Exploring Scalability of FIR Filter Realizations on Graphics Processing Units

Jeff Rebacz, Erdal Oruklu, and Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, Illinois 60616-3793
Email: jrebacz@iit.edu, erdal@ece.iit.edu, sansonic@ece.iit.edu

***Abstract*—General-Purpose Computing on Graphics Processing Units (GPGPU) has lately been of great interest due to the release of architectures and software that simplifies programming graphics cards. This study explores how performance scales with FIR digital filters by varying the number of taps and the samples. We also discuss the trade-offs with various techniques for GPGPU programming in CUDA.**

I. Introduction

GPGPU computing has recently become exciting because graphics cards, which have possessed immense parallel processing power, are now easily programmable with frameworks such as Compute Unified Device Architecture (CUDA) and OpenCL[1]. CUDA provides a C-extended language to program and call kernels, which are the programs to be run on the GPU. Before CUDA, a programmer would have to represent and operate on the graphic card's data elements as if they were graphics elements (e.g. vertex coordinates, texture coordinates, pixels). Clearly, this graphics API did not provide the proper abstraction to a general purpose programmer. Despite this hurdle, researchers in [1] have ported an FIR filter to a Nvidia Geforce 6600 graphics card (CUDA became available in the 8000 series). They achieved twice the speed of an SSE-optimized CPU implementation, but only for a very large number of taps (60,000). Currently, the CUDA compiler allows the programmer to work at a comfortably high level of abstraction. Yet, to optimize kernel execution, knowledge of the underlying hardware must be known and exploited, as will be discusses later.

Another experiment on FIR filters was performed in [2]. In this study, GPGPU showed a 2-40x speedup in comparison to the CPU. However, the type of filter experimented with was very simple (16-taps) and the gpu execution time seems to exclude the memory transfers to and from the GPU. In our paper, we arrive at two general algorithms for handling large and small numbers of taps. Additionally, we account for memory latency, which consumes significant overhead for small data sets.

An explosion of GPGPU research following the released of CUDA show impressive speedups over cpu execution for many diverse applications. In [3], a classic computational geometry problem, approximating a Voronoi diagram, was sped up by a factor of 65. Real time imaging for PET scans is made possible with CUDA and high-ended graphics cards [4]. GPGPU applications are not only limited to graphics related problems. String matching can be improved by 24x [5]. MD4-RC4 can gain a speedup of 3-5x with CUDA [6]. As is clear, GPGPU can be applied to many problems. However, the performance benefit is largely dependent on how parallel the problem is. For example, string matching is most effective with large texts and small pattern sizes. Performance secondly hinges on the skill of the programmer. Hopefully, this paper can provide insight to those interested in GPGPU programming.

[1]OpenCL is another framework for programming graphics cards and additionally other parallel processing units (heteregeneous computing).

II. FIR Filters

Finite Impulse Response (FIR) filters are named so because after an impulse response/Kroenecker delta (a value of one for one sample but zero elsewhere), the filter's output will become zero in a finite number of iterations. An FIR filter performs a convolution of a series of coefficients with an input signal, as expressed in Equation (1), where y is output, n is the sample iteration, N is the number of coefficients or taps, c is the coefficient series, and x is the input series.

$$y(n) = \sum_{i=0}^{N} c_i * x[n-i] \tag{1}$$

In this experiment, two variables will be parameterized: the number of taps and the sample size. As any of these variables grow, more memory accesses are required. To implement the a 1024-tap FIR filter in C++ code, the following loop is written (with variables corresponding to Equation (1)):

```
for(n=1023;n<N;n++)
  for(i=0;i<1024;i++)
    y[n]+= x[n-i]*c[i];
```

A. FIR Applications with a GPU

Correlators or pattern matchers do the exact same operations as FIR filters (the sequence of coefficients is the pattern). With correlation, more sample points and longer patterns help increase the output energy when matches are found, positively affecting the overall signal-to-noise ratio. In any sort of detection like radar or ultrasound, making use of more

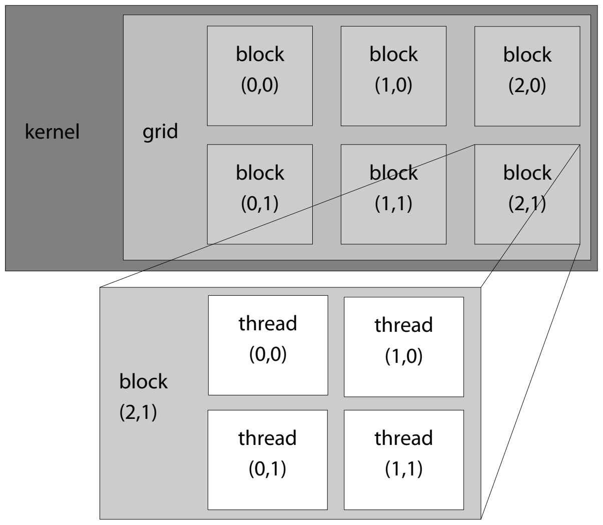

Fig. 1. This shows the kernel organization. Since all threads of a block get issued the same instructions, indexing is used to differentiate the threads. In the Figure, 2D indexing is illustrated, but for the FIR filters in this work, 1D indexing is used. The grid and block dimensions are small for illustrative purposes.

data and computational power gives better results. So, it is not unreasonable to experiment with 8K-tap FIR filters.

Audio processing often requires digital filters to operate on a lot of data and is often done on a desktop where a graphics card is available. Although there are no real-time requirements, the digital audio editor would certainly appreciate fast processing. An 8K-tap filter on our experimental workstation requires 9.14 seconds to work through 1M data points with the CPU; our modest GPU can get the same results in 95 ms, almost a 100x speedup! The only caveat is that most audio filtering is done with IIR filters, not FIR filters.

III. Programming in CUDA

This section is derived from the CUDA programming manual [7]. Programming in CUDA requires the understanding of several models used to abstract GPU hardware. The central idea to these models is that the same kernel is executed on many thread processors simultaneously. Threads are grouped together in blocks, and blocks are grouped together in a grid (see Figure 1). This organization is important to understand because it defines memory and execution behavior. All threads in a block run concurrently, executing the same instructions (unless they diverge after a control statement). Within the block, threads can access shared memory between themselves. While programming in CUDA, it is crucial for performance to keep memory accesses within shared memory (4 cycle read/write) because access to global memory (400-600 cycles) is done per-grid and is not fast. Additionally, threads contain registers, which the compiler will use as much as possible. If more space is needed, local memory must necessarily be used, which is not desirable since it is allocated from global memory.

The global memory access pattern is another factor that determines the degree to which memory accesses will tax performance. If threads access memory consecutively (thread x accesses base address+x), the memory accesses will be coalesced. Non-coalesced accesses can suffer an order of magnitude lower throughput [7].

A block of threads is issued to a Singe Instruction, Multiple Data (SIMD) multiprocessor in the GPU. There can be several multiprocessors. The number of blocks a multiprocessor can handle depends on the resources the block requires. Thus, avoiding fragmentation is desirable. Whenever possible, the choice of grid and block size should utilize all computing resources. Additionally, multiple blocks running per multiprocessor is useful so that an idle block (during a global read for example) can be swapped out for block that is ready to run. The block size, which translates to the number of threads in the block, should be chosen so that each thread has a sufficient number of registers. The larger the number of threads per block, the lesser the number of registers available per thread. All in all, CUDA programming involves great consideration of the various trade-offs involved with memory, computing resources and parallelism.

The rough calling procedure for CUDA is initialization, GPU memory allocation, transfer from CPU memory to GPU memory, CUDA kernel execution, transfer of results back to CPU memory, deallocation and shutdown. In our experiments, we time the inner three steps.

IV. Implementation of FIR Filters

The fastest FIR implementation has consistently been the one following the recommended strategies by Nvidia in [8]. The basic programming pattern for the FIR filter is:

1) Calculate the points that the thread will find using the thread and block indexes
2) Load the sample points and coefficients from global memory into shared memory
3) Synchronize with other threads so that all threads in the block can safely read the shared memory
4) Convolve the coefficients with the sample points in shared memory
5) Record the result.

```
__global__ static void fir1024(float* in,
float* coef,float* out, int n)
{
//Step 1
  int i=1024 * blockIdx.x +threadIdx.x;
  int idx=threadIdx.x;
  int j;

//Step 2
  __shared__ float sh[2048];
  __shared__ float c[1024];
  sh[idx]=in[i];
  sh[idx+512]=in[i+512];
```

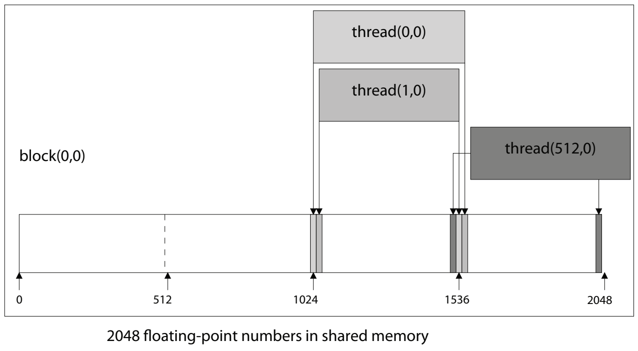

Fig. 2. To calculate 1024 points, a block will load 2048 data points into shared memory. After the convolutions, each thread will store two results into global memory (relative location shown in figure).

```
  sh[idx+1024]=in[i+1024];
  sh[idx+1536]=in[i+1536];

  c[idx]=coef[idx];
  c[idx+512]=coef[idx+512];

//Step 3
  __syncthreads();

//Step 4
  float acc=0.f;
  float acc2=0.f;
  #pragma unroll 32
  for(j=0;j<1024;j+=1){
    acc+=c[j]*sh[idx-j+1024];
    acc2+=c[j]*sh[idx-j+1536];
  }

//Step 5
    out[i+1024]=acc;
    out[i+1536]=acc2;
}
}
```

As shown in Figure 2 The 1024-tap filter reduces global memory accesses by operating on a chunk 2048 samples per block, which is only read once per block. Instead of accessing global memory 1024*1024 times to calculate 1024 points, the block reads global memory 2048 times and reads shared memory 1024*1024 times. Since a block can contain only 512 threads, each thread loads four locations and calculates two points to load 2048 points and calculate 1024 points. For example, Block (0,0) will load the data points [0,2047] and calculate [1024,2047]. Block (1,0) will load the data points [1024, 3071], and calculate [2048,3071] and so on.

One ostensible inefficiency with this implementation is that the chunks overlap 50% with its neighbors. Methods removing the overlapping global memory reads have not performed faster. One method used a circular buffer that replaced the old 1024 chunk with a new chunk after each iteration of a loop inside the kernel. Another method partially calculated points outside the higher 1024 within the chunk (this increased global memory writes and also required a degree of serialization between blocks). From these observations, we recommend avoiding a complex memory pattern in favor of keeping the kernel code smaller.

The limitation on shared memory complicates kernels implementing very large taps since both the sample points and coefficients must be loaded into shared memory in chunks. We find that instead of having a double loop that loads new chunks within the kernel, it is about 5% faster to divide the sample points and coefficients in chunks and work within that context. For example, an 8192-tap filter is best implemented using eight slightly modified 1024-tap filters. These 1024-tap filters load 2048 datapoints and 1024 coefficients. The 1024-tap kernel is called eight times with a different offset parameter specifying which chunk to operate on. With eight 1024-tap kernels working on an 8192-tap FIR, each kernel will vie with seven others to write to the same global memory location if they are all issued together. Synchronizing the threads after each kernel call is necessary to avoid memory this conflict. One iteration will issue a kernel to partially calculate all the points in the data set.

A. Loading Coefficients

For FIR filters, we can exploit constant memory for our coefficients because throughout execution, the coefficients will not change. Constant memory actually resides in global memory, but is cached. There are 64KB of cache available, and cache hits can be as fast as a register read if all the executing threads are reading the same address. This is the case in our kernel's algorithm. For each thread, the first coefficient used is always the first in the series.

Shared memory is accessible per block, whereas constant memory is accessible per grid. However, bank conflicts, which occur when threads access the same bank of memory, serialize memory accesses and reduce throughput. In a 128-tap FIR filter processing 2^{22} points, retrieving constants from constant memory will yield a kernel execution of 13.64 ms, whereas retrieving them from shared memory will take the kernel 12.58 ms. Thus, we find that for most cases, directly accessing constant memory for coefficients is not optimal, but does not compromise performance much. On the other hand, loading the coefficients from constant memory, storing them in shared memory, and then accessing them through shared memory is slightly faster than loading the coefficients from global memory (no caching).

B. Referencing Global Memory

Referencing from global memory can easily cripple performance. In implementation A, shared memory is used to hold and retrieve the data points and coefficients. In implementation B, the coefficients were instead retrieved directly global memory. Implementation A will run an 8K-tap filter on 1M points in 95 ms whereas implementation B will take 3108 ms. This clearly illustrates the importance of avoiding global memory accesses. It also highlights the limitations of GPU. A GPU has lots of processing power, but the operands must be close to the GPU arithmetic logic units in order to fully utilize that power. Shared memory and registers are crucial to achieving optimality, but they are scarce and need to be allocated carefully.

V. Testing Platform

The FIR filters ran on a Dell Dimension DXP061 computer with an Intel Core 2 6400 running at 2.13 GHz with 2 GB of DDR2 RAM running at 1066 MHz. The processor contains a 2 MB L2 cache. The graphics card used is the Nvidia GeForce 8800 GTS 512, which features 16 multiprocessors (each having 8 stream processors) 64KB of constant memory, 16KB of shared memory per block, 8192 registers per block. The clock rate of the GPU is 1.67 GHz, and 512 MB of global memory is available. On the software side, Windows XP is used with version 190.38 of the Nvidia developer drivers, version 2.3 of CUDA toolkit and SDK. The development environment was Microsoft Visual C++ 2005 Express Edition. The compiler optimizer's were set for speed. Additionally, the Streaming SIMD Extensions 2 enhanced instruction set was enabled, and a fast (as opposed to a strict or precise) floating-point model was used (to be fair since the GPU follows IEEE floating-point standard with a few exceptions [8]).

VI. Results

Two generic kernels based off the discussion above were primarily used to generate the results. One kernel worked for FIR filters with between 4 and 1024 taps. The second kernel worked for 2048-tap and larger. For a few of the configurations with a low number of taps and small data size, we have found that tap-specific kernels can slightly exceed the performance of our generic kernels.

Although not clearly visible in Figure 3 the points in the dark gray region on the bottom-right of the surface mesh have speedups less than 1. Here, the overhead of transferring data back and forth between the GPU and host memories is too costly in comparison to cpu execution time. Thus, GPGPU execution prevails for large data sets.

Another interesting note is that speedup does not appreciably increase along 1 axis. Figure 3 and Figure 4 show that when data size and taps increase together, a sharp rise in speedup occurs. The performance increase seems to cap at a speedup of 97.

CUDA Visual Profile is a convenient tool to profile kernels. One of the metrics it calculates is occupancy, which is the percentage of resources used during kernel execution. High occupancy is achieved when register usage is low. The 2^{17}-tap filter operating on 2^{22} points yields a 66.7% occupancy, 8 registers per thread, 248 out of 293880 branches were divergent (we synchronize almost immediately after diverging), and no uncoalesced memory acceses. A 66.7% occupancy is very good, and acheiving more is very difficult. The kernels presented in this paper achieve the same occupancy for all taps and data sizes.

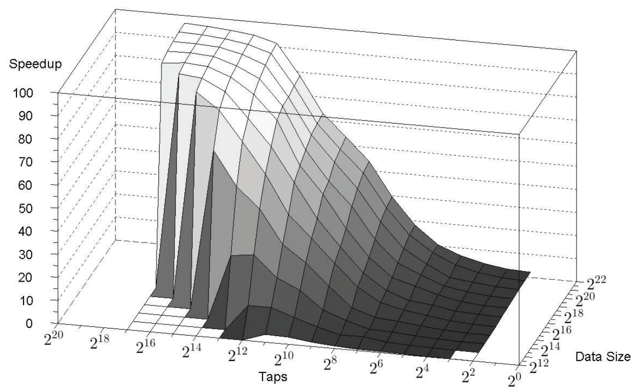

Fig. 3. This figure shows that as the data size and number of taps grow, the speedup of a gpu over cpu increase.

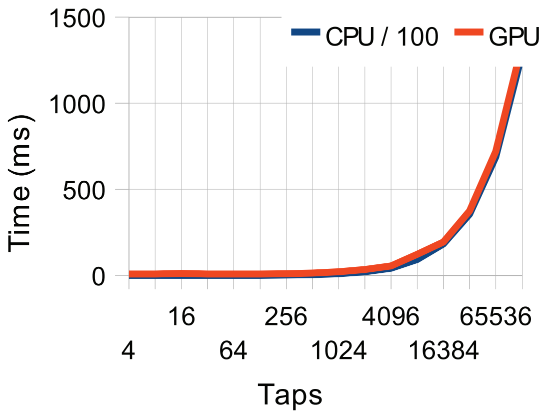

Fig. 4. If the CPU execution time is divided by 100 and these times are plotted against GPU times, the same growth curves are observed. These times were generated for 1M data points.

VII. Conclusion

By understanding the programming model, memory hierarchy and hardware of the graphics card, a high-performance GPGPU kernel can be developed. The proper use of GPU memory is most important for speed. Our best implementation uses shared memory to store 2048 data points and 1024 coefficients. When more coefficients are required, the kernels are serially executed with different offsets. This scheme can nearly achieve a 100 times speedup over the CPU for very large data sets.

References

[1] A. Smirnov and T.-C. Chiueh, "An implementation of a FIR filter on a GPU," ECSL, Tech. Rep., 2005.

[2] W. S. Sajid Anwar, "Digital signal processing filtering with GPU," in *The Institute of Electronics Engineers of Korea*, July. 2007.

[3] I. Majdandzic, C. Trefftz, and G. Wolffe, "Computation of voronoi diagrams using a graphics processing unit," in *IEEE International Conference on Electro/Information Technology (EIT)*, May 2008, pp. 437–441.

[4] S. S. Huh, L. Han, W. L. Rogers, and N. H. Clinthorne, “Real time image reconstruction using GPUs for a surgical PET imaging probe system,” in *IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*, Nov 2009, pp. 4148–4153.

[5] C. Kouzinopoulos and K. Margaritis, “String matching on a multicore GPU using CUDA,” in *13th Panhellenic Conference on Informatics*, Sept. 2009, pp. 14–18.

[6] C. Li, H. Wu, S. Chen, X. Li, and D. Guo, “Efficient implementation for MD5-RC4 encryption using GPU with CUDA,” in *3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication (ASID)*, Aug. 2009, pp. 167–170.

[7] NVIDIA, “Compute Unified Device Architecture (CUDA) Programming Guide,” 2008. [Online]. Available: http://www.nvidia.com

[8] ——, “NVIDIA CUDA C Programming Best Practices Guide,” 2009. [Online]. Available: http://www.nvidia.com