Understanding Software Approaches for GPGPU Reliability

Martin Dimitrov* Mike Mantor† Huiyang Zhou*
*University of Central Florida, Orlando †AMD, Orlando
{dimitrov, zhou}@cs.ucf.edu.edu michael.mantor@amd.com

Abstract

Even though graphics processors (GPUs) are becoming increasingly popular for general purpose computing, current (and likely near future) generations of GPUs do not provide hardware support for detecting soft/hard errors in computation logic or memory storage cells since graphics applications are inherently fault tolerant. As a result, if an error occurs in GPUs during program execution, the results could be silently corrupted, which is not acceptable for general purpose computations. To improve the fidelity of general purpose computation on GPUs (GPGPU), we investigate software approaches to perform redundant execution. In particular, we propose and study three different, application-level techniques. The first technique simply executes the GPU kernel program twice, and thus achieves roughly half of the throughput of a non-redundant execution. The next two techniques interleave redundant execution with the original code in different ways to take advantage of the parallelism between the original code and its redundant copy. Furthermore, we evaluate the benefits of providing hardware support, including ECC/parity protection to on-chip and off-chip memories, for each of the software techniques. Interestingly, our findings, based on six commonly used applications, indicate that the benefits of complex software approaches are both application and architecture dependent. The simple approach, which executes the kernel twice, is often sufficient and may even outperform the complex ones. Moreover, we argue that the cost is not justified to protect memories with ECC/parity bits.

Categories and Subject Descriptors B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance; I.3.1 [*Computer Graphics*]: Hardware Architecture – Graphics processors
General Terms Performance, Reliability
Keywords GPGPU, Reliability

1. Introduction

General purpose computing on graphics processor units (GPGPU) becomes increasingly popular due to their remarkable computational power, memory access bandwidth and improved programmability. Current GPUs contain hundreds of compute cores and support thousands of light-weight threads, which hide memory latency and provide massive throughput for parallel computations. New programming models including CUDA from NVIDIA [3], Brook+ from AMD/ATI [1], and under-development OpenCL [7], facilitate programmers by allowing them to write GPU code in a familiar C/C++ environment, instead of forcing them to map general purpose computation to the graphics domain. In these programming models, the GPU is used as an accelerator to the CPU, from which memory-intensive and compute-intensive tasks are offloaded.

However, current GPUs do not provide hardware support for detecting soft or hard errors, which may occur in computation logic or memory storage. For instance, the off-chip storage of modern GPUs such as ATI Radeon HD series uses graphics double data rate (GDDR) type memories. The latest GDDR5 memory contains error detection, however, only for transmission errors [4]. As a result, any bit-flip in a memory cell may lead to silently corrupted results, i.e., erroneous results which are not detected. With soft-error rates predicted to grow exponentially [16][17] in future process generations and permanent failures/hard errors such as gate-oxide breakdown [15] gaining importance, future GPUs are likely to be prone to hardware errors. This has an adverse impact on GPGPU since many scientific, medical imaging and financial applications require strict correctness guarantees. Unfortunately, such reliability requirements are not likely to be answered in current or near future GPU generations. The reason is that even though GPGPU applications are gaining popularity, modern GPU design remains largely driven by the video games market, where 100% correct results are not strictly necessary. As observed by Sheaffer et. al. [13], errors in video applications are often masked because they affect a single or a few pixels, or the corrupted image is promptly recomputed in the next frame. As such, in order to overcome the GPU reliability limitations, we need to develop reliability schemes for GPGPU, which are software only, or require very few hardware changes to the GPU hardware.

In this work, we first propose and evaluate three different methodologies for providing redundancy entirely in software. We implemented the redundancy methods at the application level. In other words, we assume that the programmer is responsible for writing redundant code.

GPGPU'09 March 8, 2009, Washington, DC, USA.

However, we believe that the approaches are relevant and insightful to possible compiler implementations like [10] due to their regular patterns. The first technique, which we call R-Naïve, simply executes the GPU computation (or GPU kernel) twice. This approach is simple, but naturally leads to roughly half of the throughput compared to a non-redundant version. The other two approaches, R-Scatter and R-Thread, interleave redundant execution with the original program code in different manners. R-Scatter takes advantage of unused instruction-level parallelism, while R-Thread utilizes available thread-level parallelism. The goal of R-Scatter and R-Thread is to better utilize the vast computational resources of the GPU and achieve lower performance overhead by leveraging the inherent parallelism between the original and redundant execution. Note that although leveraging unused instruction or thread-level parallelism has been proposed for central processing units (CPU) redundancy [6][8][9][11][12], we adapt the principle to GPUs and evaluate how well it may works for GPGPU redundancy. For each of the three approaches, we further consider whether adding hardware support, including ECC/parity bits to on-chip static random access memory (SRAM) and off-chip dynamic random access memory (DRAM), is justifiable. Interestingly, our experience with six commonly used applications on two GPU architectures, indicate that the benefits of the R-Scatter and R-Thread approaches are both application and architecture dependent. In comparison, R-Naïve has consistently predictable performance and may even outperform R-Scatter or R-thread in some cases. Moreover, we argue that the area and cost overhead of protecting SRAM and DRAM memories with ECC/parity bits are not justified, because they do not result in significant performance improvements for our software redundancy approaches.

The rest of this paper is organized as follows. In Section 2 we briefly review the basics of GPU architecture and GPGPU programming. In Section 3, we describe the proposed application-level software redundancy approaches. In Sections 4 and 5, we discuss our experimental methodology and results. Section 6 concludes the paper and addresses the future work.

2. GPU Architecture and Programming

In this section, we provide a high-level overview of modern GPU architectures and programming models. This overview is sufficient for the reliability techniques presented in this paper. More in-depth treatment of GPU architectures and programming models can be found in [1][3].

2.1. GPU Architectures

In this work, we experiment with two different GPU architectures, namely NVIDIA G80 and AMD/ATI R670 architectures. We first describe G80, followed by R670, highlighting their differences and similarities.

NVIDIA G80 contains 16 streaming multi-processors (Compute Units) and each Compute Unit contains 8 steaming processors (SPs) or compute cores, for a total of 128 cores. In addition, each Compute Unit has an 8K-entry (32kB) register file, 16kB read/write shared memory and 8kB read-only constant memory. Shared and constant memories are on-chip, fast access SRAM, managed explicitly by the programmer in software. The main memory storage, or global memory, is off-chip DRAM and takes hundreds of cycles to access. In order to hide global memory access latency, Compute Units can be assigned a large number of light-weight threads, up to 512 threads per Compute Unit. Thus, while some threads are waiting for global memory access, other threads can execute on the Compute Unit and overlap memory latency. Creating more threads is generally better for achieving high performance. However, the programmer has to be aware of the hardware limitations of the GPU. For instance, the number of registers or shared memory that each thread requires may limit the number of threads that can be created. If the programmer wants to assign 512 threads to a Compute Unit, then each thread should consume at most 16 registers (8k-registers / 512 = 16). In G80, threads are scheduled for execution in granularity of warps. Each warp consists of 32 threads. Threads within a warp share the same program counter (PC), following the single-instruction multiple-data (SIMD) mode. Different warps execute the same program (kernel) but maintain their own PC, thus following the single-program multiple-data (SPMD) mode.

The architecture of AMD/ATI R670 is similar to G80 in many respects. R670 contains 320 compute cores. The compute cores are organized into 4 Compute Units of 16x5 cores per Compute Unit. Similar to warps in G80, threads in R670 are organized into wavefronts. Each wavefront contains 64 threads and follows the SIMD mode, while different wavefronts follow the SPMD mode. In contrast from G80, in R670 instructions are grouped into VLIW words of 5 instructions. Thus, in order to fully utilize the computational resources of the machine, the compiler needs to find enough instruction-level parallelism to fully pack the VLIW words. R670 does not contain a shared memory. However it has two levels hardware managed caches (32kB L1 and 128kB L2 shared by all Compute Units) and a larger (1MB total or 256kB per Compute Unit) register file.

2.2. GPGPU Programming Models

The NVIDIA programming model is called CUDA [3] and the AMD/ATI programming model is called Brook+ [1]. In both programming environments, the GPU is viewed as an accelerator to the CPU. The overall process required to write a GPGPU program for both environments is summarized as follows. First, the programmer writes a kernel function(s) in extended C/C++. The extensions to C/C++ are necessary in order to control GPU specific resources, such as thread ids, vector operations or shared-memory variables. The kernel executes on the GPU and forms the task that a single GPU thread has to complete.

Thus, following the SPMD mode, every thread on the GPU will execute the same kernel, however depending on the thread id (CUDA) or index (Brook+ 1.0) each thread will work on a different portion of the data. Second, on the CPU side, the programmer allocates and initializes the problem data. Then the programmer writes code to transfer the data from the CPU to the GPU. Finally, the programmer invokes the GPU kernel and then copies the results back from the GPU to the CPU. The memory transfer from CPU-GPU-CPU is achieved using intrinsics such as "streamRead/streamWrite" in Brook+ 1.0 and "cudaMemCopy" in CUDA. In this work, for simplicity we refer to the CPU-GPU-CPU memory transfer as memcopy.

While CUDA and Brook+ 1.0 share many similarities, the most notable difference between the two is their thread management model. In CUDA, threads are created explicitly and arranged into a thread hierarchy. For instance, threads are grouped into three dimensional thread blocks. Thread blocks are further arranged into a two dimensional grid. At runtime, each thread block will be assigned to an available Compute Unit for execution. The threads find their working data, by querying their thread id and block id. In contrast, Brook+ 1.0 adopts a streaming model, in which the kernel operates concurrently on each data element of an input stream(s) and updates the corresponding element of an output stream(s). Thus, thread creation in Brook+ 1.0 is implicit. An implicit thread is created for each streaming element, and the thread operates on that element. The number of threads created depends on the size of the data stream. Since the pure streaming model is too restrictive, in Brook+ 1.0 there are exceptions which allow for scatter/gather or random access streams. Scatter and gather streams allow the kernel to access arbitrary memory locations within a stream. However, some restrictions still apply. In particular, at least one stream in the kernel (either an input or an output stream) has to be declared as a proper stream (i.e., not a scatter/gather stream). This is because the proper stream drives the implicit creation of threads. Similar to "thread id" in CUDA, in Brook+ 1.0 we use "index of" to find out which data element we are working on. Later versions of Brook+ (version 1.3) add flexibility for explicit thread creation.

3. Proposed Software Redundancy Approaches

In this section we describe the three approaches that we propose for software-only reliability. We also discuss how each approach can be enhanced by adding ECC/parity protection in off-chip global memory and on-chip memory. The R-Naïve approach simply executes the kernel twice and then compares the two results for discrepancy. The R-Scatter and R-Thread approaches are motivated by the fact that even a highly optimized GPGPU application is likely to leave some unused instruction or thread-level parallelism. These two approaches, attempt to take advantage of the unused parallelism in order to reduce the performance overhead of redundant execution. However, because the two different architectures, G80 and R670, achieve high performance in different ways, some of the reliability approaches are more suitable to one architecture and not the other. For example, in R670 instruction-level parallelism is critical in order to fully pack the VLIW words with operations, which can issue together. On the other hand, in G80, instruction-level parallelism does not play such a large role and the focus is shifted to thread-level parallelism.

In both architectures, our goal is to provide 100% redundancy, and we only assume that the CPU is reliable, i.e. any GPU component – global off-chip memory, on-chip memory, ALUs, on-chip interconnect, etc, is considered unreliable. While it is difficult to prove/guarantee that a certain approach provides 100% error detection, we present a best-effort approach. Moreover, the focus of our work is to study if we can effectively reduce the performance overhead by better utilizing resources.

3.1. R-Naive

In this approach, we duplicate both the memcopy (CPU-GPU-CPU) and the kernel executions. By allocating two copies of the data on the GPU, we provide spatial redundancy for GPU memory. Executing the computation kernels twice (once for each copy of the data) provides temporal redundancy to computational logic and communication links. In order to improve reliability further and detect permanent defects, it is desirable for the original and redundant input/output streams to use different communication links and compute cores (i.e., to achieve spatial redundancy). For some applications we can easily achieve that by rearranging the input data appropriately. For example, in matrix multiplication of two matrices M and N, we can circular-shift the columns of M by 1 (the first column becomes last) and the rows of N by 1 (the first row becomes last). The resulting matrix multiplication will produce the same result as before, however we have ensured the input streams are different from the original. On the other hand, rearranging the data to obtain identical results may not as easy for other applications. In this case, it would be desirable to have a software controllable interface to assign the hardware resource. For example, if the software can specify the Compute Unit where the first thread block should be dispatched to, the original dispatch and its redundant copy can be assigned to different Compute Unit so that a permanent error will not affect both of them in the same manner. Similarly we may use this software interface to allocate redundant memory resources at a user-defined offset from the original data in order to detect permanent errors in memory. Since rearranging the input and subsequently the output is most naturally done on the CPU, while the data is initialized or consumed, we do not include this overhead when evaluating the performance on the GPU. We also duplicate the transfer of the results from the GPU back to the CPU to account for any data corruption which may occur during the transfer as well as to check the results. Once the results are transmitted to system's memory (which is ECC protected) the CPU will

compare the results and use them if they are identical. In the rare case that there is an error, the CPU may re-submit the request to the GPU or perform more elaborate checks to determine the cause of the error.

A pseudo code of implementing R-Naïve is presented in Figure 1, where the redundant code is highlighted in bold. The original version of the code is presented in Figure 1 (a) and three different implementations of R-Naïve are presented in Figure 1 (b), (c), and (d). In Figure 1 (b), we first transfer the data to the GPU twice and then invoke the computation kernels. Thus, there is no overlap between CPU-GPU transfer and kernel computation. In Figure 1(c), the redundant memory transfer is after the asynchronous kernel invocation, which in theory means that *StreamRead(in_R)* can be executed without the completion of *Kernel(in,out)*. However, in practice we did not observe any performance difference between the two approaches. We believe that this is a limitation of the programming environments, which are still being refined, and not a hardware limitation. In Figure 1 (d), the original code and the redundant code run back-to-back. So, the original data and its redundant copy do not need to reside in the device memory at the same time. However, it may happen that the redundant data are loaded to the same memory location as original data. In this case, if the redundant data is not rearranged (or allocated at an offset using a software interface as mentioned above), we may lose the capability to detect permanent memory errors. In terms of performance, we did not observe any difference between Figure 1 (d) and Figure 1 (b) or (c). Therefore, the approach shown in Figure 1(c) is the preferred one as long as the device memory can hold both the original data and its redundant copy simultaneously.

StreamRead(in) Kernel(in, out) StreamWrite(out)	StreamRead(in) **StreamRead(in_R)** Kernel(in,out) **Kernel(in_R,out_R)** StreamWrite(out) **StreamWrite(out_R)**	StreamRead(in) Kernel(in,out) **StreamRead(in_R)** **Kernel(in_R,out_R)** StreamWrite(out) **StreamWrite(out_R)**	StreamRead(in) Kernel(in,out) StreamWrite(out) **StreamRead(in_R)** **Kernel(in_R,out_R)** **StreamWrite(out_R)**
(a) Original Code	(b) Redundant code without overlap	(c) Redundant code with overlap	(d) Redundant code back-to-back

Figure 1. Pseudo code for R-Naïve, with and without overlapping memory transfers and kernel computation, and back-to-back execution. Extra code added for redundancy is in bold.

3.2. R-Scatter

In this approach, we try to take advantage of unused instruction-level parallelism. Even though this approach has some potential benefits for the G80 architectures, which we describe shortly, it is more suitable to the VLIW model of R670. The idea, as applied to R670, is illustrated in Figure 2. From Figure 2 (a), we can see that due to data dependencies, the original VLIW schedule of the GPU kernel is not fully packed, thus not utilizing the hardware. On the other hand, since the redundant instructions are inherently independent from the original code, we can interleave them and create more compact schedules, as shown in Figure 2 (b).

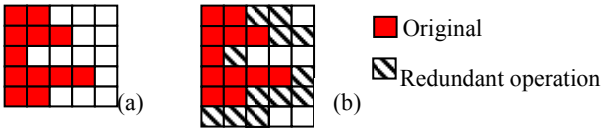

Figure 2. Original vs. R-Scatter VLIW instruction schedules.

To see how a programmer can implement this approach, we present a code sample from a simple matrix multiplication kernel in Figure 3. We first show how this approach can be implemented in Brook+ 1.0 and discuss CUDA shortly. Brook+ 1.0 uses a streaming compute model and an implicit thread is created for each proper stream element. In Figure 3 (a), the output stream P is a proper stream and the input streams M and N are gather streams. We first obtain the position into the output stream, using "*indexof*". Then, in the *for*-loop, we use this position to compute an index into the input matrices M and N and perform the computation. Figure 3 (a) also demonstrates the use of vector types and swizzle operations, which allow access to the elements of a vector in any order. In Figure 3 (b), we show the code implementing R-Scatter. The redundant code is highlighted in bold. We supply redundant input and output streams to the kernel. We also duplicate the computation within the *for*-loop and write the results to a redundant output stream. The operations within the *for*-loop are inherently independent, thus they may result in better utilized VLIW instruction schedules, which is the insight behind R-Scatter. In this particular example we have 7 VLIW words in the original and 11 VLIW words in the R-scatter version, respectively. Moreover, in R-scatter code, the input stream accesses of the original and redundant execution are overlapped, achieving higher memory-level parallelism in the kernel function.

Notice that in Figure 3 (b) we do not duplicate all the kernel code. Instead, we reuse some kernel code for redundant execution. For instance we reuse the *for*-loop and the index computation, which leads to a reduced number of redundant instructions compared to R-Naïve. However, such reuse can compromise the reliability, since an error to the variable "*i*" will affect both the original and redundant computation. To prevent this from happening, we want the kernel to compute different data elements in the original and the redundant streams so that an error to "*i*" will cause errors in different elements in the output data and its redundant copy. Computing different data elements in the original and the redundant streams can be implemented in either of the following two ways. The first is to rearrange the input data for redundant execution as discussed in Section 3.1. The second is to use the same input data but manipulate the thread index. The first approach is more suitable for Brook+ 1.0 due to its relatively restrictive streaming program model, in which the threads are implicitly generated based on the proper stream. To make a thread to process different data items in two output streams

```
kernel void mat_mult(float width, float M[][], float N[][], out float P<>){

    float2 vPos = indexof(P).xy;  // obtain position into the stream
    float4 index = float4(vPos.x, 0.0f, 0.0f, vPos.y);
    float4 step = float4(0.0f, 1.0f, 1.0f, 0.0f);
    float sum = 0.0f;

    for(float i=0; i<width; i= i+1){
      sum += M[index.zw]*N[index.xy]; //accessing input stream
      index += step;
    }
    P = sum;
}
```

(a) Original code

```
kernel void mat_mult(float width, float M[][], float M_R[][],
                     float N[][], float N_R[][],out float P<>, out float P_R<>){
  float2 vPos = indexof(P).xy;
  float4 index = float4(vPos.x, 0.0f, 0.0f, vPos.y);
  float4 step = float4(0.0f, 1.0f, 1.0f, 0.0f);
  float sum = 0.0f;
  float sum_R = 0.0f;
  for(float i=0; i<width; i= i+1){
    sum += M[index.zw]*N[index.xy]; //accessing input stream
    sum_R += M_R[index.zw]*N_R[index.xy]; //accessing input stream
    index += step;
  }
  P = sum;
  P_R = sum_R;
}
```

(b) R-Scatter Code

Figure 3. Simple matrix multiplication kernel code in Brook+ 1.0 for R-Scatter. Code added for redundancy is in bold.

```
int tx,ty;    // Obtain 2D thread id
tx = threadIdx.x;
ty = threadIdx.y;


int Pvalue = 0;    // Store the computed elem


for (int k = 0; k<Width; ++k){
    float m = M[ty*Width + k];   // global memory access
    float n = N[k*Width + tx];    // global memory access


    Pvalue += m * n;

}
P[ty*Width + tx] = Pvalue
```

(a) Original code

```
int tx,ty, tx_R, ty_R;    // Obtain 2D thread id
tx = threadIdx.x;
ty = threadIdx.y;
tx_R = (threadIdx.x +1)%(Width) ;
ty_R = (threadIdx.y +1)%(Width);
int Pvalue = 0; // Store the computed elem
int Pvalue_R = 0;

for (int k = 0; k<Width; ++k){
    float m = M[ty*Width + k]; //global memory access
    float n = N[k*Width + tx]; //global memory access
    float m_R = M_R[ty_R*Width + k]; //global memory access
     float n_R = N_R[k*Width + tx_R]; //global memory access
     Pvalue += m * n;
     Pvalue_R += m_R * n_R;
}
P[ty*Width + tx] = Pvalue;
P_R[ty_R*Width + tx_R] = Pvalue_R;
```

(b) R-Scatter Code

Figure 4. Simple matrix multiplication kernel code in CUDA for R-Scatter. Extra code added for redundancy is in bold. The global memory accesses are indicated with comments.

requires gather and scatter streams, which have additional restrictions. For example, in Brook+ 1.0, if we want to update a different element of the redundant output stream (scatter), the output stream has to be 128 bits wide. CUDA, in comparison, supports explicit management of threads, which makes the second approach easy to implement, as shown in Figure 4. In the figure, the redundant code (b) is interleaved with the original code (a) and the *for*-loop is reused. We use explicit thread id management "*tx_R = (threadIdx.x + 1) % (Width)*" to force the kernel to compute on different elements of the original and the redundant matrices. This way, each thread computes a *different* redundant element from the *same* corresponding thread block. Similarly, if the application uses multiple thread blocks, we can also force each thread to compute the *same* element from a *different* thread block. This is achieved by re-using the thread id, but modifying the block id.

The performance benefit from R-scatter in CUDA, as seen from the example in Figure 4, mainly comes from reused instructions and overlapped memory accesses, a degree of 4 in Figure 4 (b) compared to a degree of 2 in Figure 4 (a). The tradeoff, however, is the extra registers/shared memory to hold the loaded data. Such increased register/shared memory pressure in the kernel may affect thread-level parallelism or shared memory utilization since all concurrent threads in a Compute Unit share the register file and shared memory.

3.3. R-Thread

In this approach, we try to take advantage of unused thread-level parallelism. Recall that threads are specified explicitly only in CUDA and not in Brook+ 1.0. While it is possible to create more threads in Brook+ 1.0, by combining the original and the redundant stream into a single large stream, the stream scatter restrictions of Brook+ 1.0 mentioned in Section 3.2 prevent us from implementing this approach. Thus we consider R-Thread only for CUDA. The idea of R-Thread is as follows. Each thread does the same amount of work as in the original kernel. However, we allocate double the number of thread blocks per kernel. The extra thread blocks will perform the redundant computations and are scheduled for execution on the Compute Units together with the original thread blocks. In case the original GPGPU application does not utilize all the Compute Units of G80, then the redundant thread blocks will be able to utilize those otherwise idle Compute Units and reduce the redundancy overhead. An example from the simplified version of matrix multiplication is presented in Figure 5 to illustrate R-thread. In this example, address computation is omitted for clarity. The portion of the code, which is marked in

bold, belongs to R-Thread. Intuitively, in the bold portion of the code, we check to see to which thread block the current thread belongs. If this thread block is one of the redundant thread blocks, then we simply re-direct the memory pointers to point to the redundant copies of the input and output matrices. The rest of the code remains the same and the thread will automatically compute a redundant element.

```
if(by >= NumBlocks){
  M = M_R;
  N = N_R;
  P = P_R;
  by = by - NumBlocks;
}
float Pvalue = 0;
 for (int k = 0; k < Block_Size; ++k){
   float m = M[addr_md];
   float n = N[addr_nd];
   Pvalue += m * n;
}
P[ty*Width + tx] = Pvalue;
```

Figure 5. Pseudo code of simple matrix multiplication for R-Thread. Extra code added for redundancy is in bold.

3.4. Hardware Support for Error Detection in Off-Chip and On-Chip Memories

The three software approaches proposed in this work will benefit from added hardware support for error detection/correction to off-chip (global) memory and on-chip (caches, constant, shared) memory. In G80, caching is explicitly controlled in software, i.e. the programmer specifies which data will be stored in constant or shared memory. Thus, we can decide to protect either off-chip memory only, or both off-chip and on-chip memory in hardware. If off-chip memory is protected, then duplication of the input data is not necessary and the redundant CPU-GPU memory transfers may be eliminated. In R670 caching is implicit, thus we may not separate off-chip and on-chip hardware error protection. If we protect only off-chip memory and keep only one copy of data in off-chip memory, an on-chip, cached copy of the data may be corrupted leading to incorrect computation in both original and redundant execution. Thus, in Brook+ 1.0 we assume that we can eliminate the redundant transfer from CPU-GPU only if both off-chip and on-chip memory is parity/ECC protected.

We may use either the CPU or GPU to perform result comparison for error detection. If using CPU, we still need to transfer the redundant computation results from GPU-CPU. When using GPU, although the redundant GPU-CPU transfer can be removed, there is a penalty on reliability as an error may happen in the GPU when it is busy comparing results. Thus, in this work we do not consider checking the results using the GPU.

R-Naïve, R-Scatter and R-Thread can all benefit from reduced CPU-GPU transfers. R-scatter on CUDA may additionally benefit from on-chip memory protection. This is because a thread block may fetch data into shared/constant memory only once, and then the interleaved original and redundant computation will use the same cached data. For R-Naïve and R-Thread, the data sharing among original and redundant codes is not applicable as they are treated as separate and independent threads.

Hardware protection for off-chip and on-chip memories is advocated by Sheaffer et al. [14] and they also propose hardware protection for computation logic by using two compute cores to perform the same computation. As discussed in Section 1, such extra hardware investment and reduced computation throughput may be hard to justify as graphics applications are fault tolerant and hardware errors are rare events. In our work, we analyze how much the hardware memory protection benefits our software redundancy approaches.

4. Experimental Methodology

We evaluate our proposed approaches by providing redundancy to six commonly used applications from different fields, as shown in Table 1. When available, we used the source codes distributed with CUDA and Brook+ 1.0 development samples, otherwise we coded and optimized our own versions. Among Brook+ 1.0 codes, *matrix multiplication*, *black scholes*, *Mandelbrot* and *bitonic sort* are from the Brook+ 1.0 samples. Among CUDA codes, *black scholes* is from the version in the CUDA samples. The rest of the applications were coded and optimized with our best effort. Our version of *matrix multiplication* on CUDA [2] achieves 149 GFLOPS for 2kx2k matrices, significantly outperforming the carefully tuned Nvidia CUBLAS library implementation as well as the existing CUDA sample code for matrix multiplication.

The Brook+ experiments were conducted using Brook+ 1.0 Alpha on a PC running Windows XP, with Intel Core2 Quad CPU at 2.4 Ghz with 3.25GBytes of RAM and an ATI R670 card with 512MB memory and 825MHz core clock frequency. The CUDA experiments were performed using CUDA SDK 1.1 on a Linux workstation with quad core Intel Xeon at 2.3 GHz and 2GBytes of RAM and an Nvidia GTX 8800 card with 768 MB memory and 575MHz core frequency. Both machines have PCIe x16 to provide 3.2 GB/s bandwidth between the GPU and CPU. Each execution time is collected using an average of 100 runs of the same code.

Table 1. Evaluated Applications

Benchmark Name	Description	Application Domain
Matrix Multiplication	Multiplying two 2k by 2k matrices	Mathematics
Convolution	Applying a 5x5 filter on a 2k by 2k image	Graphics
Black Scholes	Compute the pricing of 8 million stock options	Finance
Mandelbrot	Obtain a Mandelbrot set from a quadratic recurrence equation	Mathematics
Bitonic Sort	A parallel sorting algorithm. Sort 2^20 elements	Computer Science
1D FFT	Fast Furrier Transform on a 4K array	Mathematics

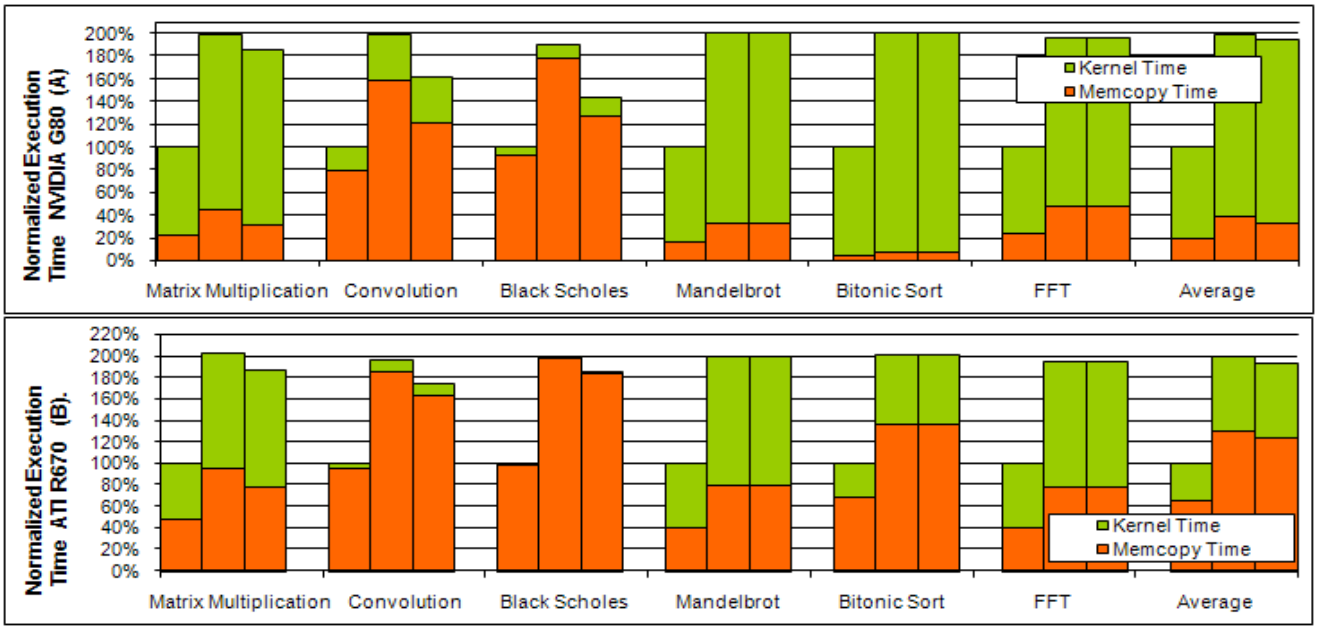

Figure 6. Execution time of R-Naïve for NVIDIA (A) and ATI (B). We show 3 bars for each application: Original, R-Naïve, and R-Naïve with hardware DRAM protection. Execution time is normalized to Original.

5. Experimental Results

In this section, we evaluate our proposed approaches – R-Naïve, R-Scatter and R-Thread. For each of them, we first discuss the performance results and then address the impact of hardware support for memory protection.

5.1. R-Naive

In this experiment, we evaluate the performance overheads of providing redundant execution using R-Naïve. For each benchmark, we present the original execution time, the R-Naïve execution time, and the R-Naïve execution time with hardware DRAM protection, all normalized to the original execution time. Each execution time is also broken down to kernel computation time and memory copy time and the results are shown in Figure 6 (a) and (b) for NVIDIA G80 and AMD/ATI R670, respectively. The last set of bars in the figure is the average across the six benchmarks, which is computed as the arithmetic mean of the execution times for each configuration and then normalized to the arithmetic mean of original execution without redundancy.

The first observation that we can make from this is that the execution time of R-Naïve is consistently close to 2x the original execution time: 199% for both Brook+ 1.0 and CUDA. This behavior is expected, since we duplicate both memcopy as well as kernel executions. In some cases it is possible for R-Naïve to incur less than 2x overhead. This behavior is due to pipeline setup times in the GPU. Since in R-Naïve, we issue two copies of the kernel so that some of the pipeline setup overhead can be avoided.

The second observation is that protecting global memory with ECC bits, thus eliminating one CPU-GPU memory transfer as discussed in Section 3.4, helps some applications much more than others. The reason is that some applications spend a much larger fraction of their execution time moving data between the CPU and GPU. For example, in *matrix multiplication* memcopy accounts for 22% of the total execution time on CUDA. On the other hand, the memcopy time for the benchmark *black scholes* is as much as 92% on CUDA, as seen from Figure 6. Thus eliminating a memory transfer is much more important for some applications than others. Another subtle issue that we noticed was that the memcopy time from GPU back to the CPU is longer than the memcopy time from CPU to GPU. This effect is more pronounced in Brook+ 1.0 than CUDA, due to the early development stage of the tool (in hardware both G80 and R670 use high bandwidth PCIex16 bus to communicate with the CPU). Since, we do not eliminate the redundant memory transfer of results from the GPU back to the CPU, the benefit of protecting global memory with ECC is diminished. Furthermore, as obvious from Figure 6, some applications such as *mandelbrot*, *bitonic sort* and *fft* do not reap any performance benefit from protecting global memory with ECC. For instance, the *mandelbrot* application has small input data sets that result in large output data that is always replicated. In *bitonic sort* and *fft*, the application performs multiple passes, where the inputs and outputs are toggled in each pass, i.e. the outputs of one pass become the inputs of the next pass. Since the outputs are always duplicated, this forces us to duplicate the inputs as well even if the memory is protected. On average, the execution time of R-Naïve with hardware protection is 192% and 194% of the original for Brook+ 1.0 and CUDA respectively. Compared to non-memory-protection R-Naïve, the performance gains (7% the original for Brook+ 1.0 and 5% the original for CUDA) do not well justify the hardware cost.

Table 2. VLIW words, Texture operations and General Purpose Registers used in the original vs. R-Scatter kernel.

Benchmark Name	Original VLIW	Original TEX	Original GPR	R-Scatter VLIW	R-Scatter TEX	R-Scatter GPR
Matrix Multiplication	33	8	15	64	16	32
Convolution	21	3	8	29	6	12
Black Scholes	66	5	7	111	10	12
Mandelbrot	19	0	9	33	0	15
Bitonic Sort	39	2	4	46	4	5
1D FFT	72	4	6	78	8	8

5.2. R-Scatter

5.2.1. R-Scatter on ATI R670

For AMD/ATI R670, we found strong evidence that R-Scatter results in better, more compact VLIW schedules as shown in Table 2, which lists a summary of the number of VLIW words, texture operations and general purpose registers used in the original vs. the R-Scatter version of the kernels.

From Table 2 we can see that R-Scatter results in significantly better utilized schedules. Note that the number of TEX operations is always double in R-scatter, which shows that we perform each data input redundantly. Taking the benchmark *bitonic sort*, as an example, the original kernel has 39 VLIW words compared to only 46 VLIW words in the R-Scatter version. Using the ATI Shader Analyzer we verified that the significantly smaller number of VLIW operations is due to the fact that most VLIW words are much better packed in R-Scatter than the original version. Figure 7 shows an extract of the VLIW schedules for the original kernel (a) and the R-Scatter (b) kernel. From the figure, we can see that only about 2-3 out of the 5 VLIW slots are populated in the original kernel, while the redundant kernel populates the VLIW slots fully thus utilizing the machine resources better. The kernel, which benefits the most from R-Scatter, is 1D *fft* with 72 vs. 78 VLIW instructions in the original and redundant version respectively. On the other hand, for other applications such as *matrix multiplication*, we did not observe such large improvements in the VLIW schedules. There are two primary reasons why this may happen. First, if the original scheduling is almost fully populated to begin with, this will result in less opportunity to pack better. The second reason is if the original scheduling contains a lot of control flow or transcendental operations (*sin, cos, log*, etc.). Only 1 control path or 1 transcendental operation can be handled per VLIW word. Thus, even if those operations are independent of each other, we will not be able to construct a better schedule. In the case of *matrix multiplication*, the original schedule was relatively well populated, thus limiting the benefit that we can achieve with R-Scatter.

Figure 8 shows the overall performance overhead of R-Scatter compared to a non-redundant implementation. We can see that generally those applications such as *bitonic sort* and *fft*, which resulted in better VLIW schedules, as shown in Table 2, perform significantly better, with redundant *fft* having only 40% performance overhead over the original execution. The performance of *convolution* is not significantly improved by R-Scatter, because the performance of *convolution* is dominated by CPU-GPU-CPU memory transfer and thus improvements in the kernel execution do not translate into overall speedup. *Matrix multiplication* has a very well utilized schedule to begin with and thus does not benefit much from R-scatter. In addition, we had to reduce the number of output streams used in the optimized *matrix multiplication* in order to incorporate the redundant output streams, since Brook+ 1.0 has a restriction of 8 output streams per kernel. In *mandelbrot*, there are too many branch instructions and R-scatter was not able to create a more efficient schedule. On average, the execution time of R-Scatter is 193% the original execution time.

```
16  x: MUL_e        ____,  T1.w,  T3.z
    y: FLOOR        ____,  T0.z
    z: SETGE        ____,  T0.y,  |KC0[5].x|

17  x: CNDE         T1.x,  PV16.z,  T0.y,  T0.w
    y: FLOOR        T1.y,  PV16.x
    z: ADD          T0.z,  PV16.y,  0.0f

18  x: MOV          T0.x,  |PV17.y|
    y: ADD          ____,  |KC0[5].x|,  PV17.x
    w: MOV/2        ____,  |PV17.y|

19  z: TRUNC        ____,  PV18.w
    w: CNDGT        ____, -T1.x,  PV18
```

(a) Original code

```
16  x: SETGE        ____,  PS15,  |KC0[5].x|
    y: ADD          ____,  T1.w,  KC0[2].x
    z: MULADD       T2.z, -T0.y,  T2.x,  T1.x
    w: ADD          ____, -|KC0[5].x|,  PS15
    t: ADD          ____,  T1.w,  KC0[8].x
17  x: ADD          ____, -|KC0[11].x|,  PV16.z
    y: SETGE        ____,  PV16.z,  |KC0[11].x|
    z: CNDE         T3.z,  PV16.x,  T1.y,  PV16.w
    w: FLOOR        ____,  PV16.y
    t: FLOOR        ____,  PS16
18  x: ADD          R2.x,  PV17.w,  0.0f
    y: ADD          ____,  |KC0[5].x|,  PV17.z
    z: ADD          R1.z,  PS17,  0.0f
    w: CNDE         T0.w,  PV17.y,  T2.z,  PV17.x
    t: MUL_e        ____,  T1.w,  T2.y
19  x: FLOOR        R0.x,  PS18
    y: MUL_e        ____,  T1.w,  T3.x
    z: ADD          ____,  |KC0[11].x|,  PV18.w
    w: CNDGT           , -T3.z,  PV18.y,  T3.z
```

(b) R-Scatter code

Figure 7. An extract of VLIW instruction schedules for bitonic sort on ATI R670. These schedules show VLIW words from 16 to 19. Each VLIW word may contain up to 5 instructions – x,y,z,w,t. The R-Scatter schedule results in more fully packed VLIW words. The instructions do not correspond exactly between the two versions, because the compiler has reordered them.

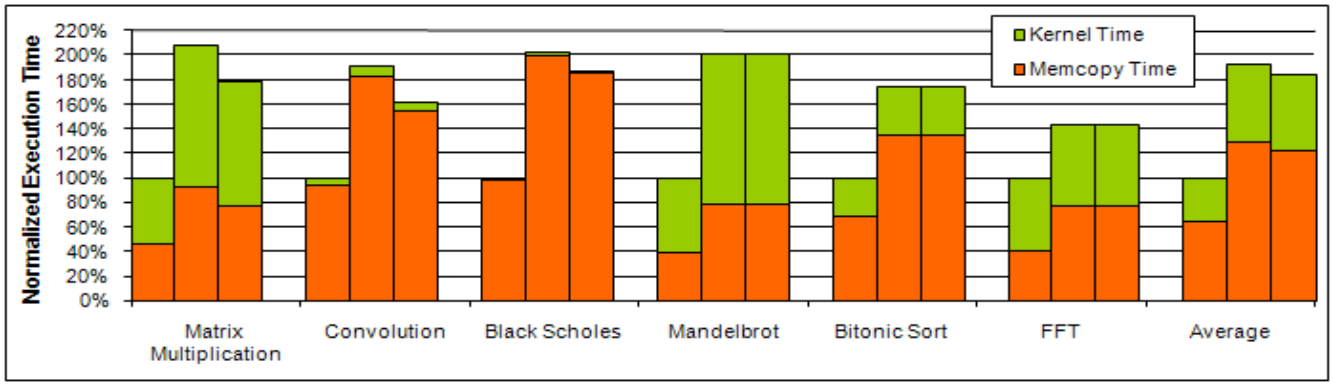

Figure 8. Execution time of R-Scatter for ATI R670. We show 3 bars for each application: Original, R-Scatter, and R-Scatter with hardware DRAM protection. Execution time is normalized to Original.

In terms of hardware protection of off-chip and on-chip memory, we did not observe a significant enough improvement (184% the original execution time on average, a 4.6% reduction compared to non-memory-protection R-Scatter), even for applications which are highly dominated by memcopy time such as *convolution* and *black schools,* to justify the implementation of ECC. The reason is that using Brook+ 1.0 it is much more expensive to transfer data from the GPU back to the CPU, than it is from CPU to GPU, as observed in Section 5.1. Since we always duplicate output streams, and we sometimes duplicate input streams, we conclude that it is not justified to include hardware support for off-chip and on-chip memory protection in ATI R670.

5.2.2. R-Scatter on NVIDIA G80

The performance overhead of R-Scatter for CUDA is presented in Figure 9. Interestingly, for all applications, the performance of R-Scatter is the same or worse than the simple R-Naïve. We observe the most evident increase in execution time in *matrix multiplication*, up to 3x. The reason for this behavior is that when we interleave the redundant code with the original code, we also impact the hardware resource usage in a negative way. The *matrix multiplication* application is a highly optimized code with large 16x256 tiles loaded in shared memory and registers, combined with prefetching [2]. When we interleaved the redundant code, we also increased the register and shared memory usage. Thus we had to reduce the tile size to 8x64 as well as the number of prefetched elements in order to satisfy the hardware limitations. Reducing the tile size and prefetching, however, has a fairly negative impact on performance.

One of the potential benefits of R-Scatter to CUDA, as discussed in Section 3.2 is overlapping independent memory accesses from the original and redundant code. The reason why latency hiding did not benefit our applications is because GPUs are much more efficient in hiding memory latency using thread-level parallelism and shared memory, than using memory-level parallelism within each thread. Since our applications are sufficiently well optimized, interleaving memory accesses using R-Scatter does not benefit and may actually hurt in some cases. To analyze this issue further, we dissect two examples. In those examples we compare R-Scatter to R-Naïve, because R-Naïve consistently incurs 2x overhead in execution time. First, we look at how program optimizations impact the benefit of R-Scatter. We evaluated a well optimized *matrix multiplication* (with 8x64 tiles, loop interchange and pre-fetching into registers, which achieves 113 GFLOPS on G80 for 2kx2k matrix multiplication) and a simple, un-optimized matrix multiplication (which does not use shared memory and each thread calculate one element in product matrix, which achieves only 3.8 GFLOPS on G80 for 2kx2k matrix multiplication). The un-optimized code is memory bound as it does not reuse data in shared memory. As a result, the

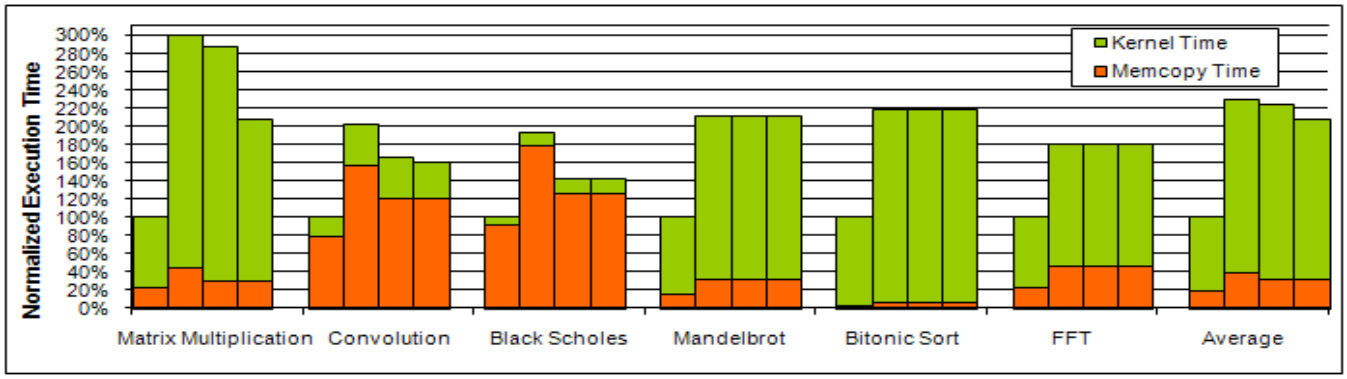

Figure 9. Execution time of R-Scatter for NVIDIA G80. We show 4 bars for each application: Original, R-Scatter, R-Scatter with DRAM and R-Scatter with DRAM and shared memory protection. Exec. time is normalized to Original.

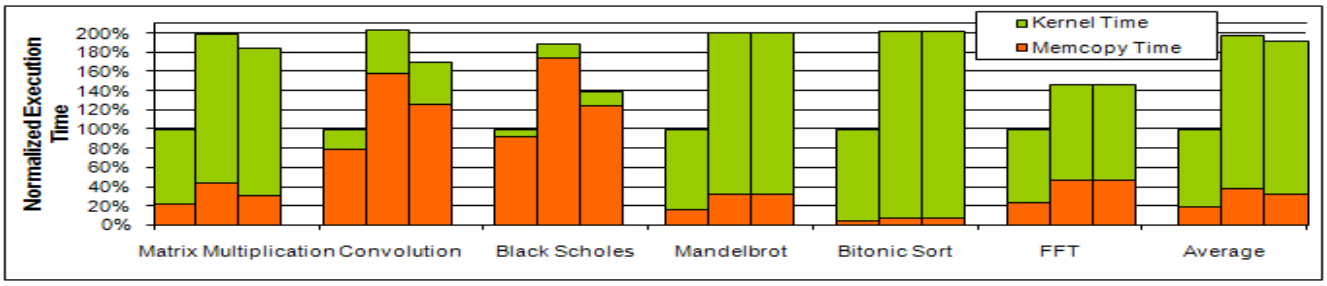

Figure 10. Execution time of R-Thread for NVIDIA G80. We show 3 bars for each application: Original, R-Thread, and R-Thread with hardware DRAM protection. Execution time is normalized to Original.

memory bandwidth is nearly saturated by the memory requests from a large number of active threads. Adding memory-level parallelism in the kernel function, R-Scatter performs about 1% better than R-Naïve on this code. On the other hand, R-Scatter performs about 17% worse on the highly optimized code than R-Naïve as the optimized code hides off-chip memory latency well using shared memory and thread-level parallelism. Introducing additional independent memory accesses to this code, does not help but actually hurts performance. In our second example, we limit the amount of thread-level parallelism available to the highly optimized *matrix multiplication*. To do that, we execute the kernel on a small input of only 64 x 64 elements matrix, which requires only 4 thread blocks. To obtain reliable timings, we looped around the kernel 100K times. The outcome of this experiment was that R-Scatter executed 15% faster than R-Naïve. The overlapping memory accesses issued by R-Scatter helped, because there was no sufficient thread-level parallelism to hide the latency. In comparison, R-Scatter is 10% better than R-Naïve on the simple version of *matrix multiplication* for 64x64 matrices since more memory bandwidth is available for redundant memory accesses than large matrices.

As discussed in Section 3.4, protecting global memory benefits applications in the same way as in R-naïve, by eliminating a redundant CPU-GPU transfer. Furthermore, R-Scatter benefits from protecting shared memory with an average of 8% speedup compared to no shared memory protection (the fourth bar of each benchmark in Figure 9). However, only *matrix multiplication* and *convolution* were able to take advantage of shared memory protection, because the other applications do not make heavy use of shared memory. However, even with shared memory protected, the *matrix multiplication* kernel of R-Scatter is still slower than R-Naïve. This is mainly due to the reduced tile size and thread-level parallelism. Since the performance of R-Scatter for CUDA was not superior to R-Naïve, and since only R-Scatter may benefit from protecting shared memory with ECC, we conclude that it is not desirable to include hardware protection to shared memory in G80.

5.3. R-Thread

As discussed in Section 3.3, given limitations of Brook+ 1.0, we were not able to apply this approach to R670 so we study R-thread using CUDA on G80. The performance results are shown in Figure 10.

As seen in Figure 10, the performance overhead of redundant execution is uniformly close to 100% (with an average of 97%), except for the benchmark *fft*. The reason why performance overhead is close to 100% is because those applications have sufficient thread-level parallelism and adding more thread blocks does not improve the overlap of memory latency. To illustrate this point better and to understand why R-Thread performs so well for *fft*, we examine the performance overhead of R-Thread on *fft* for different input sizes, as shown in Figure 11. In the figure, we also present the corresponding number of thread blocks (including redundant blocks) required to process that input size. Initially, when the input size is 4K, 8 thread blocks are needed for original execution. Thus 8 of the Compute Units (G80 has 16 Compute Units) are available to process the redundant thread blocks. This results in a complete overlap between original and redundant kernel execution and thus the performance overhead of R-Thread is small, mostly due to redundant memory transfer. However, even as the number of thread blocks increases beyond 16 and all the Compute Units are occupied, we still see substantial benefit due to thread-level parallelism and overlap of memory transfers. The reason is that the number of registers used per thread block in *fft* is not large, which allows the GPU to assign two thread blocks per Compute Unit maintaining their state in the register file concurrently. When one of the thread-blocks is stalled for memory access, the other thread block is switched in, thus hiding some of the latency. The amount of unused thread-level parallelism clearly diminishes as we grow the input size and the number of thread blocks increases. With a 64K input R-thread incurs about 95% overhead over a non-redundant execution.

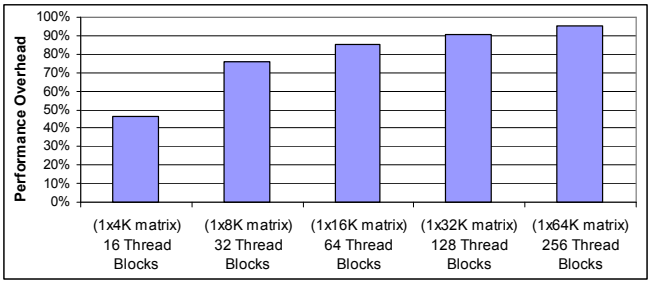

Figure 11. Performance overhead of R-Thread for various input sizes to 1D FFT.

In R-Thread, protecting global memory benefits applications in the same way as in R-Naïve, as discussed in Section 3.4. Even though in CUDA memory transfer from GPU-CPU is still slower than CPU-GPU, this effect is not as pronounced as in Brook+ 1.0, likely due to the more mature development phase of the tool. Thus applications with large memory transfer times may benefit from protecting off-chip memory in the NVIDIA G80, the overhead of *black scholes* is reduced to only 40% compared to 89% without off-chip protection. However, on average the overhead relative to non-redundant execution is 92% when off-chip memory is protected as shown in Figure 10, which is still not significant enough to justify hardware protection.

6. Conclusions

In this work, we propose and evaluate three software-only methodologies for providing redundancy for general-purpose computing on graphics processor. The first technique, R-Naïve, simply duplicates the kernel computations and naturally leads to roughly half of the throughput compared to a non-redundant version. The other two techniques, R-Scatter and R-Thread, interleave redundant execution with the original program code. R-Scatter takes advantage of unused instruction-level parallelism, while R-Thread utilizes available thread-level parallelism. Interestingly, we show that even though R-Scatter and R-Thread are quite beneficial in some cases, they also suffer from large performance overheads or increased complexity in other cases due to intricate tradeoffs among thread-level parallelism, instruction-level parallelism, data reuse with shared memory, and other factors. This means that we need to understand both the application characteristics and the hardware platform before applying software protection schemes. If the target kernel has a low degree of instruction-level parallelism and low register usage, R-scatter may be a good choice for R670. For G80, additional features like shared memory usage need to be taken into account to make sure R-Scatter will not reduce the amount of memory reuse. On the other hand, if an application does not have sufficient threads to keep all compute cores busy, R-Thread may provide substantial performance gains, especially when future GPUs feature much higher number of compute cores (240 in Nvidia GT200 and 800 for AMD R770). Detailed code analysis like this, however, may be a burden to application developers but can be done through automatic compiler analysis as the compiler has all the related information. Such compiler optimizations are left as our future work.

In addition to software redundancy, we further evaluated whether adding hardware support like ECC/parity bits to on-chip SRAM and off-chip DRAM is justifiable. Our results show that protecting those memories in hardware provides only limited performance benefit and thus is not justified for our software redundancy approaches. One of the reasons is that not all the applications are able to take advantage of the hardware support if they have small input data. Another reason is that even with hardware support, we cannot eliminate all of the memory transfer overhead, the GPU-CPU transfer is still required and takes a significant portion of the overall data transfer time.

Acknowledgements

This research is supported by an NSF CAREER award CCF- 0747062 and AMD grant and equipment donations.

References

[1] AMD Stream Computing, http://ati.amd.com/technology/streamcomputing/index.html

[2] H. Gao, M. Dimitrov, J. Kong, and H. Zhou, "Experiencing Various Massively Parallel Architectures and Programming Models for Data-Intensive Applications", *Workshop on Computer Architecture Education (WCAE-08), in conjunction with ISCA-35,* 2008.

[3] Nvidia Developing with CUDA, http://www.nvidia.com/object/cuda_develop.html

[4] Qimonda GDDR5 – White Paper, http://www.qimonda-news.com/download/Qimonda_GDDR5_whitepaper.pdf

[5] Samsung Electronics: 256Mbit GDDR3 SDRAM: Revision 1.8, April 2005

[6] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. "Compiler-Directed Instruction Duplication for Soft Error Detection". DATE, 2005Qimonda GDDR5 – White Paper, http://www.qimonda-news.com/download/Qimonda_GDDR5_whitepaper.pdf.

[7] A. Munshi, "OpenCL: Parallel computing o the GPU and CPU", tutorial, SIGGRAPH, 2008.

[8] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicating instruction in super-scalar processors", IEEE Trans. on Reliability, 2002.

[9] M. Qureshi, O. Mutlu, and Y. Patt, "Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors", DSN, 2005

[10] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "A source-to-source compiler for generating dependable software", IEEE International Workshop on Source Code Analysis and Manipulation, 2001.

[11] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading", ISCA 2000.

[12] A. Reis, et. al., "SWIFT: Software implemented fault tolerance", CGO 2005.

[13] J. Sheaffer, D. Luebke, and K. Skadron, "The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware", In Proceedings of Graphics Hardware 2006.

[14] J. Sheaffer, D. Luebke, and K. Skadron, "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors", In Proceedings of Graphics Hardware 2007.

[15] J. Srinivasan, S. Adve, P. Bose and J. Rivers, "The Impact of Technology Scaling on Lifetime Reliability", DSN 2007.

[16] N. Wang and S. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors", in DSN, 2005.

[17] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor", ISCA-31, 2004