready to issue during its turn, then the next warp in round robin order will be issued. In TL, warps are separated into an active warps set and an inactive warps set. Warps in the active warps set are ready (or close to ready) to issue. Warps in the inactive warps set are not ready to issue (waiting on long latency instructions, such as memory accesses), or they can be ready but there is no space in the active warps set. Warps within the active warps set are then issued in a LRR manner. GTO issues from a single warp until it has stalled, then picks the oldest ready warp. GTO is able to efficiently capture intra-warp locality.

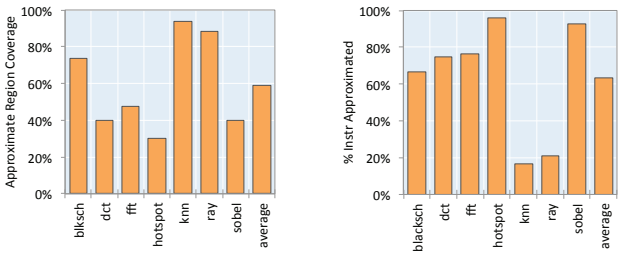
5.1.1 Benchmarks

Table 1 details the benchmark that we evaluated. We selected several benchmarks to represent a diverse set of application. `blackscholes` implements Black-Scholes options pricing for financial analysis. `dct` (discrete cosine transformation) is a widely used compression algorithm with application in audio and images. `fft` is a fast fourier transform with application to signal processing. `hotspot` is a widely used processor temperature estimator, which implements a transient thermal differential equation solver for a 512 by 512 grid. `knn` finds the k-nearest neighbors by calculating the euclidean distance between each neighbor and a specified location, and returns a list of closest neighbors. `ray` is a raytracing kernel with numerous application to graphics and gaming. `sobel` implements the sobel edge detection.

5.1.2 Code Annotation Regions

We annotated each benchmark kernel by identifying regions of code that can be approximated by using the guidelines detailed in section 4. We simply enclose identified regions using cudaStartApproximate() and cudaEndApproximate() without any algorithmic changes. We found it straightforward in identifying regions of code which exhibits patterns which can be safely approximated and are also used frequently.

In `blackscholes` we annotate the entire PDE solver. In `dct` we annotated the discrete cosine transform and the inverse discrete cosine transform kernel, both consisting only of arithmetic operations. In `fft` the annotated region encompasses the twiddle operation and the discrete Fourier transform function, which contained no loops or branches. A large portion of the fft kernel (such as transpose) includes a large number of loads and stores, which we do not approximate, resulting in lower approximate region code coverage. In `hotspot` the annotated region encompasses the differential equations solver as detailed in list 1. In `knn` the annotated region covers a simple kernel computing the euclidean distance.`ray` is one of the more complex CUDA kernel. In



(a) Approximate Region Coverage (b) Approximated Instructions

Figure 5: Approximate Region Coverage

`ray` the annotated region covers a large portion of the render kernel and the notShadowRay function. In `sobel` the annotated region covers all computation without any control flow present. A large portion of the kernel deals with accesses to get adjacent pixel data, which are not approximated, and results in a relatively low approximate code coverage.

5.2 Code Coverage

Figure 5a shows the approximate region code coverage. The annotated code regions covers anywhere from 40% to 90% of all dynamic instructions. On average, our annotated regions account for 59% of the dynamic instruction count. Figure 5b shows the percentage of instructions within the annotated region that is approximated. Not all code within the annotated region will be approximated, rather, only instructions that exhibit value similarity will be opportunistically approximated.

Note that in `fft`, even if we have a random input set, there still exist approximation opportunities. In `fft`, we did not approximate transpose stage (mostly ld/st). Thus, `fft` was relatively fine-grain annotated. The approximated stages have large number of calculation on constants (twiddle stage), and type conversions, where value locality exists independent of input.

5.3 d-similarity vs Application Error

We specify a static *d*-similarity across the entire benchmark run as a knob to vary application error and sweep the values of *d*-similarity for each benchmark. To evaluate the output quality of each application, we used the metric specified in table 1. For benchmarks that output images (`dct`,`ray`, and `sobel`) we use the average root-mean-square image difference. For `fft` and `knn` we calculate the percentage of output elements that differs. For `hotspot` and `blackscholes` we measure the average relative error of each data point.

Figure 6 shows the application error as we vary the level of

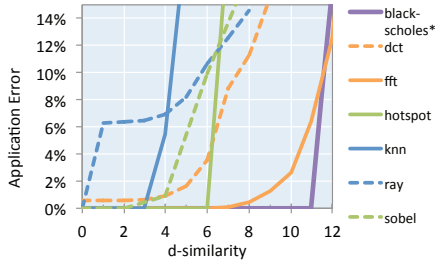
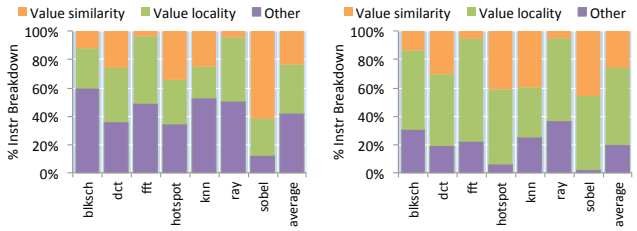


Figure 6: d-similarity vs application error. *black-scholes x-axis should be scaled by 2x (0-24). Shown on same chart for convenience.

d-similarity. Most benchmark's application error degrades gracefully, up until a certain threshold where the application's error increases sharply. A main cause of this is when the threshold value approaches the application's variable size. For example, in `blackscholes`, which uses floating point variables, the error spikes around 23-similar. This is because the lower 23-bits is the fraction field in IEEE 754 floating point representation. When the level of *d*-similarity is too large, it will begin to quantize the majority of variable's values into a single value. At this point, most of application's data will appear similar, causing most data to be approximated.

Each application has different d-similarity error tradeoffs. We selected a level of *d*-similarity that provides very low (typically ~1%) error, well below the accepted error value of 10% standard in prior works [11, 19, 21, 22, 24, 25, 33]. The *d*-similar value are shown in table 1. We selected the largest *d*-similarity value that gives us a non-zero but very low (~1%) error. For certain applications (`knn`,`ray`), the error rate jumps modestly from zero error, and we simply pick the lowest non-zero *d*-similarity value possible.



(a) Baseline (b) Warp Approximation

Figure 7: Value Locality distribution across workloads

5.4 Value Locality Occurrences

Figure 7a shows the breakdown of instructions within the annotated region that exhibit value locality (d=0) and value similarity (d>0) for the baseline machine. We observe on average 23.8% and 34.2% of instructions exhibit value similarity and value locality, respectively. Essentially, Warp Approximation can exploit 23.8% more instructions than prior works that rely on value locality [4, 12].

Warp Approximation converts instruction with value similarity into an output operand with value locality (an operand with value locality that probably would not exist in the baseline machine). Furthermore, future consumption of this new value locality operand may spawn more operands with value locality or value similarity operands that would not have existed in the baseline machine. Figure 7b shows the breakdown of instructions within the annotated region that exhibit value locality and value similarity under Warp Approximation. We observe on average 25.7% and 54.1% of instructions exhibit value similarity and value locality, respectively. This means that 79.8% of the instructions can be an opportunity for Warp Approximation.

Note that we do not experience any runaway approximation where 100% of the instructions are converted into value locality or value similarity. The reasons is that not all instructions in a warp are dependent on the approximated result operand.

5.5 Hardware Overheads

The overhead of the additional comparison stages, the broadcasting logic, and representative thread execution logic was estimated by synthesizing its Verilog HDL descriptions using the NCSU PDK 45nm library [34]. The comparison logic consumes 12.75 mW of dynamic power and 95.83 uW of leakage power with an area of 0.017mm^2.

The broadcasting logic and representative thread logic together consumes 19.76 mW of dynamic power and 127.80 uW of leakage power with an area of 0.031mm^2.

Overall, our additional logic added an overhead of 1.7% dynamic energy and 0.0013% leakage energy compared to total execution unit energy. When placed into the context of whole chip energy, these additional logic overheads are negligible. The additional logic only accounts for 0.098% area overhead per SM. To account for wiring overheads, we conservatively assume wiring area overhead would be about 50% of the logic area overhead. Accounting for wiring overhead, we estimate 0.147% total area overhead of all the logic blocks that were needed for our approach per SM. These energy overheads are accounted for in our energy evaluation.

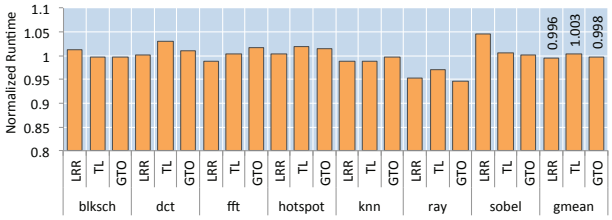
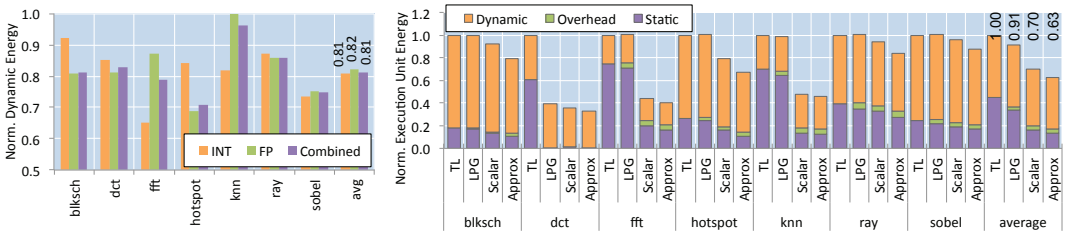


Figure 8: Performance

5.6 Performance Impact

In figure 8 we show the runtime of Warp Approximation normalized to the baseline exact architecture for each respective warp scheduler (All LRR results are normalized to LRR baseline, etc.). We show the performance of Warp Approximation with a 2-stage delay for the operand value comparison stage. On average, Warp Approximation adds no performance overhead and performs similarly across all warp schedulers used.

Note the performance improvement of Warp Approximation in `ray`. In `ray`, the kernel encompasses a large number of loops and branches. By utilizing representative values, we can sometimes avoid warp divergence, leading to some performance gains.



(a) Dynamic Energy Savings

(b) Total Energy Savings

Figure 9: Execution Unit Energy evaluation

5.7 Execution Unit Energy

Dynamic Energy: Figure 9a shows the dynamic energy consumption for the integer unit, floating point unit, and the total combined execution units. The dynamic energy is normalized to the baseline precise architecture. Dynamic energy is a function of the number of execution unit accesses, therefore it is independent of the warp scheduler used. This dynamic energy already accounts for the dynamic energy overhead of the proposed hardware additions.

For certain workloads, such as `hotspot`, we see a significant reduction in dynamic energy. This corresponds to the increased occurrence of value similarity as shown in figure 7. As discussed earlier, Warp Approximation leads to dynamic energy saving whenever value similarity is detected within an approximate region. Instead of computing accurately on 32 execution units for a warp, we instead pick a representative thread to compute on a single execution unit. On average, we save 19% dynamic energy in the integer units, 18% in the floating point units, and 19% overall.

Static Energy: Warp Approximation can also be exploited to reduce leakage energy by enhancing execution unit power gating [16, 35]. Figure 9b shows the overall energy consumption breakdown of the SP units using the two-level scheduler. The energy is broken down into dynamic energy, static energy, and overhead energy (power gating overheads and hardware overheads). For these results we assume 10 cycle idle detect, 3 cycle wakeup delay and 14 cycle breakeven time for power-gating a lane as has been used in prior studies [16]. We assume that lane power gating (LPG) exist in the hardware as in [16]. By only enabling lane power gating (labeled as LPG) on the base machine (labeled as TL), static energy is reduced from 45.0% to 33.5% of total runtime energy. The effectiveness of lane power gating is proportional to the level of divergence in the workloads. Therefore, if workloads have low divergence, then the effectiveness of lane power gating is reduced. When a warp is approximated, the effective active mask for execution will be 1 for the representative thread lane, and 0 for all other lanes. Therefore, Warp Approximation induces more idleness to SIMT lanes, independent of application divergence levels, which can translate to more power gating opportunity for the power gating controller. With Warp Approximation, the static energy is slashed from 45.0% to 13.5%, a 3.3x reduction.

Overall Energy: Warp Approximation is able to reduce the total execution unit energy by 37% with minimal performance overhead and quality degradation

Comparison to related work: Figure 9b also contains the energy savings for scalar execution units [12] (labeled Scalar) where warps with input operand exhibiting value locality are scalarized and executed on a dedicated scalar execution unit. We also augment the vector SP units with power gating in the scalar execution unit architecture. Scalar execution unit is able to save 30% of the total execution unit energy. In contrast, Warp Approximation can approximate not only scalar vectors, but also vectors with value similarity. Warp Approximation can save a total of 37% execution unit energy, a 23% relative savings increase over scalar execution units. This relative increase is in line with the relative increase in value similarity opportunity seen earlier, therefore, Warp Approximation was successful in taking advantage of the value similarity opportunities.

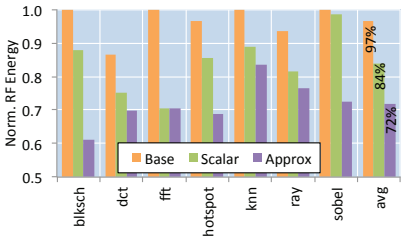


Figure 10: Register File Energy Savings

5.8 Register File Energy

We evaluate the energy savings in the register file by enabling clock gating in the register file [31]. In figure 10, we show the register file energy normalized to the base case without register file clock gating. In the baseline case with clock gating enabled (labeled Base), we observe limited energy savings of 3% on average. It has been observed that register file energy savings is proportional to the level of divergences in warps [31]. Our workloads have low levels of divergence, which also impacted the effectiveness of lane-level power gating as shown in figure 9b. We compare Representative Value Storage to a Scalar register file [12] which can only exploit value locality. Scalar register file can save 16% of the register file energy. Representative Value Storage enables register access energy savings by only reading and writing value similar scalars to a single representative thread. The overheads for broadcasting and the dummy MOV instructions were modeled in the simulator and are accounted for in all results in this section. In figure 10, Representative Value Storage (labeled Approx) can save 28% of the register file energy on average; extracting 75% more savings relative to Scalar register file.

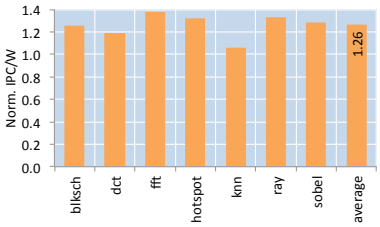


Figure 11: Energy Efficiency Improvements

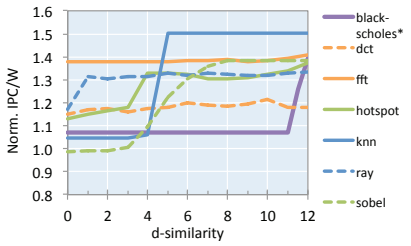


Figure 12: *d*-similarity vs Energy Efficiency Tradeoff

5.9 Whole GPU Energy Efficiency

To evaluate the impact of Warp Approximation on the entire GPU, we look at the energy efficiency of the GPU. We define energy efficiency as IPC/W. Figure 11 shows the energy efficiency improvement of Warp Approximation normalized to the energy efficiency of the baseline machine. Warp Approximation achieves energy efficiency improvements across all workloads. Over 50% of the SM power is consumed by execution units and register file [31], both of which Warp Approximation targets. Warp Approximation can achieve an average of 26% improvement to energy efficiency, all with minimal quality degradation.

5.10 Energy vs *d*-similarity tradeoffs

Figure 12 shows the energy efficiency tradeoff as we vary the level of *d*-similarity. The corresponding error for each *d*-similarity level was presented in figure 6. With a *d*-similarity of 0, Warp Approximation can only exploit value locality and essentially converts these values into scalars. As we increase the level of *d*-similarity, we find that the energy efficiency improvements correlates to where error increases in figure 6. That is, wherever we see a "knee" in the error tradeoff curve is also where we see the knee in the energy tradeoff curve, demonstrating that Warp Approximation effectively tradeoff error and energy. One workload of note is `fft`, where the energy efficiency curve stays relatively flat across all *d*-similarity. Recall that we fed `fft` with a random dataset where value similarity is rare. Therefore, the majority of the energy savings opportunity in `fft` comes from value locality. This behavior was also reflected in the register file energy results, in figure 10, where Warp Approximation and Scalar register file have the same savings. Overall, Warp Approximation achieves significant energy efficiency improvement across all levels of *d*-similarity.

6. DISCUSSION ON ERROR BOUNDING

Currently, there are no mechanisms preventing unbounded errors in Warp Approximation. Similar to other approximate computing work, it is left up to the programmer [11, 17, 21, 24] or compiler [19, 22] or runtime [18, 20] to identify levels of *d*-similarity and approximate regions that satisfies the desired level of output quality.

For approximate computing techniques that builds on non-deterministic hardware (where hardware errors manifest from voltage overscaling or process variation), it is difficult to guarantee a level of output quality [17,21,24,36]. The errors in hardware are unpredictable, and manifest in inconsistent ways, creating difficulty in anticipating and controlling quality. In Relax [17], a runtime will replay approximate regions until the errors are below a certain desired threshold.

For approximate computing techniques that builds on deterministic hardware, profiling or runtimes can be used to converge on approximation parameters that can meet a target quality [18–20, 22]. For example, in SAGE [19], during compilation multiple approximate kernels are generated based on various approximation parameters. During runtime, a tuning stage is used to prune out the kernels that violates the desired quality of result and to select the kernel with the best speedup without violating a target output quality. Similarly, the deterministic nature of Warp Approximation can leverage such compiler/runtime framework to provide quality guarantees.

Even in the domain of precise execution, many "precise" applications also require significant hand tuning from programmers in order to achieve satisfactory results. For example, all numerical applications and floating point operations have errors in the model and computation. In these scenarios, programmers hand tune the models, for example, by adjusting time steps, until the results converge.

Warp Approximation can not guarantee output quality without the help of compiler or runtime-level techniques. From the hardware point of view, application error is not observable due to the need for a baseline to compare against. What is controllable in hardware is the level of *d*-similarity. We found that it is possible for Warp Approximation to provide statistical guarantees on application output error by identifying microarchitectural observations that correlates well to application-level error.

6.1 Providing Statistical Error Bound Guarantees in Warp Approximation

To this end, we performed a sensitivity analysis in order to identify any correlation that exists between microarchitecturally available statistics and application output error, in order to statistically guarantee an application error target.

We discovered that independent of applications, the average rate of new instructions with output `SimilarBit` set through the similarity detector (figure 2a) is strongly correlated with application output error. A new instruction with output `SimilarBit` set occurs when an instruction has inputs operands which don't all have *SimilarBit* set. This represents new approximation opportunities. Recall, that if an instruction has all input operands with `SimilarBit` set, then the output `SimilarBit` may be automatically set if the input and output *d*-similarity level is the same. This represents approximation opportunities that already exist and reused.

Figure 13 shows our observation. The x-axis shows the approximation rate which we define as new approximation

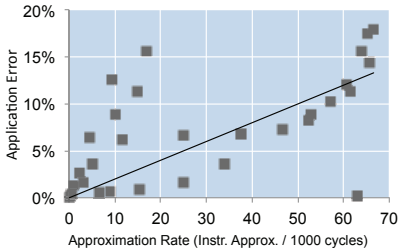


Figure 13: Strong relationship exists between approximation rate and application output error

opportunities detected per 1000 cycles, and the y-axis shows the application-level output error as observed in figure 6. The solid line represents the linear regression fit model. Here we see a fairly strong linear correlation with coefficient of determination value of 0.7. Therefore, this suggests that it would be reasonably possible to probabilistically meet a given application level error by regulating the rate of instructions approximated.

For instance, if the application error rate is set at 10%, roughly 50 instructions are having `SimilarBit` set every 1000 cycles. On the other hand, if the application error rate is capped at 5%, then only 25 instructions are approximated. To increase the probability that the error rate is met, one can conservatively offset the number of instructions approximated (ex. use 40 instructions approximated for 10% error).

This can also simplify programmer effort, by simply requiring programmers to just specify a target application error target bound, which the API will convert to a `SimilarBit` target rate based on our linear regression equation. At runtime, the *d*-similarity will be adjusted on the fly to either increase or decrease the observed *SimilarBit* rate. An increase in *d*-similarity would result in more `SimilarBit` detections due to a more relax definition of value similarity, and vice versa for a decrease in *d*-similarity.

7. RELATED WORK

Hardware Approximation techniques: Approximate computing techniques, such as Truffle [24], NPU [22], QPP [20], and stochastic processors [36], are all hardware-based approximation techniques where application errors manifest due to hardware mechanisms. Similar to these techniques, Warp Approximation is also a hardware-based approximation technique. Unlike prior hardware approximation techniques, Warp Approximation leverages intra-warp operand value similarity to trade off quality and energy. Recently, techniques were proposed to approximate load values [11, 37]. Our technique is orthogonal to load value approximation as our techniques target the execution units and register files rather than memory components.

Software Approximation techniques: Software-level approximation techniques tends to rely on transforming programs to achieve approximation. For example, NPU [22] transforms functions into hardware-accelerated neural networks, Paraprox [38] approximates data-parallel patterns, Loop Perforation [26, 39] approximates loops by selectively skipping iterations, Green [25] enables programmers to approximate expensive functions and loops. Most similar to our work is SAGE [19], where adjacent threads are fused into a single thread that is execute with the results forwarded. SAGE relies on the compiler to fuse threads and identify which threads can tolerate fusion. In Warp Approximation, threads are fused dynamically in hardware, and the decision as to when to approximate is guided by the live values at runtime.

Power Gating in GPGPUs: Power gating of GPGPU execution units are presented in Warped Gates [35] and PATS [16]. Warped Gates can power gate execution units at the SP-level. PATS introduced lane-level power gating which can take advantage of idle lanes due to warp divergence. We utilize the lane-level power gating mechanism in Representative Thread Execution to save static energy in execution units. Clock gating of register files was presented in [31]. We implemented and use this technique to save energy in the register files for Representative Value Storage.

Exploiting Scalars in GPGPU: Value locality of GPUs has been explored through scalarizing of uniform vectors [12]. In addition to the scalar execution unit and scalar register file, which we compared against in our evaluation, [12] also presented sliced data path so the execution unit can dual execute instructions simultaneously. Our work is orthogonal and can be applied on top of sliced data path, amplifying the benefits of Warp Approximation. Similarly, Warp Approximation is also able to avoid redundant computation, but our technique is also able to reduce redundant "similar" computation, allowing us even more opportunity for power savings. In addition, Warp Approximation does not require the area overhead associated with additional scalar execution unit and scalar register files.

[40] has also explored reducing redundant execution in GPGPUs through fragment memoization. In contrast to our technique, this work does not approximate any computation, and exploits temporal redundancies while our technique exploits spacial redundancies. [41] exploited value structures based on compact affine execution of arithmetic, branch and memory instructions in GPGPUs. Although values may be affine, they may not necessarily exhibit value similarity, which is what our work exploits. Similar to our work, affine execution can also take advantage of uniform vectors.

8. CONCLUSION

In this paper we identified a type of value similarity prevalent in GPUs, called intra-warp operand value similarity. To demonstrate an application for intra-warp operand value locality, we propose Warp Approximation. Warp Approximation consists of Representative Value Storage and Representative Thread Execution. When intra-warp operand value similarity is detected dynamically in hardware, Representative Value Storage will store a single value similar scalar in the register file in place of the original 1028-bit vector. With Representative Thread Execution, rather than executing all lanes in a warp precisely, we pick a single representative thread to execute. We detailed hardware support for Warp Approximation and exposed knobs to control for degree of approximation through CUDA API and ISA extensions. We showed that Warp Approximation can reduce execution unit energy by 37% and register file energy by 28%, improving overall GPGPU energy efficiency by 26% with minimal quality degradation (typically ~1%).

Acknowledgement

This work was supported in part by NSF CCF-0954211, CCF-0953603, CNS-1217102, and DARPA (HR0011-12-2-0020). Nam Sung Kim has a financial interest in Samsung Semiconductor and AMD.

9. REFERENCES

[1] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *ISCA '97*.

[2] A. Sodani and G. S. Sohi, "Understanding the differences between value prediction and instruction reuse," in *MICRO '98*.

[3] V. Petric, A. Bracy, and A. Roth, "Three extensions to register integration," in *MICRO '02*.

[4] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," in *ICS '13*.

[5] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *ASPLOS '96*.

[6] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *MICRO '96*.

[7] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?," in *MICRO '97*.

[8] B. Calder, G. Reinman, and D. M. Tullsen, "Selective value prediction," in *ISCA '99*.

[9] R. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, and M. Valero, "A content aware integer register file organization," in *ISCA '04*.

[10] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*

[11] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *MICRO '14*.

[12] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Power-efficient computing for compute-intensive GPGPU applications," in *HPCA '13*.

[13] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS '09*.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC '09*.

[15] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," in *ISCA '15*.

[16] Q. Xu and M. Annavaram, "Pattern aware scheduling and power gating for gpgpus," in *PACT '14*.

[17] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *ISCA '10*.

[18] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An Online Quality Management System for Approximate Computing," in *ISCA '15*.

[19] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *MICRO '13*.

[20] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality Programmable Vector Processors for Approximate Computing," in *MICRO '13*.

[21] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: approximate data types for safe and general low-power computation," in *PLDI '11*.

[22] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *MICRO '12*.

[23] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," in *ASPLOS '12*.

[24] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS '12*.

[25] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *PLDI '10*.

[26] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *ESEC/FSE '11*.

[27] NVIDIA Corporation, 2011. CUDA Toolkit 4.0.

[28] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *GPGPU '10*.

[29] P. Rogers, "cuda-samples/sobel," 2010. github.com/hellopatrick/cuda-samples/tree/master/sobel.

[30] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *MICRO '11*.

[31] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi1, "Gpuwattch: Enabling energy optimizations in gpgpus," in *ISCA '13*.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO '09*.

[33] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," in *ISCA '14*.

[34] "The freepdk process design kit." http://www.eda.ncsu.edu/wiki/FreePDK.

[35] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for gpgpus," in *MICRO '13*.

[36] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE '10*.

[37] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry, "Rollback-free Value Prediction with Approximate Loads," in *PACT '14*.

[38] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS '14*.

[39] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures.," Tech. Rep. Technical Report MIT-CSAIL-TR-2209-037, EECS, MIT, August 2009.

[40] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ISCA '14*.

[41] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," in *ISCA '13*.