Rotationally Invariant Sparse Patch Matching on GPU and FPGA

Zachary K. Baker and Reid Porter Los Alamos National Laboratory
Los Alamos, NM 87545
Email: {zbaker,rporter}@lanl.gov

Abstract

1 Introduction

Local neighborhood operations are used extensively in image processing and computer vision, can be substantially accelerated with custom hardware platforms. By tailoring the memory architecture, performance improvement of two orders of magnitude compared to von-Neumann architectures can often be obtained. One of the reasons why these applications can be accelerated so successfully is due to the fact that the neighborhood operation is dense. That is, the same operation is applied independently to each pixel in the image. Image pixels lie on a regular grid, and the memory access can take advantage of this regularity.

In recent years, a number of algorithms in image processing have been proposed which are not dense, and instead apply local neighborhood operations to a sparse, irregular set of points. These algorithms are now used as widely as their dense counterparts and appear in many similar applications e.g. finding correspondences, recognition and segmentation. Sparse local neighborhood operations have two main motivations: 1) Since the local neighborhood operation is applied to a subset of the pixels, computation is significantly reduced 2) The subset of pixels can be selected for the application at hand, and hence improve the performance of some algorithms.

The goal of this work is to demonstrate that high levels of computational burden can be extracted from otherwise sparse computation, namely, the all-pairs rotationally-invariant patch match problem.

Modern processors have more processing power than they have memory bandwidth. This is a common theme even when considering next-generation processors such as the IBM Cell. In these processors, performance improvements will not be achieved if the same code used on an old machine is used. Even if the code is ported to take advantage of the new architectural enhancements of the new processor, the memory bandwidth will still constrain the ability of the processor to take advantage of all of its capability.

In these cases, the processor's potential can be exploited through the expansion of computation. If, for the same data requirements, multiple computation operations can be performed, the relative performance of the system can be increased. Given a static codebase, the changes that this new optimization paradigm leads to are conversions from look up tables to on-the-fly computation, or recomputing an intermediary value in lieu of saving and retrieving it for later use.

2 Problem Addressed

Given two sequential frames of video data, we hope to track elements between the two frames. Candidate elements for matching are determined by searching the area around interest points for their best match in the next frame. This is done by comparing a $k \times k$ pixel block around the interest point in a pixel-wise comparison. A convolutional filter can be applied to the pixels to reduce misleading noise. Weighting can be performed to bias the system toward patches that have smaller displacements between frames. In the spirit of increasing the computational burden

Given a large number of patches to compare between two frames, the problem becomes bottlenecked by the cost of moving data in and out of the processing units. This is particularly a problem for moving data out, as the number of results is proportional to the computation complexity of the problem, or n^2. While the input data is well-matched to the complexity, requiring only nk^2 input elements for the n^2k^2 computation required, the output does slow the whole system.

When converting this algorithm to accelerated hardware platforms, there are several approaches we can take the improve the overall performance. The first of these is reducing the number of output elements by determining the weighted best match over all of the possible candidates. This allows us to only output n outputs. The second and more interesting approach is to introduce rotational invariance to the problem.

Rotational invariance is computationally intensive. Invariance is produced by simply rotating all of the patches over a range of angles to determine if they would better match at a different angle. This is particularly important in a tracking application as vehicles and people tend to turn as they move through the frame. By rotating and comparing patches over a range of rotational angles, it is possible to more accurately track them through the scene.

However, the rotations are clearly computationally expensive. Assuming that the angles are fixed, a designer can avoid recomputing the sine and cosine values required for the rotation, but still must do the vector multiplications and interpolation required to produce a smoothly rotated image.

3 Accelerated Hardware

3.1 Field Programmable Gate Arrays

FPGA's provide a fabric upon which applications can be built. FPGAs, in particular, SRAM based FPGAs from Xilinx [?] or Altera [?] are based on a look-up tables, flip-flops, and multiplexers. In these devices, a SRAM bank serves as a configuration memory that controls all of the functionality of the device, from the logic implemented to the signaling standards of the IO pins. The values in the look-up tables can produce any combinational logic functionality necessary, the flip-flops provide integrated state elements, and the SRAM-controlled routing direct logic values into the appropriate paths to produce the desired architecture. The device is composed of many thousands of basic *logic cells* that include the basic logic elements, and based on the device variety, includes fast ASIC multipliers, ethernet MACs, local RAMs, and clock managers.

We believe that intelligent architectural design is worth more than a simple implementation. There are many tools for automatically converting a high-level language program into a low-level design. With recent advances in compiler and synthesis technology, it is now possible to map the computationally intensive modules of a program in Fortran, C (SRC Carte [?] and Celoxica [?]), or Java (Xilinx Forge [?]) to hardware. Due to the development of these high-level design tools [?, ?], the scientific computing community can utilize reconfigurable devices without steep learning curves.

However, while the ease and popularity of using FPGAs for application design has increased, the automatically generated architectures tend to be inefficient compared to well-researched and thought-out architectures for complex applications. In these situations, the creativity and domain knowledge relevant to a design provided by a human designer is a valuable asset.

3.2 Graphics Processors

The Graphics Processing Unit (GPU) is 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 [3]:

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 fewer 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.

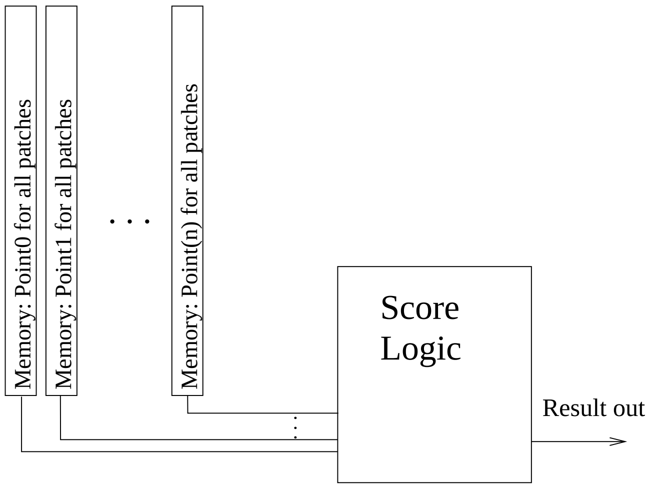
Single result outputs - this is a particular restriction of GPU architectures before the NVIDIA GeForce 8800 [2], released in late 2006. This model assumes that all calls to a subroutine will produce a single return value. This is a reasonable assumption for many graphics-centric applications, as well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

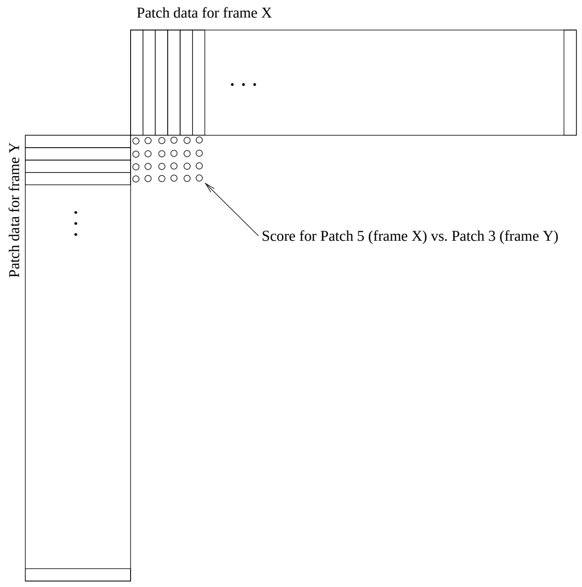
In this model, computations executed on the GPU behave much like single-return value functions. This can be inconvenient, as we will see later, as many computations have side effects and multiple results. In these situations, the single pixel output restriction causes us to repeat computation or build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

Vectorizable Code - There are two paradigms for this ap-


Memory: Point0 for all patches
Memory: Point1 for all patches
. . .
Memory: Point(n) for all patches
Score Logic
Result out



proach 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.

4 FPGA Base Architecture

The FPGA system is based around an Annapolis Wildstar II board with a PCI-X interface and dual Virtex II Pro 100's. This card provides sufficient bandwidth between main memory and the FPGA, as well as large QDR SRAMs useful in storing intermediary results and image data.

Software is responsible for filtering out the area around interest points into patches. This allows much higher efficiency compared with the FPGA accepting entire frames and then deciding what to keep. An alternate system was considered that filtered the entire frame. However, the system was only competitive if the number of patches in a frame is very high. Specifically, because the number of computational steps for n input patches is n^2, the ratio of k element patches to frame size must be at least n^2/kn to keep a FIFO full.

The architecture computes the sum of absolute differences on an entire patch with a throughput of one cycle per pair of patches. Even at the 50MHz clock rate of the board, the ability to compute a large number of computation in parallel that would normally be carried out sequentially is large benefit.

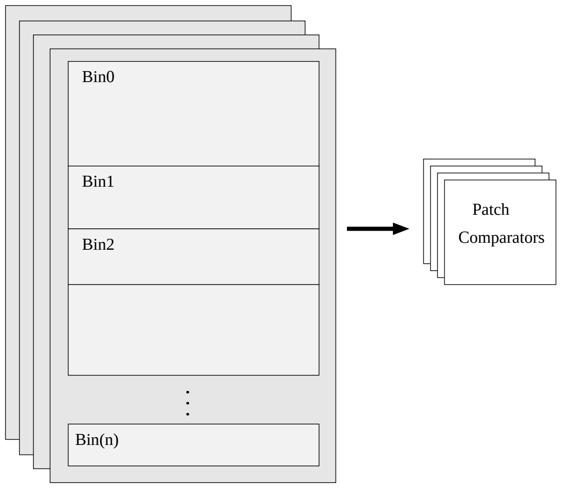
The amount of parallelism in patch comparisons is limited only by the size of the FPGA. Table **??** illustrates the area requirements of the FPGA for a variety of patch sizes. For a relatively large 10x10 patch, the 100 accumulators are generated.

```
for i = 1 to n
  for j = 1 to n
    parallel: for k = 1 to k
        sum += abs(patcharray(i)(k) -
                   patcharray(j)(k));
```

The large amount of distributed RAM that can be synthesized in modern FPGAs makes possible some architectural enhancements that make the system possible. The vast computational capabilities of the FPGA can only be harnessed if a similar amount of on-chip bandwidth can be matched to the computational units.

Figure **??** illustrates the datapath within the system. Each pixel in a patch has its own memory. There are a total of k memories in the system, each having dual memory ports. Each of these memories contains the ith pixel element in the patch for all n. Because the memories have two ports, there is only one copy of the pixel in the system,yet the comparators can be fed two different patches simultaneously. For the k=100 case, having 2*100 parallel memory ports on any traditional microprocessor is clearly out of the question. However, on an FPGA, it is well within reason. The Virtex-II Pro has 1.3 Mbits of available on chip distributed RAM and 7.9 Mbits of Block RAM. In this case, Block RAM is utilized, which provides true dual memory ports and does not use any logic slice resources.

Several other options were considered for this architecture, including a systolic array approach that might have some of the benefits of a closer interconnect and a natural approach to doing the all-pairs comparison of the patches.



The memory issue was the deciding factor in moving toward the more direct approach. The problem is that in order for each element of the systolic array (each doing the work for a single patch) to compare against a patch streaming through the linear array, there must be a great deal of interconnect within the array, specifically, k buses of w data bits. This is not out of the question, but the combination of the large amount of data bus and the size of the individual units, the overall interconnect requirements made the systolic array approach less viable. Additionally, the individual RAM-per-pixel approach requires far less logical control.

5 Rotational Invariance

Computational burden per byte largely determines the efficiency of the FPGA system. Much like the IBM Cell, there is far more computational capability than bandwidth outside of the chip. Thus, if we can increase the computation while keeping the bandwidth requirements constant, the FPGA can become far more useful.

Rotational invariance in the patch matching operation is achieved by explicitly rotating one of the patches over a range of angles, providing linear interpolation of the rotated pixels, and then proceeding as before with the sum of absolute differences.

Figure 5 shows the result of a hardware rotation with only nearest neighbor interpolation. In hardware, we provide linear interpolation between the nearest pixels. This provides a much higher quality end result, as illustrated in Figure 5.

Depending on the choice of architecture, rotational invariance can dramatically increase the total number of memory ports required in the FPGA. Normally, the system requires $2k$ memory ports. However, with the addtion of rotation, we have to support the ability to randomly address a location in memory. In general, the system would require $4k$ ports for the patch to be rotated, and then another $2k$ ports for the unrotated comparision patch.

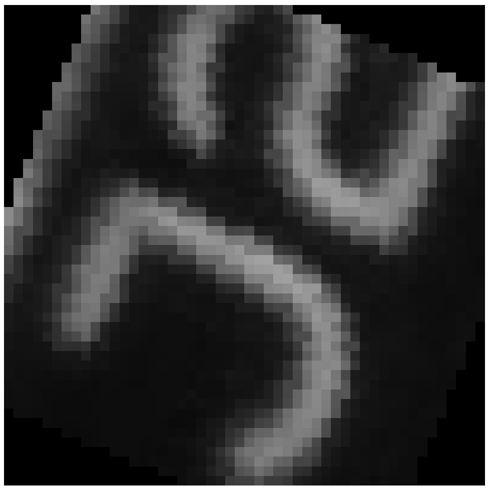


Figure 1. Patch rotated without hardware interpolation

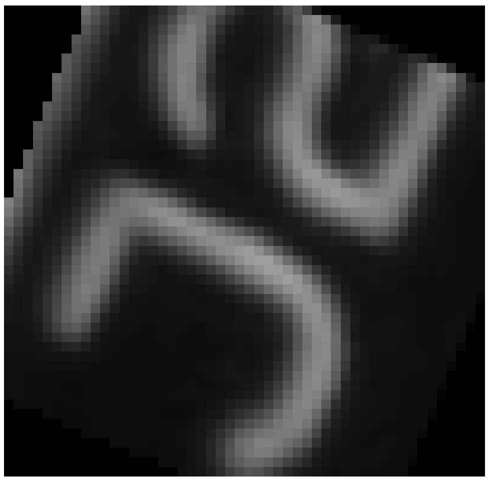


Figure 2. Patch rotated with hardware interpolation

Bandwidth	Size	
k	Slices	Block RAM
25	3112	
64	7881	
100	10880	

Table 1. Bandwidth vs resource usage for FGPA implementation

However, a small limitation can dramatically simplify the architecture and requirements. If the available angles are limited to a limited number of fixed values, the computation for interpolation system can be hardwired into the FPGA. We provide eight segments between 0 and 180 degrees, allowing a car or person to rotate in the most expected range of angles.

The interpolation is further simplified by only providing linear interpolation with three bits of precision.

An example of the generated VHDL for a few points at 45 degrees is as follows:

The code is generated by a script parameterized to allow for customizable angle selection, number of patches, number of elements per patch, and width of data.

The code is pipelined to allow it to run over 100MHz. Note that the interpolation is computed from four individual memory ports. First, the horizontal weighted mean is computed, and then the vertical weighted mean is computed from the two horizontal means. Because the interpolation is completely hardwired, there is no requirement for flexibility in the memory controller. All of the memory ports controlled for the interpolated patch have the same memory address. Thus, only k memory ports are required for the interpolation. If arbitrary angles were allowed and it was not possible to use hardwired computational units, the system would require $4k$ ports. This is due to requiring 4 corners for each of the interpolated points.

The choice of computing in octal (three bits) allows for highly efficient multiplication in the interpolation process. Any interpolation is a maximum of three shifts and adds of the pixel output from the patch memory. To normalize the interpolated pixel, all that is necessary is right shift by three digits.

When a patch is rotated, clearly it does lose some of its data as the patch hangs off the corners of the new display area. In these cases, the difference between the patches is forced to zero, as the undefined data should not be counted as a penalization.

The result from each of the rotation units is collected and the best match is reported, along with the angle at which it occured. The n^2 results are buffered to SRAM and then streamed out via DMA.

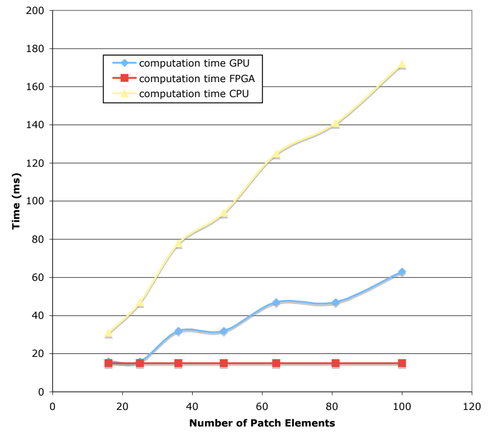


Figure 4. FPGA computes entire patch in one cycle (1000 patches)

6 Results

Tests preformed on CPU is dual 2.6 GHz Opteron Annapolis Wildstar II FPGA board with dual Xilinx V2P100 and an NVIDIA GeForce 7900 GTX. The two cards are based on the same motherboard, with the GPU on a PCI-E bus interface and the FPGA on a full-width PCI-X interface.

Figure 6 illustrates the compute time vs the number of patches. This figure does not include the setup time, captured in Figure 6. As the number of patches k increases, the total amount of computation increases as $O(n^2)$. The FPGA has a predictable computational curve, as it operates in lockstep with one result generated every cycle at 50MHz. After the system is loaded, the internal data is held entirely in on-FPGA memory, providing guaranteed service.

For a 25 element patch, the GPU is somewhat more efficient than the FPGA. The FPGA wins when there are is a huge amount of parallelism available. As the size of the patch increases, the amount of parallel computation available also increases. Figure 6 demonstrates that a patch size of 25 elements is roughly the high end of the range where the GPU can out-perform the FPGA. The FPGA dataline in Figure 6 is interesting because it is constant across the varying number of patch elements. This is due to the architecture of the patch computation on the FPGA; because the FPGA has enough computation resources to compute a large number of parallel patch elements in parallel, the increased size of the patch does not result in increased latency of computation.

Figure 6 illustrates the total time required for a single

```
int_0_6_9_h_l <= (6 * conv_integer(rd_data(9*10+3)));
int_0_6_9_h_r <= (2 * conv_integer(rd_data(9*10+2)));
int_0_6_9_h <= int_0_6_9_h_l + int_0_6_9_h_r;
int_0_6_9_l_l  <= (6 * conv_integer(rd_data(8*10+3)));
int_0_6_9_l_r  <= (2 *  conv_integer(rd_data(8*10+2)));
int_0_6_9_l  <= int_0_6_9_l_l + int_0_6_9_l_r;
int_0_6_9_l_t <= 5 * int_0_6_9_h;
int_0_6_9_r_t <= 3 * int_0_6_9_l;
int_0_6_9 <= int_0_6_9_l_t + int_0_6_9_r_t;
int_out(0)(69) <= int_0_6_9;
```

Figure 3. Interpolating Code Sample

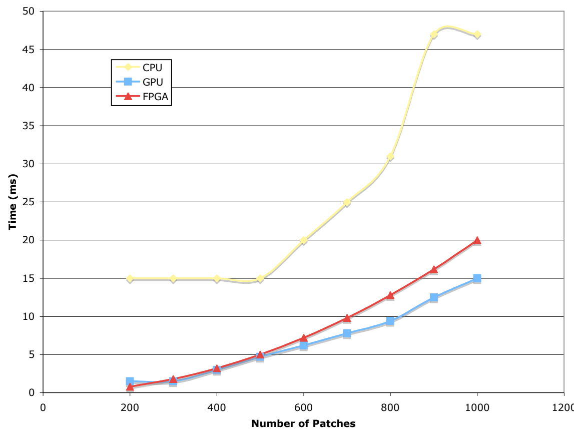


Figure 5. 25 element patch

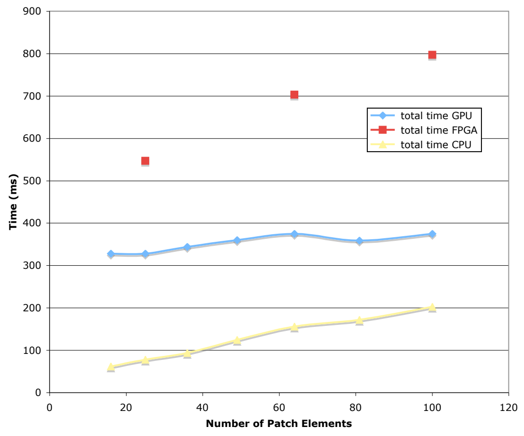


Figure 6. 1000 patches : Most of total time is board setup (can be amortized)

set of 1000 patches compared against each other for various sizes of patches. For these results, the total time includes the time required for board setup on both the GPU and the FPGA.

On the FPGA, this time can easily be amortized across multiple sets of patches. While the patch data has to be transferred for every set of patches, the FPGA programming is a high one-time setup cost. After the FPGA is programmed, requiring hundreds of milliseconds, it can continue opererating indefinitely.

The GPU setup cost is largely initializing the card and making requests through the OpenGL driver system. This does not require nearly as much time as the FPGA initialization, but is still noticeable compared to the CPU, which has very little overhead except opening the input and output files.

¡insert various discussion of beautiful rotational results here¿

References

[1] Advanced micro devices - graphics and media processors, 2006. `http://ati.amd.com/`.
[2] Nvidia corporation, 2006. `http://www.nvidia.com/`.
[3] M. Pharr, editor. *GPUGems2*. Addison Wesley, 2005.