LA-UR- 07-00371

Approved for public release; distribution is unlimited.

Title:	Matched Filter Computation on FPGA, Cell and GPU
Author(s):	Zachary K. Baker Maya B. Gokhale Justin L. Tripp
Intended for:	IEEE Symposium on Field-Programmable Custom Computing Machines

Matched Filter Computation on FPGA, Cell and GPU

Zachary K. Baker, Maya B. Gokhale, Justin L. Tripp
Los Alamos National Laboratory
Los Alamos, NM 87545
Email: {zbaker,maya,jtripp}@lanl.gov

Abstract

The matched filter is an important kernel in the processing of hyperspectral data. The filter enables researchers to sift useful data from instruments that span large frequency bands.

In this work, we evaluate the performance of a matched filter algorithm implementation on accelerated co-processor (XD1000), the IBM Cell microprocessor, and the NVIDIA GeForce 6900 GTX GPU graphics card.

We provide extensive discussion of the challenges and opportunities afforded by each platform. In particular, we explore the problems of partitioning the filter most efficiently between the host CPU and the co-processor.

Using our results, we derive several performance metrics that provide the optimal solution for a variety of application situations.

1. Introduction

Even though clock speeds of current microprocessor-based systems are 10-30 times that of a typical FPGA based design, the large amount of spatial parallelism afforded by modern FPGAs offers the potential for speedup. FPGAs now deliver peak floating-point performance equaling or surpassing that offered by microprocessor-based systems [10], a trend that will likely intensify in the future as FPGA logic area continues to grow.

Other architectures, including the IBM Cell processor and Graphics Processing Units (GPUs) can act as accelerators for general purpose computing. The devices were initialy designed for use in gaming. The Cell is a part of the Sony Playstation 3, and GPUs, such as the NVIDIA GeForce series that target, are used for accelerating the rendering performance on commodity CPU-based systems.

On all platforms, the devices are meant for a large, consumer audience. Cell and GPUs are targeted at the gaming market, and FPGAs are widely used in telecommunications and even in consumer products. This enables the prices to be relatively low compared to the cost of highly customized chips, and make them very interesting and accessible to the research community.

The ability to stream data to/from memory and overlap control and data flow, combine to help these coprocessor-based systems attain high levels of performance. Capitalizing on the potential of these coprocessors, supercomputing vendors such as SRC Computers, IBM, Cray and SGI are now offering high-performance computing systems coupling standard CPUs with FPGA, Cell, and other co-processors. These system architectures make it feasible to adapt existing scientific codes to use FPGA accelerators.

The matched filter is a vital kernel for extracting useful data from hyperspectral instruments. The term *Hyperspectral* implies that the spectra collected covers a very large swath of the frequency spectrum. Generally, this data is broken into many frequency bands that are independently collected. Generally this is done through the use of a prism (Figure 1) for near-visible bands or a set of filter banks for wider bands. In our application, floating point data is collected for each band over a few hundred bands. The main unit of data in hyperspectral processing is the *datacube*, layers of 2-D image for each spectral band of interest. The *signature* is simply a vector of single-precision floating point values for every spectral band, in our case, 240.

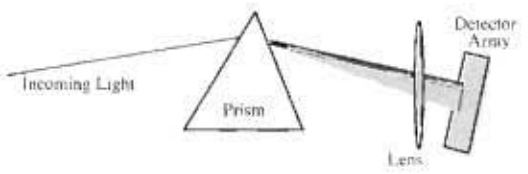

Figure 1. Splitting spectral bands for indepenent detection

The matched filter takes the hyperspectral data and matches it against a particular signature. The signature is a vector of coefficients that represent the spectral reflection or transmission of a particular material. The matched filter is most commonly used in detecting chemical plumes. An

interesting plumes might signify pollution, illegal arms production, contraband drug manufacturing, etc. It can also be used to detect particular types of vegetation. For instance, the spectral signature of the invasive species *athel tamarisk* is unique [4]. When this signature is applied against a datacube, the tamarisk is highlighted and is distinct against the background. Because tamarisk is such an agressive species out of its native habitat, it must be destroyed if it nears fragile ecosystems. The matched filter is useful for detecting it remotely and monitoring its spread.

The version of the matched filter we have implemented on various platforms is focused on long-term surveilance with a large number of signatures. In terms of kernel design, this means that the signatures are pre-processed – which reduces overhead and setup costs – but that there will be huge volume of data that has to be processed as quickly as possible.

The various implementations we have investigated are customized for each platform. In all systems we have the ability to partition the design between a CPU and the co-processor. Because the strengths of each platform vary, we have attempted to exploit the strengths of each in our designs. In the following sections, we will attempt to explain and justify out design decisions, while simulateously exploring the strengths and weaknesses of each platform.

1.1 Matched Filter Algorithm

We implement a reduced set of operations for the match filter. Using pre-processed signature templates, the operations required are simplified. The kernel remains a realistic application as the signature templates can be generated offline with no loss of practicality. The simplification focuses attention on the real-time costs of data transfer and bulk computation.

Figure 2 illustrates the basic blocks of the kernel. First, the signatures and the datacube are read in from the input file. The datacube is then transposed, as the data arrives from the sensor in spectral frames, meaning that the entire 2-D frame for a spectral band is contiguous. After one frame is taken, the filter bank switches to the next spectral band, and another frame is saved. Because the matched filter loops over all of the spectral bands for a single pixel during the main filter phase, it is more efficient to have all of the spectral data for each pixel in sequential order. Otherwise, the memory stride would be the size of an entire frame, or 240,000 32 bit words. This is not efficient, and thus the data is transformed early on in the process.

The main pixel loop is performed for every pixel in the image. In this phase, each value in the datacube will be accessed several times. Here, the ability to reuse data and parallize computation is highly valuable. First, the values of an entire column are averaged. Second, that average is subtracted from the entire column. Third, the dot-product is executed over the entire pixel column and the signature. Various scalar operations are executed on the result of the dot-product, and the results are returned. The process is repeated over all pixels in the frame.

2 Related Work

Zhuo et al. [11] explored a variety of linear algebra operations, including dot-product. As dot-prodct is the main kernel we implement on the FPGA, this is an important prior work. The authors' strategy for dealing with the floating-point latency is to execute the dot product accumulation as a tree-based reduction circuit. This of course requires more routing and more adders than the systolic array approach that is enabled by our application. The matched filter application allows us to be less concerned with the latency of any particular dot-product result and more focused on the overall throughput.

Meredith et al. [7] have implemented a variation of the matched filter application on GPU co-processors. The authors report a 26.5x speedup over a Pentium 4 CPU. This performance improvement is higher than what we claim, but their application is somewhat better for the GPU as it is more computationally intensive. Our version of the application only processes the pre-prepared signature templates and data cube, rather than processing the signature library. As processing the signature library requires the calculation of a covariance matrix, and thereby a matrix inversion, it is of high computational complexity. Because we expect to process the signatures once, and then apply them against datacubes over long time spans, it is sensible for our application to only support pre-processed signature databases.

One interesting approach to matched filter design was explored by Love et al. [?]. In this approach, a Digital Micro-Mirror Array is used to select bands of interest. Micromirrors, commonly found in video projectors, provide a tiny moveable mirror for each pixel of an image. These mirrors are independently controlled at high refresh rates. In the DMMA approach, a single line of an image is directed through a prism. The prism defracts the light into its various components, which are then directed onto the micromirror device. By selectively turning on and off rows of pixels in the DMMA, frequency bands can be selected. The bands that of are interest reflect off of the DMMA and are condensed. This, in essense, is the matched filter output, except it is accomplished optically.

3 FPGA Implmentation

The implementation of the matched filter on the FPGA has been carefully partitioned between the CPU and the

Figure 2. Main processing blocks of the matched filter

FPGA in order to make best use of the hardware. Modern FPGAs allow a designer to configure any arbitrary logic function. Based on the design, these operations can function at high clock rates and behave similarly to a CPU-based version. Because the FPGA does not have to be a "general-purpose" device, a designer can take advantage of the application-specific attributes of the device to achieve higher performance than a CPU. This can be true even with a CPU that operates at perhaps 10x the clock frequency of the FPGA. However, this is only possible through hardware parallelism, and careful design.

Because an FPGA – for any given configuration – is responsible for only one function, the entire resources of the device can be leveraged for that function. In the case of the matched filter, it is possible to implement dozens of the memory and hardware resources to build a single-precision dot product unit. This is because the resource requirements for the single-precision multiply and accumulate allow for significant parallelism without much control overhead. As we will discuss, the dot-product portion can be parallelized in multiple ways, allowing the data and computationally intensive portion of the kernel to be executed very quickly on the FPGA.

While the dot-product is straight-forward to implement in hardware, the other aspects of the matched filter are not appropriate for hardware. For instance, after calculating the dot-product, various scalar multiplies are performed on the dot-product result. While it would be possible to implement this functionality in hardware, these units would be both expensive and under-utilized. One of the scalar operations is a floating-point square root. This kernel would occupies several times more area on the FPGA, but would only be in operation at the end of every dot product. This means the duty cycle is roughly one result every 240 cycles. This is not the most effective way to use the scarce FPGA resources, thus pushing the occasionally-used, but expensive scalar operations to the CPU.

3.1 Architecture

In the first stage of operation, the signature values are loaded onto the XD-1's SRAM memory banks. These will be moved in and out of the FPGA as required. The FPGA's internal block memory resources are then loaded from SRAM memory. The SRAM words are 64-bit, allowing two 32-bit floating point values to be loaded into the block RAM in each cycle.

The computational stage of the dot-product is a streaming of image data through the FPGA. The dot-product units are arranged in a systolic array(originally developed in [6]). The system architecture is depicted in Figure 3. The power of the systolic array lies in its ability to reduce total device interconnect. By allowing only short connections within a small, repeated unit, a design will achieve a higher operating frequency as well as lower area consumption. Each unit is only connected to the previous and following units, and each controls its own data movement and computation. This eliminates the need for large, multiplexed data and address buses. The systolic array prevents the proliferation of global routing resources. The clock is the only element absolutely necessary and thus we desire to limit global routes to it.

We implement the dot-product systolic array using the Sandia floating point cores [5].

Through the use of the systolic array we allow for increased frequency performance, decreased interconnect, and simple, easily scalable units. We implement the parallelizable and computation intensive operations within the systolic array and implement the serial and control intensive operations within a microcontroller.

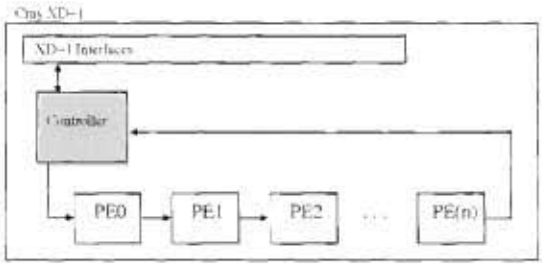

Figure 3. Illustration of the general dot-product systolic array in the XD-1 system

3.2 Extracting Parallelism

The design of the one-signature-per-unit systolic architecture is meant to provide parallelism over signatures. A struggle with implementing systems with floating point of FPGA has always been the latency associated with floating point cores. In order to provide a fast clock frequency, it is

necessary to pipeline the units, and this means that any operation that requires feedback, as in a counter or a series of additions, cannot accept new inputs every cycle. This is addressed in [11] through the use of multiple adder elements and elaborate scheduling.

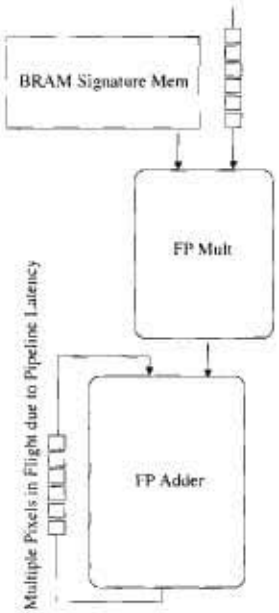

Figure 4. Due to the latency of the floating point adder, multiple dot-products are interleaved within the adder pipeline

Our approach grows from vast amount of dot-product operations required in the matched filter. For every datacube, there is a dot-product executed for every pixel across every signature template. For our datacube, the dot product is executed on a 240 element spectrum vector, over thousands of signature templates. Because we are interested in throughput of entire datacubes rather than a fast result on a single operation, we can interleave computations to mitigate the latency. Using just one adder of k cycles latency, and interleaving k separate dot-product calculations, the dot-products for k pixels can be computed simultaneously without incurring any additional resource costs. Thus, the parallelism is both signature-wise across the systolic array and pixel-wise within a single unit.

3.3 FPGA Results

Using the Cray XD-1 with a Virtex 2-Pro 50 -7, we placed and routed the design with 20 dot product units to 130 MHz. This rate is less due to the design of the dot product units and more to the SRAM interfaces and arrangement of the pin constraints for the FPGA board. Each unit requires 2 block RAM elements and roughly 4000 logic cells.

The partitioning choices have a big impact on the performance of the design, and Amdahl's law becomes a limiting factor. Only the dot-product is moved to the FPGA, and the dot-product accounts for somewhat more than 50% of the running time per signature. Thus, the maximum speedup is 2x. Calculating the per-signature speedup is complicated by the systolic array. Because the system has 20 dot-product units, computing the result for one signature requires essentially the same time as calculating twenty. At 130 MHz, a 240x1000 frame with 240 spectral components requires, on average, 12.4 seconds to transfer the datacube, perform the computation, and return the results to the CPU. This is equal to 0.62 seconds per signature for only the dot-product. Integrating the FPGA-based dot-product with the rest of the matched filer application, the per-signature time is roughly 3 seconds.

While the architecture could easily be implemented in a custom ASIC – in fact, the simple units that make up the systolic array are designed explicitly for ease of ASIC implementation – the use of FPGA allows the user to utilize parameterized designs which allow for variable numbers of spectral bands as well as optimized memory sizes for a particular problem. As well, FPGAs allow the design to be scaled upward easily as process technology allows for ever-larger gate counts.

4 Matched Filter Implementation on the Cell

4.1 Cell Broadband Engine

The Cell Broadband Engine (CBE) is a new microprocessor architecture designed that extends the 64-bit PowerPC Architecture. The Cell is the result of collaboration between Sony, Toshiba and IBM. Despite the fact that the processor was primarily designed for game consoles and media centered electronic devices, it was designed to overcome fundamental problems in microprocessor design. To this end, the Cell is being used in other application areas, such as supercomputing [9].

The Cell consists of a single-chip multiprocessor with nine separate processors which share a single coherent memory. The cell follows the current microprocessor trend of added multiple cores to a single chip, but instead of all of the cores being the same, the Cell has one PowerPC Processor Element (PPE) and eight Synergistic Processing elements (SPE).

The PPE is a general purpose 64-bit PowerPC architecture processor capable of running 32-bit and 64-bit operating systems and applications. It supports two simultaneous threads of execution, which appear to the software as two independent processing units. The PPE has a typical virtual memory subsystem with a 32 KByte L1 instruction cache, a 32 KByte L1 data cache and a unified 512 KByte L2 cache.

The caches and memory subsystem allow the PPE to see all of the shared memory as a flat memory space. The PPE also supports integer, floating point and a SIMD/vector unit.

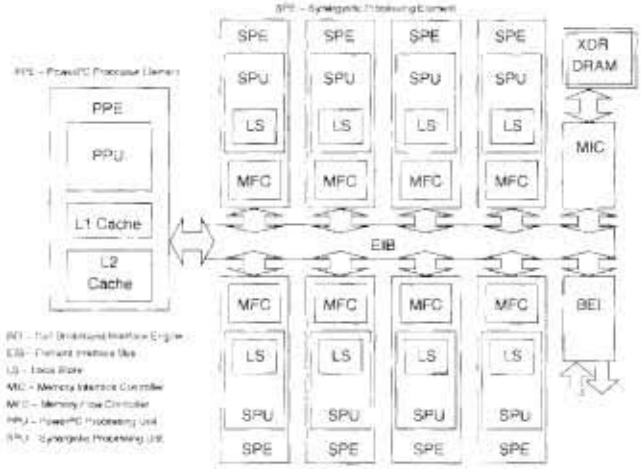

Figure 5. Cell Broadband Engine Overview

The SPE is optimized for running compute-intensive functions and is not optimized for larger applications or operating systems. The SPE is an independent processor that runs its own program and has full access to the shared I/O and memory. The SPE is called synergistic due to is reliance on the PPE to run the operating system and to provide the high-level application control. In turn, the PPE relies on the SPE to provide application performance.

The SPE achieves its compute intensity through a special SIMD instruction set. Although the SPE can be programmed in high-level languages and it has as special Vector/SIMD Multimedia instruction set extension for increasing the parallel computation on each SPE. Operations that do not take advantage of the available vectorization reduce overall performance. This is due, in part, to the 128 128-bit registers of the SPE. All operations use the wide registers regardless of data type. Vectorization allows the most efficient use of the registers.

The SPE does not have any cache, but has a 256 KByte software controlled local store (LS). The local store is for both instruction and data, and has no protection or translation for the access from its own SPE. Since there is no cache or memory translation, the SPE cannot directly access the main shared memory. It must transfer the data via DMA to its own local store using the Memory Flow Controller (MFC).

The MFC contains a DMA controller for transferring instructions and data to the SPU's local store. A DMA can be initiated by the SPU, by another SPU or by the PPE. It is the function of the MFC interface to the Element Interface Bus (EIB) and synchronize the data transfers with all the other processors in the system. The MFC operates asynchronously with respect to the SPU, so that it is possible to overlap DMA transfers with other concurrent operations.

The EIB provide coherent communication between the PPE processor, the SPE processors, main memory storage and I/O. The EIB is a four ring structure for data and a tree structure for commands. The internal bandwidth of the buss is 96 bytes per cycle and it is possible to have more than 100 outstanding DMA requests between main storage and the SPEs. Besides the SPEs and PPE, the EIB is connected to two other elements: the Memory Interface Controller (MIC) and the Cell Broadband Engine Interface (BEI). The MIC provides the interface between the EIB and main storage. The BEI provides the interface, control and translation for all external I/O communication.

4.2 Matched Filter

Three main areas impact the implementation of the Matched Filter on the Cell processor. First, the limited memory of the local store requires that the implementation use no more that 256 KBytes for a working set. This is further reduced by instruction storage since both need to be in the local store at the same time. Since the memory working set is much smaller than the total data (230 MBytes), then DMA transfers must be used to page through the data. Although the MFC provides a DMA controller, there are very specific limitations on the size and alignment of that data. Finally, there are two levels of parallelism that can be exploited to increase the compute intensity. First, multiple SPEs can operate in parallel. Second, vector operations can be used to further compute effectiveness.

The DMA controller supports naturally aligned transfer sizes of 1, 2, 4, or 8 bytes, and multiples of 16-bytes, with a maximum transfer size of 16 KBytes. Peak performance is obtained when the data is 128 byte aligned and the size of the transfer is an even multiple of 128 bytes. The image data cube of the matched filter is three dimensional with the dimensions 240 by 240 by 1000. In order to DMA a 240 float row to the SPE, the data must be rearranged so that the data necessary is adjacent. This inversion of the data is expensive, but allows the DMA to be padded to a transfer sixteen 256 float rows (16 KBytes). Otherwise, a single value must be taken from each column to form the data needed for the dot-product.

Since the SPE cannot directly access external memory and must explicitly transfer all data via DMA, the allocation of data in the LS is very important for effective performance. The best DMA performance is achieved when transferring the largest DMA size of 16 KBytes. Thus, the data input to the dot product will be sixteen rows of 256 floats. Each of the sixteen will rows are used in the dot product with the signature before the next DMA transfer. The signature data changes only after all the data (230 MBytes)

has been processed. This is transferred only once and transformed once before use in the dot product.

The parallelism is achieved by transferring a single signature to each of the SPEs and have each SPE look for a match using all of the data. Since the application is interested in processing thousands of signatures, this approach easily keeps all of the SPEs busy and can be made to efficiently process all of the data input. The individual SPEs calculate the dot product using vector add and multiplies, which allow four floats to be processed concurrently. Not all operations can be vectorized, but vectorization of key computations increases the parallelization at the operation level. The SPE also has two execution pipelines and can schedule certain operations to run in parallel on its own execution pipelines.

4.3 Results

Initial results for the Cell implementation took over 30 seconds to process the data. Rearranging the data for the DMAs to the SPEs took 26 seconds and the remaining 4 seconds were all that was needed to process from one to eight signatures. The required transpose is very expensive on the PPE of the Cell due to the fact that none of the reads can take advantage of caching. Since the arrangement of the data was determined to be arbitrary, preprocessing the data reduced the cost of reading the file to just 7 seconds.

5 GPU Implementation of Matched Filter

5.1 Overview of GPU Architecture

Graphics Processing Units (GPU) are a Commodity-Off-The-Shelf (COTS) product that is meant for accelerating the rendering of images to the screen. The intended audience of these products is largely the gaming market, where high frame rates and elaborate, complex 3-D graphics require state-of-the-art technology. Because of the demand from the gaming community, companies such as NVIDIA [2] and AMD/ATI [1] have provided processing capabilities that have outstripped the development of general-purpose microprocessors.

Part of the ability of GPU developers to innovate comes from the restrictions that come with their target applications. Graphics code tends to be image centric, with data access and result production occurring in a very predictable manner. In particular, codes tend toward easily vectorizable computation with limited data usage. Other common characteristics include [8]:

- Single-instruction, multiple data (SIMD) structure - the same code is executed for each pixel of an image. Because all of the processors are performing the same operations, there are fewer synchronization problems and less resources dedicated to instruction queues and branch prediction. This architectural structure does not easily support branching, as data movement needs to occur in lockstep through many parallel processors.

- Single pixel outputs - this is a particular restriction of GPU architectures before the NVIDIA GeForce 8800 [2], released in late 2006. This model assumes that all operations will produce a single result. This is a reasonable assumption for many graphics-centric applications, as well as image processing applications that do not necessarily result in a displayed image. Convolution, for instance, and template matching produce a single output.

  In this model, computations executed on the GPU behave much like single-return value functions. This can be inconvenient, as we will see later, as many computations have side effects and multiple results. In these situations, the single pixel output restriction causes us to repeat computation or build elaborate workarounds.

  The single pixel output restriction also means that scattering data is impossible. Because a function can only produce one output, and that output is ultimately the rendering output for a pixel, it is impossible to randomly place data into memory. While gathering data from arbitrary locations is supported, data outputs are restricted in the graphics paradigm. In that paradigm, pixels are rendered to a image frame, and scatter is rarely required.

- Data locality - this is less of a constant across all GPU applications and more of determination of performance for a given application. Considering simple kernels like convolution and template matching, the data locality is high, with each pixel output only considering the pixels in its immediate vicinity. While the internal architecture of GPUs is proprietary and closely held by NVIDIA and ATI, the importance of cache locality remains. Neighborhood operators continue to be a strength of GPU devices even as their capabilites become more general.

- Vectorizable Code - There are two paradigms for this approach within the GPU. Because of the SIMD nature of the intended applications, there is benefit in performing computation in a vector format. For instance, in a series of computations on successive pixels, the computational pipeline can remain full as there is a large volume of data and independent computation.

  The second opportunity for vector acceleration is by packing multiple values into the Red, Green, Blue, and Alpha (RGBA) components of a pixel. In some situations, this allows the computation to be spread across

parallel vector units. However, the performance of the packing seems to be a driver and implementation dependent. We have observed kernels where using only the Red channel is faster than packing the data across RGBA.

These restrictions and behavioral patterns have allowed the advancement of GPU architectures and performance at a much higher rate than general purpose microprocessors. Current GPU architectures actually appear much like a collection of classic vector supercomputers on a single device. Because of the restrictions on the form of the output, the problems with cache coherency and other memory issues that complicate other multi-processing architectures are avoided.

Before the details of the implementation are presented in Section 5.2, some GPU terminology should be defined. The *texture* is the basic storage structure on the GPU. It is used normally for providing storage for a texture, or a series of values that can be mapped to a surface to provide the appearance of a 3-D surface. This is used because it is computationally cheaper to map a tiled texture to a surface than it is to actually store the surface with detailed surface mapping. For general purpose use, matrices can be mapped to texture memory, and recalled randomly. Multiple textures can be used to calculate the output of a given function.

5.2 Implementation Details

We have implemented the main pixel loop of the matched filter on a NVIDIA GeForce 7900 GTX, with an Intel Pentium D 2.8 GHz with 2 GB RAM. The main pixel loop is not the whole of the computation in the application, but it does represent at least 85% of the execution time. Restricting the GPU's usage simplified development time as well as allowing us to focus our attention on the most time consuming part of the application.

We break the computation into four parts, in order to work within the confines of the GPU architecture.

The first operational block is the calculation of the mean for each row of the hyperspectral data cube, and the subsequent subtraction of the mean from each element of the row. The calculation for a single row is as follows:

```
for (i = 0; i < n; ++i) sum += b[i];
        mean = sum/n;

for (i = 0; i < n; ++i)
        b[i] -= mean;
```

The mean calculation reveals many of the oddities that make programming for the GPU not as straightforward as programming for a general purpose CPU.

Because only one result can be generated per pixel, a vector subtraction can not be executed in one step. Ideally, the summation of the vector could be computed, the division executed, and then that result subtracted from each of the components of the vector. However, because temporary results cannot be stored between individual pixel, two rendering passes are used.

The array b[0..n] is laid out as a row in a texture. A block of sums are calculated in one pass, with each b[0..n] row producing one element in a vector of sums. In this format, a single column of the output frame buffer is rendered. The coordinate of the particular row of each pixel to be rendered is used along with a loop variable to sum the entire row. The mean is calculated from the sum and stored as the output pixel Red channel.

The second rendering pass subtracts the previously computed mean from all elements of the texture. Conveniently, while the rendering kernel has changed, the textures remain in memory, allowing the new computational kernel to compute the new texture values from the data still on the GPU.

The computation of the f_images and mf_images results are the most complicated kernels. They require both the dot-product of the newly mean-subtracted b[] with another matrix, a[], and a dot-product of b[] with itself. The difference f_images and mf_images is a collection of scalar operations.

The interesting and problematic aspect of these final results is that they both share the results of the two dot-products, which are the most data intensive computations of the entire pixel loop. However, because of the single-result restriction of the GPU, the products must be calculated twice. This is highly wasteful. Feasibly, the dot products could be calculated first and then stored for use in calculating the final results independently. It is currently unclear if the overhead cost of the two additional rendering passes would outweigh the cost of recomputing the dot-products.

5.3 Results

We have implemented the main pixel loop of the matched filter on a NVIDIA GeForce 7900 GTX, in an Intel Pentium D 2.8 GHz with 2 GB RAM. For the input_5.bin dataset, which contains a single datacube and 5 signatures, the CPU-only time is 35 seconds. Of this time, about 6 seconds is required for pre-processing, and then 6 seconds per signature. The preprocessing stage includes reading and transposing the input data cube.

The GeForce 7900 GTX implementation for the same dataset requires only 12 seconds, almost a 3x improvement. Because only the pixel loop has been modified, the 6 second pre-processing time remains constant. In the pixel loop, however, each signature now only requires about one second. For large signature databases, this means that the sig-

nature throughput will approach a 6x speedup over the microprocessor implementation.

5.4 Remaining Work

5.4.1 Floating Point Inconsistencies in Results

There are some interesting inconsistencies in the results that we are still exploring. The GPU results largely match the CPU results to several decimal places. However, in some cases the results are up to 0.9 in error. This sort of error is rare (perhaps one in 10,000 results) but they do occur. We have executed the code on various platforms and thus various implementations of floating point, and no errors have been observed that approach the differences between a CPU and GPU implementation.

There was some suspicion that the limited use of double-precision operands in the microprocessor implementation could have caused some of the differences. The GeForce 7900 only supports single precision floating point, but reducing the few double-precision operands to single-precision did not move the CPU results toward the GPU results.

In the following table, any result with an error larger than 0.1 is recorded for the first row and the first few signatures:

Sig	Col	GPU	CPU
0	57	5.507	5.650
0	396	6.591	6.703
1	178	6.032	6.146
2	179	-4.846	-5.540
2	673	1.738	1.617
2	678	1.689	1.584

It should be clear that there is no particular pattern – the errors do not occur on the same columns, or in the same number for each signature. This is an issue that needs to be resolved.

5.4.2 Additional Improvements to the Approach

We are interested in exploring various improvements to the GPU design. The time to implement the improvements is one constraint that has limited our ability to explore the design space. It will also be useful to more carefully evaluate the likely effectiveness of some of the more complex changes before they are actually implemented.

The following list orders potential changes from the highest and most likely to be profitable to the least.

- Block signature execution - currently, a single column is rendered. During the operation of the matched filter, a hyper-spectral datacube is reduced to a single 2-D frame for each signature. Rendering a single column is an effective way of generating results without resorting to large-strided cache-unfriendly 3-D matrix representations. If a complete result frame for a single signature is produced, there is no alternative.

  However, we have noticed that there is some opportunity for data reuse as well as an efficient data representation if signatures are processed in parallel instead of the pixel-level parallelism that we are currently using. In this mode, a row is computed in parallel for a block of signatures. This should allow the GPU instruction scheduler to be extract more parallelism.

6 Results

We have implemented the matched filter on a variety of platforms. In each system, our objective was to produce a system that is reasonable for production use. This largely means that a CPU, bus interfaces, power supplies, and involved support hardware are included in all of the performance measurements.

We have considered several performance metrics. Speedup compared to the CPU implementation is the primary metric, as throughput per signatures is a primary deciding factor when data volumes are high. Cost and power are secondary factors. Cost is often an object, after all, and for large data volumes the difference in price between a commodity GPU retailing for roughly $500 and an $8,000 IBM Cell processor board from Mercury Computing [3]. The cost of GPUs, FPGA, and Cell-based systems also must include the price of a host machine that can support the add-on board. The ideal approach for large data centers is to maximize the speedup per cost, captured in the second to last column.

Power is of utmost concern in embedded systems. While we are only concerned with a basic matched filter kernel, a more elaborate system would perform the matched filter on the embedded system and then only transmit the interesting bits to the base station. This is an appropriate approach for satellite or aerial reconaissance. In these situations, power is measurement of instantaneous power when the system is active. Of course, because the add-on cards provide an increase in time performance, the system could feasible be shut off for some percentage of the time, so energy – (time * power) – is a useful metric. We capture energy in the form of speedup/watt, the final column of the table.

Our results for the various implementations are listed Table 6. The results are normalized against an Opteron 2.8 GHz with 2 GB RAM and a 15k RPM SATA harddrive. In all situations, the timing results are based on the average of several runs of the filter. The highest speedup we post, 12.4x over the CPU, is for the IBM Cell processor. This is for the first version of the Cell, running at xx GHz.

In terms of speedup per dollar, the GeForce 7900 GTX GPU far outstrips its competition. Because the GPU is a

commodity add-on card that is sold in vast numbers, its price is very reasonable compared to the more esoteric FPGA and Cell co-processors. The value of the GPU in performance per dollar is orders of magnitude faster than the Cell.

The FPGA does not fare well in the comparisons. Because only the dot-product was implemented on the FPGA, the maximum possible speedup over the sofware version was 2x. We have achieved nearly that through extensive parallelism, but are limited by the difficulty of implementing the more control intensive and scalar operations of the filter.

7. Conclusions

In this work, we evaluated the performance of a matched filter algorithm implementation on accelerated co-processor (XD1000), the IBM Cell microprocessor, and the NVIDIA GeForce 6900 GTX GPU graphics card.

We found the FPGA-based implementation to be highly scalable and very effective at calculating the dot-product portion of the computation in parallel, but ultimately found Amdahl's law to be a limiting factor.

The Cell proved to be the highest performing solution from a time and throughput perspective. The Cell implements vector processors that easily map the matched filter, as well as the miscellaneous scalar operations and setup data transfer.

The GeForce 7900 GTX GPU co-processor was found to be the highest value in terms of performance per dollar. The GPU also maps the vectorizable filter computations easily, but requires more elaborate software design as the devices are meant for graphics-centric applications and not general purpose computation.

References

[1] Advanced micro devices - graphics and media processors, 2006. http://ati.amd.com/.

[2] Nvidia corporation, 2006. http://www.nvidia.com/.

[3] Mercury Computer Systems, 2007. http://www.mc.com/.

[4] J. H. Everitt and C. J. Deloach. Remote sensing of chinese tamarisk (tamarix chinensis) and associated vegetation. *Weed Science*, 38:273–278, 1990.

[5] K. S. Hemmert and K. D. Underwood. Open source high performance floating-point modules. *fccm*, 0:349–350, 2006.

[6] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Symp. Sparse Matrix Comput. Appl.*, pages 256–282, November 1978.

[7] J. Meredith, J. Conger, Y. Liu, and J. Johnson. Evaluating mobile graphics processing units (gpus) for real-time resource constrained applications. Technical report, Laurence Livermore National Laboratory, Nov 2005.

[8] M. Pharr, editor. *GPUGems2*. Addison Wesley, 2005.

[9] K. N. Roark. Laboratory reaches for the petaflop. September 2006. http://www.lanl.gov/news/index.php/fuseaction/nb.story/story_id/8932/nb_date/2006-09-07.

[10] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *ACM/SIGDA Twelfth ACM Int'l Symposium on Field-Programmable Gate Arrays (FPGA 2004)*, pages 171–180, Feb. 2004.

[11] L. Zhuo and V. K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *Proceedings of SuperComputing 2005*, Nov 2005.

Implementation	Speedup over CPU	Time per Signature	Cost	Power	Speedup/$k	Speedup/mWatt
FPGA	2.1	3.2 sec	$10000	350 W	0.2	3
GPU	6.2	1.0 sec	$2550	350 W	26	3
Cell	12.4	0.5 sec	$8000	315 W	1.55	3
CPU	1	6.2 sec	$2000	300 W	0.5	3