Backprojection algorithms for multicore and GPU architectures

William Chapman, Sanjay Ranka, Sartaj Sahni, Mark Schmalz
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Linda Moore, Uttam Majumder
ARFL/RYAP, Dayton OH 45431

Bracy Elton
High Performance Technologies Group
Dynamics Research Corporation, WPAFB, OH 45433

May 23, 2012

Chapter 1

Backprojection algorithms for multicore and GPU architectures

1.1 Summary of Backprojection

Backprojection is an algorithmic technique that generates 2-dimensional images from synthetic aperture radar data. The data input to the Backprojection algorithm are usually collected by airborne sensors circling around a target area, which emit a series of radar pulses, then receive and record the reflected temporal response. A single pulse provides information about the intensity of reflectors at many distances from the pulse location. Reflectors far from the pulse emitter will appear later in the received response than proximal reflectors. The time t at which a reflector's contribution will appear in the response can be predicted using the speed of light c and the distance d from the pulse emitter, as

$$t = \frac{d}{c}$$

The relationship above facilitates the division of each pulse response into discrete time intervals or range bins, which correspond to the average pulse in each discretized time interval. For example, consider the diagram of Figure 1.1, where a pulse is transmitted toward a target area containing a single point reflector.

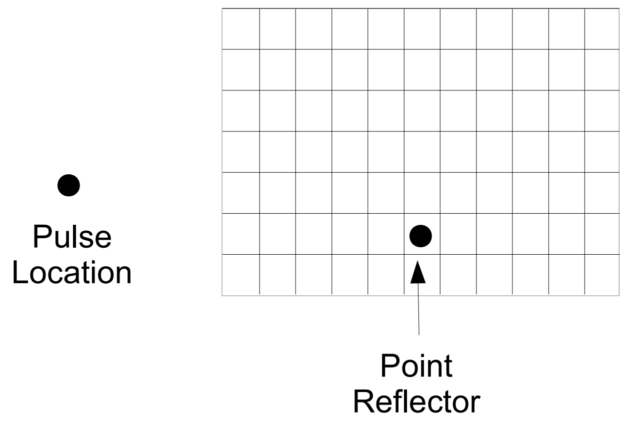

Figure 1.1: Point Reflector in a Target Area

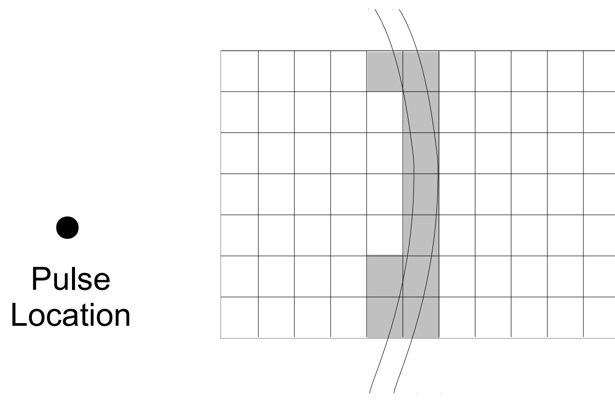

Figure 1.2: Single Pulse View of Point Reflector

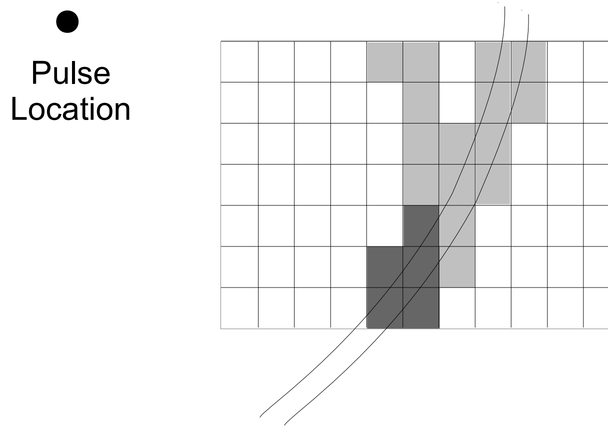

Figure 1.3: Two Pulse View of Point Reflector

An ideal pulse response for this arrangement would consist of a single range bin with a high response intensity, where all other bins would have zero intensity. Note that this type of response contains enough information to infer that one or more reflectors occur at a known distance from the pulse location. Thus, the corresponding image of a target area would appear as shown in Figure 1.2.

The collection of a second pulse at a different sensor location (shown with respect to Figure 1.2 as the superposition of two pulse returns in Figure 1.3) can help determine the presence of one or more reflectors at known distances from both pulse locations. If only one point reflector is known to be present, then a likely position of the reflector can often be inferred. However, in the general case of numerous reflectors, such an inference is not necessarily correct.

The Backprojection algorithm applies this superposition process to numerous pulses, to produce a clear reconstruction of the target area. For each pixel in the output image, and for each pulse which contributes to that pixel, the range bin corresponding to the given pixel is computed, then the value of that range bin is summed to the output image.

This simple Backprojection algorithm presents both data movement and computational challenges. With respect to data movement, the need for high resolution output images implies a large number of pulses divided into a large

number of range bins. In practical implementations, neither the pulse data nor the output image can be stored in the GPU memory that sits closest to processing cores, referred to as shared memory or L1 cache. Thus, an efficient implementation of Backprojection on a GPU must ensure that the correct subset of pulse data and image data are available in shared memory when needed.

In addition to data movement overhead, primary computational cost is incurred by summation of each pulse's contribution to each pixel of the output image. This accumulation process begins with a range calculation that determines (1) the distance from the pixel to the pulse location, then (2) the weighted average of the two range bins that most closely correspond to this distance. A phase correction step that handles pixels occurring at fractional multiples of the radar frequency band is applied to this weighted average before it is summed to the output image. In a Backprojection implementation we are aware of, these operations require 43 floating point operations per pixel, per pulse. The high level structure of Backprojection is captured by the following equation:

$$\begin{aligned} image[x][y] = \sum_{i=0}^{|p|} (p_i[bin(i,x,y)]w0(i,x,y) + \\ p_i[bin(i,x,y)+1]w1(i,x,y))dphase(i,x,y) \end{aligned} \tag{1.1}$$

where $image$ is the image data; x and y are the x and y locations of a pixel; p is the pulse response data; p_i is the response from pulse i; $bin(i,x,y)$ is a function which returns the range bin corresponding to the distance of pixel x, y from pulse i; $w0$ and $w1$ are coefficients used to interpolate the response for pixels that fall between range bins; and dphase is a function which produces the phase offset of pixel x, y with respect to pulse i.

1.2 Partitioning Backprojection for Implementation on a GPU

The key challenge of Backprojection is data size: both the input data (array of pulses (rows) and range bins (columns)) and the output image are often too large to fit in the GPU's shared memory or L1 cache. Consequently, a partitioning scheme is required that facilitates access locality on both data structures.

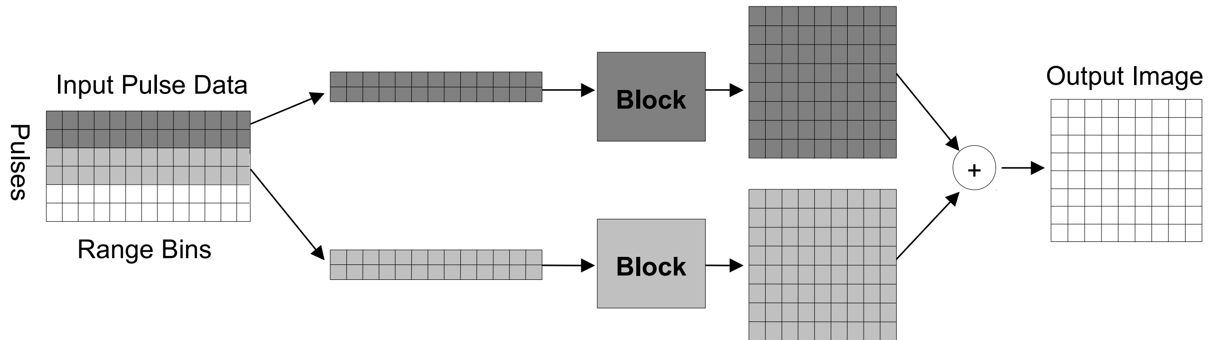

Figure 1.4: Partitioning Backprojection Along Pulse Data

The most natural approach partitions along the rows (along pulses) of the echo data matrix. A block of pulses is sent to each multiprocessor, which produces an output image that corresponds to the contribution of its pulses. Then, the resulting output images are summed to create a reconstructed image. This approach, shown schematically in Figure 1.4, exploits Backprojection's natural parallelism with respect to pulses, as each pixel can be thought of as the sum of the contribution of each pulse.

Although having the advantage of simplicity, this approach is confounded by the size of the output image. Partitioning along the pulse dimension requires that each partition maintain a copy of the output image, or share access to a single copy. The former approach is prohibited by the small size of GPU shared memory, while the latter suffers the performance penalties of global memory.

A second approach is not inhibited by these complications. Partitioning the image into two-dimensional (2-D) tiles allows small parts of both the output image and pulse data to be cached together in shared memory. This follows from the fact that the number of pulses required for rendering an image tile is linearly related to the width and height of the tile. The partitioning is shown in Figure 1.5.

The key process of selecting a correct tile size is examined in greater detail in Section 1.4.1.

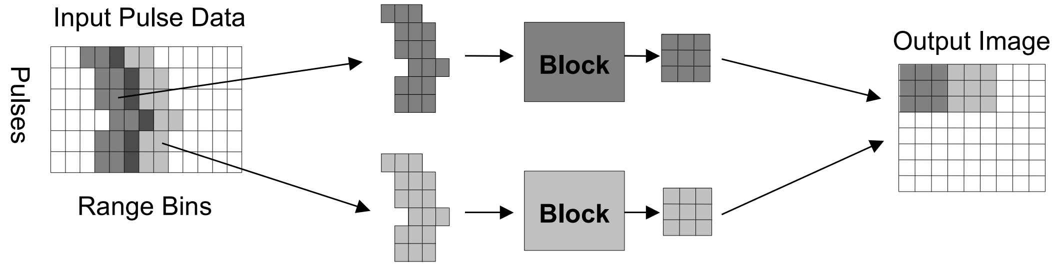

Figure 1.5: Tile Partitioning of Backprojection

1.3 Single Core Backprojection

The single-core C code to implement Backprojection is a straightforward implementation of the algorithm shown in the introduction, and is presented below. Note that a small improvement in performance can be realized by iterating over the pulses in the outer loop instead of the inner loop. Unfortunately, this does not facilitate block parallelization, so pulses are iterated inside the main loop for consistency with other code blocks.

The following data structures are input parameters to this algorithm: X and Y, the real coordinates corresponding to each pixel in the target area; r0_data, the range of the first bin in the pulse data; f0_data, frequency information about radar pulse; x_obs, y_obs, z_obs, the real coordinates corresponding to each pulse; pulse_data, the complex pulse response data; and c4df, a constant that contains information necessary to compute the bin index and phase corrector for each pixel. The algorithm produces a single output, image, which contains the complex response projected onto each pixel.

This code computes a 4096 x 4096 pixel radar image in 1625.7 seconds, at a throughput of 0.44 Gflop/s on a 6 Core 3.3 GHz Intel(R) CoreTM i7 CPU with a 12 MB Cache.

```
for(j=0; j < image_height; j++){
   for(k=0; k < image_width; k++){
      //get position of pixel
      pixel0_X = X[k]; //load pixel x-coordinate;
      pixel0_Y = Y[j]; //load pixel y-coordinate;
      //these values can be computed outside the
      //pulse loop
      pre1 = (num_bins-1)/(2.0*c4df);
      pre2 = 4.0f*pi/sol/2.0f;
      pre3 = c4df*num_bins/(num_bins-1.0f);
      // loop through all pulses for each subtile
      for (i=0;i<num_pulses;i++){
         //determine range to pulse location
         r0 = r0_data[i]; //load r0
         Rstart=r0-pre3;
         //load pulse location x,y,z
         tmpRA = pixel0_X - x_obs[i];
         R = tmpRA*tmpRA;
         tmpRA = pixel0_Y-y_obs[i];
         R += tmpRA*tmpRA;
         tmpRA=z_obs[i];
         R += tmpRA*tmpRA;
         //compute the range
         R=sqrt(R);
         //compute bin index and coefficients
         binFloat = (R-Rstart)*pre1;
         binFloor = (int)binFloat;
         w2 = binFloat - binFloor;
         w1 = 1.0f-w2;
         //validate bin index
         if(binFloor+1 < num_bins && binFloor > 0){
            //add pulse offset to memory address
            binFloor += i*num_bins;
            //load the frequency
            f0 = f0_data[i];
            //retrieve the bins
            tmpRB=pulse_data[(int)(binFloor)];
            tmpRC=pulse_data[(int)(binFloor+1)];
            //extract components from bins
            tmpRF=(float)tmpRB.x; //bin1 real
            tmpRG=(float)tmpRB.y; //bin1 imaginary
            tmpRD=(float)tmpRC.x; //bin2 real
            tmpRE=(float)tmpRC.y; //bin2 imaginary
            //compute the non phase corrected sum
            tmpRD = w1*tmpRF + w2*tmpRD;
            tmpRE = w1*tmpRG + w2*tmpRE;
            //populate the phase corrector (f2)
            tmpRF = pre2*f0*((float)R);
            tmpRH = static_cast<int>(tmpRF / pi);
            tmpRG = tmpRF - static_cast<float>(tmpRH)
               * ((float)pi);
            tmpRF = tanf((float)tmpRG);
            tmpRG = tmpRF*tmpRF + 1.0f;
            f2.x = (2.0f - tmpRG) / tmpRG;
            f2.y = (2.0f * tmpRF) / tmpRG;
            //sum phase corrected result to the image
            pixel.x += (tmpRD*f2.x-tmpRE*f2.y);
            pixel.y += (tmpRE*f2.x+tmpRD*f2.y);
         }
      }
      //write back to the image
      image_data[(int)
         (k+j*image_width)].x=(float)pixel.x;
      image_data[(int)
         (k+j*image_width)].y=(float)pixel.y;
   }
}
```

Figure 1.6: Single Core C Implementation of Backprojection

```
for(pb=0; pb < num_pulses; pb += pulse_block_size){
   for(by=0; by < blockCountY; by++){
      for(bx=0; bx < blockCountX; bx++){
         for(j=by*BLOCK_SIZE; j < (by+1)*BLOCK_SIZE; j++){
            for(k=bx*BLOCK_SIZE; k < (bx+1)*BLOCK_SIZE; k++){

               //The main body of the loop is consistent with Figure 6
               //except for the following change to the inner for loop.

               // loop through a pulse block for each sub tile
               for (i=pb; i < pb+pulse_block_size ; i++){
                  ...
               }
            }
         }
      }
   }
}
```

Figure 1.7: Single Core Cache Aware Implementation of Backprojection

1.3.1 Single Core Cache-Aware Backprojection

Improvements to the single-core code can be realized by considering the size of the CPU cache, then implementing a tiled partitioning scheme that ensures access locality on both pulse data and image data. Each partition in this scheme is visited sequentially, allowing data required for each partition to be stored entirely in cache until starting the next partition. This cache-aware strategy improves the performance of Backprojection by reducing the number of cache misses and, in turn, the number of system memory accesses.

The implementation of this scheme only requires a modification to the outer loop of Figure 1.6. In particular, rather than loop through the pixels in an entire output image first along one dimension (e.g. y-axis), then along the other dimension (e.g., x-axis), the approach shown in Figure 1.7 visits the pixels and pulses in blocks.

1.3.2 Multicore Cache-Aware Backprojection

A multithreaded implementation of the code in Figure 1.7 can exploit a CPU's ability to hide memory latency by overlapping memory access with

Table 1.1: Single Core, Cache Aware Backprojection Performance

Pulse Block Size	Tile Size (px)	Latency (seconds)	Throughput (Gflop/s)
1	4	1922.45	0.35
1	8	1933.31	0.35
1	16	1964.24	0.34
1	32	1984.44	0.34
10	4	1536.37	0.44
10	8	1538.51	0.44
10	16	1545.15	0.43
10	32	1553.52	0.43
20	4	1506.62	0.45
20	8	1509.02	0.45
20	16	1512.09	0.44
20	32	1517.19	0.44
30	4	1798.66	0.37
30	8	1803.19	0.37
30	16	1805.68	0.37
30	32	1805.76	0.37

computation, and can also utilize multiple cores. The corresponding extension of the code in Figure 1.7 is trivial, as a block partitioning scheme has already been implemented. Pthreads can be used to launch N instances of the code, where each instance is provided with a thread ID that can be used to identify itself. Inside the main block loop, a single line of additional code could then skip to process each Nth block. This modification is shown in Figure 1.8, with experimental performance figures listed in Table 1.2. A speedup of more than the number of processing cores was obtained by launching more threads than cores, thus allowing the CPU to mask memory access by overlapping it with computation.

```
for(pb=0; pb < num_pulses; pb += pulse_block_size){
   for(by=0; by < blockCountY; by++){
      for(bx=0; bx < blockCountX; bx++){
         if( (by*blockCountX+bx)%4 != id) continue ;
         for(j=by*BLOCK_SIZE; j < (by+1)*BLOCK_SIZE; j++){
            for(k=bx*BLOCK_SIZE; k < (bx+1)*BLOCK_SIZE; k++){

               //The main body of the loop is consistent with Figure 7

            }
         }
      }
   }
}
```

Figure 1.8: Modification of Code in Figure 7 to Realize Multithreaded Execution

1.4 GPU Backprojection

1.4.1 Tiled Partitioning

An initial implementation of Backprojection on the GPU is straightforward using the tiled partitioning scheme described in Section 1.2. Each block of threads on the GPU corresponds to one image tile, where each thread corresponds to one pixel. The pulse data, image, and other data structures are transferred to the GPU using multiple invocations of the Nvidia CUDA operation cudaMemcpy prior to launching the kernel. The pulse data and image are stored directly in global memory, while the other data structures are stored in texture memory to improve memory read performance. These latter structures are read-only and are comparatively small.

The kernel code for a GPU implementation of Backprojection is given in Figure 1.9.

The code that invokes this kernel is quite long due to the number of data structures that must be transferred to device memory. The code illustrated in Figure 1.10, which has been truncated for compactness, demonstrates the required steps. Firstly, memory is allocated on the device to hold the data structures, which are then copied from host to device memory. Pointers to the pulse data (d_pulse_data) and image data (d_image) are passed as arguments

Table 1.2: Multithreaded, Cache-Aware Backprojection Performance

Pulse Block Size	Tile Size (px)	Latency (seconds)	Throughput (Gflop/s)
1	4	230.89	2.91
1	8	217.63	3.09
1	16	216.08	3.11
1	32	216.85	3.10
10	4	190.10	3.53
10	8	185.15	3.63
10	16	189.56	3.54
10	32	187.52	3.58
20	4	187.24	3.59
20	8	185.92	3.61
20	16	186.11	3.61
20	32	186.55	3.60
30	4	222.42	3.02
30	8	223.82	3.00
30	16	225.83	2.98
30	32	224.47	2.99

```
__global__ void bploop(float2 *pulse_data, float2
   *image_data,float c4df, int num_pulses,int
   num_bins,int image_width,int image_height){

   float r0, f0, pixel0_Y, pixel0_X, w1, w2,
      binFloat, binFloor, pre1, pre2, pre3, R,
      Rstart, tmpRA, tmpRD, tmpRE, tmpRF, tmpRG;
   float2 pixel0, f2, tmpRC, tmpRB;
   int i, j, k, tmpRH;
   int2 tmpRI;

   //the pixel that corresponds to this thread
   j = (threadIdx.x + blockIdx.x *
      INTERNAL_BLOCK_SIZE);
   k = (threadIdx.y + blockIdx.y *
      INTERNAL_BLOCK_SIZE);

   //load pixel location
   tmpRI=tex1Dfetch(tex_X,j+0);
   pixel0_X=(float) __hiloint2double(tmpRI.y,tmpRI.x);
   tmpRI=tex1Dfetch(tex_Y,k+0);
   pixel0_Y=(float) __hiloint2double(tmpRI.y,tmpRI.x);

   //precompute constants to be used inside the loop
   pre1 = (num_bins-1)/(2.0*c4df);
   pre2 = 4.0f*pi/sol/2.0f;
   pre3 = c4df*num_bins/(num_bins-1.0f);

   // loop through all pulses for each subtile
   for(i=0; i<num_pulses; i++){

      //load r0
      r0=tex1Dfetch(tex_r0_data,i);
      Rstart=r0-pre3;

      //determine range to pulse location
      tmpRA = pixel0_X - tex1Dfetch(tex_x_obs,i);
      R = tmpRA * tmpRA;
      tmpRA = pixel0_Y - tex1Dfetch(tex_y_obs,i);
      R += tmpRA * tmpRA;
      tmpRA = tex1Dfetch(tex_z_obs,i);
      R += tmpRA * tmpRA;
      R = sqrt(R);

      //compute bin index
      binFloat = (R - Rstart) * pre1;
      binFloor = (int) binFloat;
      w2 = binFloat - binFloor;
      w1 = 1.0 - w2;

      //verify the bin is in the subset of the pulse
      //data this thread can access
      if(binFloor+1 < num_bins && binFloor > 0){
         binFloor += i*EXTERNAL_PHD_CACHE;

         //load f0 for the pulse
         f0=tex1Dfetch(tex_f0_data,i);

         //read the bins
         tmpRB = pulse_data[(int)(binFloor)];
         tmpRC = pulse_data[(int)(binFloor+1.0f)];
         tmpRF = tmpRB.x; //bin1.real
         tmpRG = tmpRB.y; //bin1.imag
         tmpRD = tmpRC.x; //bin2.real
         tmpRE = tmpRC.y; //bin2.imag

         //compute f1, the non phase corrected sum
         tmpRD = w1*tmpRF + w2*tmpRD;
         tmpRE = w1*tmpRG + w2*tmpRE;

         //populate the phase corrector (f2)
         tmpRF = pre2 * f0 * R;
         tmpRH = static_cast<int>(tmpRF / pi);
         tmpRG = tmpRF-static_cast<float>(tmpRH*pi);
         tmpRF = __tanf(tmpRG);
         tmpRG = tmpRF * tmpRF + 1.0f;
         f2.x = (2.0f - tmpRG) / tmpRG;
         f2.y = (2.0f * tmpRF) / tmpRG;

         //sum phase corrected result to the image
         pixel0.x += tmpRD*f2.x - tmpRE*f2.y;
         pixel0.y += tmpRE*f2.x + tmpRD*f2.y;
      }
   }

   //write back to global memory for pixel
   image_data[(int)(k+j*image_height)].x=pixel0.x;
   image_data[(int)(k+j*image_height)].y=pixel0.y;
}
```

Figure 1.9: GPU Implementation of Backprojection Kernel

```
float2* d_pulse_data;
float* d_image;
float* d_r0;
...
cudaMalloc((void**) &d_pulse_data, 2*sizeof(float)*num_pulses*num_bins);
cudaMalloc((void**) &d_image, 2*sizeof(float)*BLOCK_SIZE*BLOCK_SIZE);
cudaMalloc((void**) &d_r0, sizeof(float)*num_pulses);
..
cudaMalloc((void**) &d_r0, sizeof(float)*num_pulses);
cudaMemcpy(d_image, &(image[startImageAddress]),
   2*sizeof(float)*BLOCK_SIZE*BLOCK_SIZE,cudaMemcpyHostToDevice);
cudaMemcpyAsync(d_r0, r0, sizeof(float)*BLOCK_SIZE,
   cudaMemcpyHostToDevice);

cudaBindTexture(0, tex_r0, d_r0, sizeof(float)*PULSE_BLOCK_SIZE);

dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(image_width / threads.x, image_height / threads.y);

bploop<<<grid, threads, 0>>>((float2 *)d_pulse_data, (float *)d_image,
   c4df, num_pulses, num_bins, image_width, image_height);

cudaMemcpy(&(image[0]), d_image, 2*sizeof(float)*BLOCK_SIZE*BLOCK_SIZE,
   cudaMemcpyDeviceToHost);
```

Figure 1.10: GPU Implementation of Backprojection Host

to the kernel, while the remaining structures are passed as textures.

The kernel code in Figure 1.10 contains several optimizations that are instrumental in achieving high performance, which are listed as follows.

1. Utilization of L1 Cache Rather than Shared Memory

   The code in Figure 1.10 was developed for use on an Nvidia Tesla C2050, which was the first Nvidia GPU to provide developers with a traditional L1 and L2 cache hierarchy for reducing global memory access costs. Previous GPUs included only a shared memory which had to be manually controlled by the code developer. Operating this memory structure required the addition of code into the kernel that populated and maintained the state of the cache, which incurred additional costs in clock cycles and registers. Because the Tesla C2050's L1 and L2 caches are managed transparently by hardware, they do not incur a performance penalty to initiate or maintain. The effective utilization

of this cache hierarchy is achieved by the same technique traditionally applied to other processing architectures. For all thread blocks that are active on each multiprocessor, memory accesses are ordered in such a way to ensure that proximal reads and writes are localized to a region of memory that is smaller than the cache size divided by the number of active blocks. This achieves a high cache hit ratio. In Backprojection, access locality is attained by implementing the tiled partitioning scheme described in Section 1.2, where the tile size is sufficiently small to permit all pulse data and all image data for each tile to fit inside the L1 cache.

2. Reuse of Variable Names

   Minimizing the number of registers allocated per thread is essential to achieving high occupancy, a condition where many thread blocks are scheduled for execution on each streaming multiprocessor at the same time. In most instances, the CUDA register allocator is effective at analyzing the kernel and determining the lowest number of registers required by the program. However, in some instances, the CUDA compiler is not able to detect when it is safe to reuse a register that is storing old data. Explicit overwriting of the variable with new data ensures that its register will be used to hold the new value.

3. Computation of Values Outside Loops

   This optimization is not specific to the GPU, but is useful because moving expensive operations outside loops often has a positive impact on performance. In the code above, this was done for the variables pre1, pre2, and pre3. On a GPU, the cost of this optimization is seen in register utilization. Values computed outside a loop must be maintained in register memory, and their registers cannot be released until the loop has completed. It is always useful to weight the spatial cost of a single register against the temporal cost of recomputing the value inside the loop.

Recall that the success of a GPU implementation of Backprojection depends on the data partitioning strategy. From the principle of locality, within a group of neighboring pixels, the collection of range bins the pixels correspond to are neighbors or near-neighbors. This access locality permits a subregion of the image and a subregion of the pulse data to be transferred

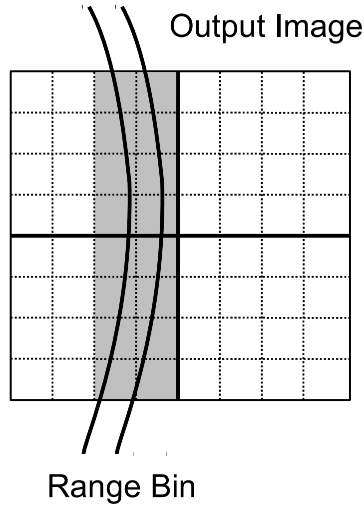

Figure 1.11: Pulse Data Reuse Increases with Tile Size

to shared memory or L1 cache to facilitate complete rendering of the pixel with the given set of pulses.

This technique is advantageous because it allows a GPU to transfer large blocks of pulse data memory at one time, rather than requiring transfer of a single bin each time it is needed. More importantly, this approach also permits frequently-accessed values to be loaded from global memory once, then read from cache for all subsequent read operations (called load-once read-many).

Figure 1.11 helps the reader understand this notion of range bin reuse. For example, let a target area be partitioned into nine tiles or blocks. The bin corresponding to the illustrated pulse location is used by five blocks a total of 12 times. If blocking was not used, this bin would be to be loaded from global memory each of the 12 times it was used, yielding 12 memory accesses. In contrast, the load-once read-many blocking strategy permits the same image to be reconstructed using only five global memory accesses, with the remaining 7 read operations resulting in a cache hit.

In the general case, where blocking is used on high-resolution images having many blocks, the amount of data transfer required to render an image

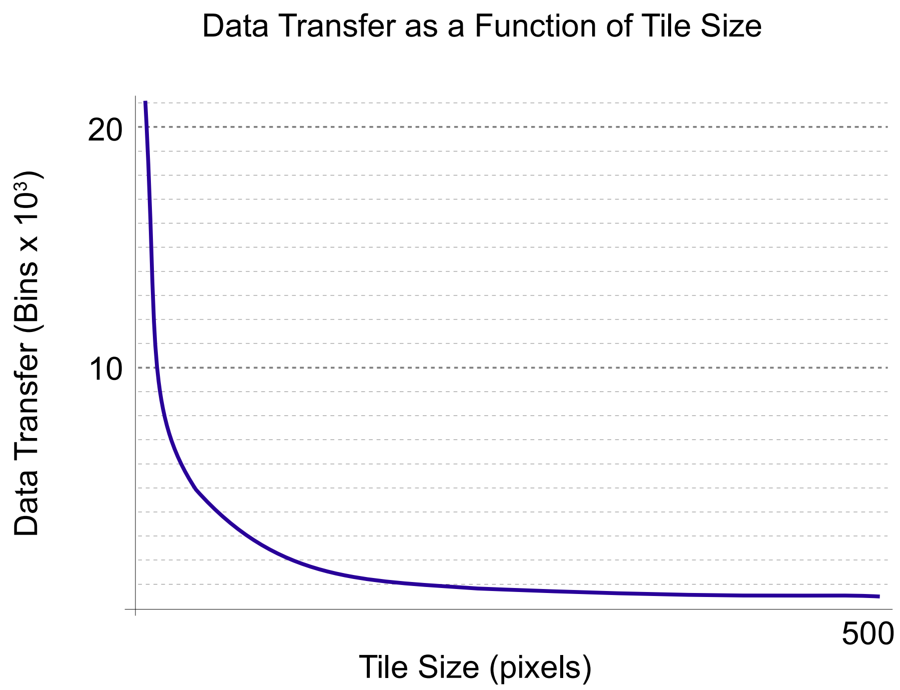

Figure 1.12: Data Transfer Cost Decreases with Tile Size

Table 1.3: Performance of Tiled Backprojection

Tile Size (px)	Latency (seconds)	Throughput (Gflop/s)
4 x 4	14.9	44.1
8 x 8	4.79	131.8
16 x 16	3.71	181.1
32 x 32	3.68	182.4

can be modeled as function of tile size. A smaller block size results in less block reuse, thus increasing communication cost. This effect is shown in Figure 1.12, using simulated low resolution image data (500 x 500 pixels) with 4096 range bins. The two graphs in Figure 1.12 denote upper and lower bounds on the memory access requirements for a given block (tile) size. The bounds differ because pulse look angle can vary with respect to block orientation.

Similar to the preceding discussion, a larger block size reduces communication cost but decreases the number of blocks and the degree of parallelism. Large block sizes can cause low occupancy if register utilization is high, or if pulse data and image data exceed available storage in a block's region of L1 cache and shared memory.

Performance results are shown in Table 1.3, as a function of tile size varying from 4 x 4 pixels to 32 x 32 pixels, for a 4096 x 4096 pixel output image. These results are consistent with the observation that higher tile sizes lead to lower latencies, subject to the constraints of register availability and memory. There does not seem to be a substantial benefit going from 16 x 16 to 32 x 32, suggesting that memory access due to tile size is not significantly limiting performance at these sizes.

1.4.2 Overlapping Host - Device Communication with Computation

While improvements presented in the previous section are impressive when compared to the CPU implementation, one can further increase performance by reducing the latency incurred by transferring the pulse data from host to device before launching the kernel. Since pulse data is a large structure, this transfer latency can account for a significant portion of the application runtime. Fortunately, the Backprojection algorithm sequentially uses each

pulse from the response, without needing to revisit previous pulses or look ahead to subsequent pulses. This permits the pulse data to be treated as a datastream, so pulses can be copied into device global memory while the kernel is running. This overlap allows masking of nearly all the pulse data transfer latency.

Modern GPU devices support the overlap of communication and computation via a CUDA structure called a stream that is a set of GPU instructions, including kernel invocations and memory transfer requests, which is executed sequentially. Such CUDA operation streams should not be confused with datastreams mentioned in the previous section. Within a CUDA stream, operations cannot be overlapped, so must be executed in the order they were placed in the stream. However, between different streams, operations may overlap. For example, a kernel from one stream can be running while an asynchronous memory transfer in another stream is running.

The approach used to achieve this overlap with Backprojection (1) transmits a block of pulses to global memory; (2) launches a kernel that generates an output image from these pulses while a second block of pulses is being copied to another location in global memory; (3) leaves the image in global memory, and (4) launches a kernel on a new block of pulses.

The implementation of CUDA stream-based processing requires only one change to the kernel code shown in Figure 1.9. Previously, a single kernel call processed all pulses in the response, so there was no need to read in the pixel value currently being computed before beginning execution; it was safe to assume the pixel's value was 0. In the CUDA stream implementation, each kernel must read the current pixel value from global memory before entering the main pulse loop. This modification is shown in Figure 1.13. The corresponding modifications to the host code are given in Figure 1.14.

In this code, the host begins by allocating two blocks of memory in the device for pulse data, denoted by d_phd0 and d_phd1, and one block of memory for the output image. Additional memory is allocated for smaller supporting data structures that will later be assigned to textures. CUDA streams are created to hold each memory transfer and kernel invocation pair. The primary loop iterates through the pulse blocks, starting at an index value of -1 and stopping at the block count. During the first pass through the loop, the first block of pulses is transferred to the device. During the second pass, the pulse data for the second block is sent and a kernel is invoked on the first block. Execution continues in this manner until the last iteration, when a kernel is launched on the last block and no data is sent to the device.

```
__global__ void bploop(float2 *pulse_data, float2 *image_data,float c4df,
   int num_pulses,int num_bins,int image_width,int image_height){
   float r0, f0, pixel0_Y, pixel0_X, w1, w2, binFloat, binFloor, pre1,
   pre2, pre3, R, Rstart, tmpRA, tmpRD, tmpRE, tmpRF, tmpRG;
   float2 pixel0, f2, tmpRC, tmpRB;
   int i, j, k, tmpRH;
   int2 tmpRI;

   //determine the pixel that corresponds to this thread
   j = (threadIdx.x+blockIdx.x*INTERNAL_BLOCK_SIZE);
   k = (threadIdx.y+blockIdx.y*INTERNAL_BLOCK_SIZE);

   //read the image from global memory
   pixel0.x = image_data[(int)(k+0+j*image_height)].x;
   pixel0.y = image_data[(int)(k+0+j*image_height)].y;

   for (i=0;i<num_pulses;i++){
      //compute the contribution of each pulse in the block
      //(No changes to the loop body from Figure 8.)
   }

   //write back to global memory for pixel
   image_data[(int)(k+j*image_height)].x=(float)pixel0.x;
   image_data[(int)(k+j*image_height)].y=(float)pixel0.y;
}
```

Figure 1.13: Tiled Backprojection Kernel with Overlapped Communication

```
// allocate device memory for 2 copies of pulse array
float2* d_pulse_data0;
cudaMalloc((void**) &d_pulse_data0,2*sizeof(float)*
   PULSE_BLOCK_SIZE*EXTERNAL_PHD_CACHE);
float2* d_pulse_data1;
cutilSafeCall(cudaMalloc((void**) &d_pulse_data1, 2*
  sizeof(float)*PULSE_BLOCK_SIZE*EXTERNAL_PHD_CACHE));

//allocate memory for other structures used to store
data about the pulses
float* d_x_obs;
cudaMalloc((void**) &d_x_obs, sizeof(float)*
   PULSE_BLOCK_SIZE);
...

//allocate image memory
float* d_image;
cudaMalloc((void**) &d_image, 2*sizeof(float)*
   image_width*image_width);

//zero the image
cudaMemset(d_image, 0, 2*sizeof(float)*
   image_width*image_width);

//make a stream for each block of pulses. The stream
//will later include a memory transfer instruction and
//a kernel invocation.
streams = (cudaStream_t*) malloc(sizeof(cudaStream_t)*
   num_pulses/PULSE_BLOCK_SIZE);

for(k=0; k < num_pulses/PULSE_BLOCK_SIZE; k++)
   cudaStreamCreate(&(streams[k]));

//set up the kernel image partitioning
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(image_width / threads.x, image_width /
   threads.y);

// loop through all pulse blocks
for(k=-1; k<render_num_pulses/PULSE_BLOCK_SIZE; k++){
   //determine if this is an even or odd block index
   pm = ((k+1) % 2 == 0);
   //launch a kernel if this is not the first
   //iteration of the loop
   if(k >= 0){
      // copy small host data structures to device
      cudaMemcpyAsync(d_x_obs,&(x_obs[PULSE_BLOCK_SIZE
          *k]), sizeof(float)*PULSE_BLOCK_SIZE,
          cudaMemcpyHostToDevice,streams[k]);
      ...
      // bind the data to textures
      cudaBindTexture(0, tex_x_obs, d_x_obs,
         sizeof(float)*PULSE_BLOCK_SIZE);
      ...
      // execute the kernel
      bploop<<< grid, threads, 0, streams[k]>>>
         ((float2 *)(pm?d_pulse_data1:d_pulse_data0),
         (float *)d_image,c4df, PULSE_BLOCK_SIZE,
         num_bins,image_width, image_height);
   }

   //send data for the next pulse block if we are not
   //on the last pulse block
   if(k < num_pulses/PULSE_BLOCK_SIZE-1){
      int kplus1 = k+1;
      //send the buffer to the device
      cudaMemcpyAsync( pm ? &(d_pulse_data0[0]):
         &(d_pulse_data1[0]),
         &(pulse_data[num_bins*kplus1*
         PULSE_BLOCK_SIZE*sizeof(float2)]),
         num_bins*PULSE_BLOCK_SIZE*sizeof(float2),
         cudaMemcpyHostToDevice,streams[kplus1]);
   }
   // wait to start the next iteration of the loop
   // until both the kernel and memory transfer are
   // complete
   cudaThreadSynchronize();
}

// copy image from device to host
cudaMemcpy(&(image[startImageAddress]), d_image,
   2*sizeof(float)*image_width*image_width,
   cudaMemcpyDeviceToHost);
```

Figure 1.14: Tiled Backprojection Host Code with Overlapped Communication

Table 1.4: Tiled Backprojection Kernel with Overlapped Communication Performance

Pulse Block Size (px)	Latency (seconds)	Throughput (Gflop/s)
10	3.865	173.8
20	3.547	189.4
50	3.369	199.4
100	3.324	202.1
125	3.327	201.9
200	3.346	200.8
500	3.483	192.9

This technique allows all but the transfer of the first block to be masked by computation time. This fact encourages the selection of small block sizes. However, extremely small block sizes are disadvantageous when the latency associated with launching a kernel exceeds the latency for transferring a single block. This break-even point is hardware dependent.

Performance results are shown in Table 1.4 as a function of pulse block size varying from 10 to 200, for a 4096 x 4096 pixel output image. The image tile size is held constant at 32 x 32 pixels, which was the best-performing tile size from previous tests (Table 1.3), yielding 182.4 Gflop/s. These results demonstrate the observation that, at very low pulse block sizes, latency is high due to communication overhead associated with launching a large number of kernels. At large block sizes, transfer cost for the first pulse block reduces the amount of feasible overlapping. The highest performance was observed at a pulse block size of 100, which increased throughput from 182.4 Gflop/s to 202.1 Gflop/s, an improvement of 10.8%.

1.4.3 Improving Register Usage

The partitioning scheme described in Section 1.4.1 dictates that each kernel should process one pixel. It was subsequently determined that optimum tile size is 32 x 32 pixels, for a total of 1024 kernels per block. In Section 1.4.2, we used streams to overlap computation and communication to attain an improvement in performance. This decreased total latency, but did not impact kernel performance.

Additional improvement can be realized by increasing occupancy attained by our current scheme. In particular, decreasing the resource utilization of

a single thread block allows more thread blocks to be active on a streaming multiprocessor, thus supporting reduction of global memory access times.

Compiling the CUDA code using –ptx-options=-v provides information about the resources being used by each thread, for example:

```
ptxas info: Compiling entry function '...' for 'sm_20'
ptxas info: Used 21 registers, 68 bytes cmem[0], 8 bytes cmem[16]
```

Here, Line 2 indicates that each thread uses 21 registers. Recalling that there are 1024 registers per block, each block thus requires 21,504 registers on a streaming multiprocessor, which is more than half of the 32,768 registers available per streaming multiprocessor. For this reason, only one block can be active at a time, and global memory access cannot be masked.

Although it is possible to reduce the number of registers per block by reducing block size, Table 1.4 illustrates that optimal block size is 32 x 32 pixels. This implies that any benefits from higher occupancy are overcome by the higher communication costs associated with dividing the image into smaller tiles. Fortunately, it is possible to maintain a large tile size while decreasing the number of registers per block, which is accomplished by increasing the number of pixels per thread.

The success of this scheme depends on the notion that, although each thread uses 21 registers, many of those registers are not storing information about the pixel, but about the state of the thread. This includes values computed outside the body of the primary loop (to reduce computation latency), as well as index variables and other intermediate values associated with running the algorithm. One could infer that if each thread processed two pixels, the tile size would double, but the number of registers per thread would increase by a smaller factor. This would allow tile size to be maintained at a large value while also increasing occupancy.

To implement this modification, the kernel must be modified to compute two pixels concurrently. This involves declaring an extra variable pixel_01 to store pixel state, and adding code anywhere image data is accessed or used, to ensure the computation runs twice. It is tempting to wrap the entire application in a large loop where the index is varied from 0 to 1, but this is not an acceptable solution. GPU registers are non-addressable from the perspective of the developer, meaning there is no way to access a different register based on the value of the loop variable, except by using conditional logic. This would deteriorate performance, as it would violate the single

Table 1.5: Tiled Backprojection with Multiple Pixels Per Thread Performance

Pixels Per Thread	Registers Per Thread	Tile Size	Latency (seconds)	Throughput (Gflop/s)
2	25	16 x 32	2.98	221.3
4	32	32 x 32	3.01	219.5

instruction multiple data principle. Such a loop would also require an extra register to maintain the value of the loop variable.

Figure 1.15 illustrates kernel modification for processing two pixels per thread. Implementing this strategy requires few modifications to the host code. Only the grid needs to be corrected, since each block now computes twice as many pixels as it has threads. The modified code is given in Figure 1.16.

Performance results are shown in Table 1.5 for a 4096 x 4096 pixel output image. Pulse block size is fixed at 100, and block size is fixed at 16 x 16 pixels, with the number of pixels per thread being varied from 2 to 4 in steps of 2.

These results show that increasing the number of pixels per thread and increasing occupancy significantly improves performance. For the case of two pixels per thread, each block used 6400 registers, permitting five blocks to be active at a time. When the number of pixels was increased to four, register usage increased to 8192, supporting only four active blocks. This resulted in a small performance decrease, as the smaller number of blocks available for scheduling per multiprocessor offset the decrease in communication cost.

1.5 Conclusion

Given a single Nvidia Tesla C2050 GPU, it is possible to obtain significant performance improvements over traditional CPU-based cache-aware implementations of Backprojection coded in C. A tiled partitioning scheme ensures locality of access for input and output data within each partition. This facilitates an efficient, straightforward parallelization of Backprojection, where each kernel is responsible for each pixel, and each thread block is responsible for each image tile. Such an implementation is capable of attaiting 182.4

```
__global__ void bploop(float2 *pulse_data, float2
   *image_data, float c4df, int num_pulses, int
   num_bins,int image_width,int image_height){
   float r0,f0, pixel0_Y, pixel0_X, pixel1_Y, w1, w2,
   tmpRF,tmpRG, f1, f2, binFloat, binFloor, pre2,
   pre3, tmpRA, pre1, R, Rstart;
   float2 tmpRC, tmpRB, pixel0, pixel1;
   int i,j,k,tmpRH;
   int2 tmpRI;
   //the pixel that corresponds to this thread
   j = (threadIdx.x+blockIdx.x*INTERNAL_BLOCK_SIZE);
   k = (threadIdx.y+blockIdx.y*INTERNAL_BLOCK_SIZE);
   //multiply the pixel y index number by 2
   k *= 2;
   //Get the current pixel value from global memory.
   //Necessary because we stream the pulse blocks to
   //the kernel.
   image_00.x=image_data[(int)(k+0+j*nyout)].x;
   image_00.y=image_data[(int)(k+0+j*nyout)].y;
   image_01.x=image_data[(int)(k+1+(j+0)*nyout)].x;
   image_01.y=image_data[(int)(k+1+(j+0)*nyout)].y;
   //read the pixels location
   tmpRI=tex1Dfetch(tex_X1,j+0);
   pixel0_X=(float) __hiloint2double(tmpRI.y,
      tmpRI.x);
   tmpRI=tex1Dfetch(tex_Y1,k+0);
   pixel0_Y=(float) __hiloint2double(tmpRI.y,
      tmpRI.x);
   tmpRI=tex1Dfetch(tex_Y1,k+1);
   pixel1_Y=(float) __hiloint2double(tmpRI.y,
      tmpRI.x);
   // precompute constants for use in the main loop
   pre1 = (num_bins-1)/(2.0*c4df);
   pre2 = 4.0f*pi/sol/2.0f;
   pre3 = c4df*num_bins/(num_bins-1.0f);
   // loop through all pulses for each subtile
   for (i=0;i<num_pulses;i++){
      //load r0
      r0=tex1Dfetch(tex_r0_data,i);
      Rstart=r0-pre3;
      //determine range to pulse location
      tmpRA = pixel0_X-tex1Dfetch(tex_x_obs,i);
      R = tmpRA*tmpRA;
      tmpRA = pixel0_Y-tex1Dfetch(tex_y_obs,i);
      R += tmpRA*tmpRA;
      tmpRA=tex1Dfetch(tex_z_obs,i);
      R += tmpRA*tmpRA;
      R=sqrt(R);
      //compute bin index
      binFloat = (R-Rstart)*pre1;
      binFloor=(int)binFloat;
      w2=binFloat-binFloor;
      w1=1.0f-w2;
      //verify the bin is in the subset of the pulse
      //data this thread can access
      if(binFloor+1 < num_bins && binFloor > 0){
         binFloor += i*EXTERNAL_PHD_CACHE;
         f0=tex1Dfetch(tex_f0_data,i);
         //read the bins
         tmpRB=pulse_data[(int)(binFloor)];
         tmpRC=pulse_data[(int)(binFloor+1.0f)];
         tmpRF=(float)tmpRB.x; //bin1.real
         tmpRG=(float)tmpRB.y; //bin1.imag
         tmpRD=(float)tmpRC.x; //bin2.real
         tmpRE=(float)tmpRC.y; //bin2.imag

         //compute f1, the non phase corrected sum
         tmpRD = w1*tmpRF + w2*tmpRD;
         tmpRE = w1*tmpRG + w2*tmpRE;
         //populate the phase corrector (f2)
         tmpRF = pre2*f0*((float)R);
         tmpRH = static_cast<int>(tmpRF /
            ((float)pi));
         tmpRG = tmpRF - static_cast<float>(tmpRH) *
            ((float)pi);
         tmpRF = __tanf((float)tmpRG);
         tmpRG = tmpRF*tmpRF + 1.0f;
         f2.x = (2.0f - tmpRG) / tmpRG;
         f2.y = (2.0f * tmpRF) / tmpRG;
         //sum phase corrected result to the image
         pixel0.x += (tmpRD*f2.x-tmpRE*f2.y);
         pixel0.y += (tmpRE*f2.x+tmpRD*f2.y);
      }

      /*** Begin Second Pixel Computation***/

      //determine range to pulse location
      tmpRA = pixel1_X-tex1Dfetch(tex_x_obs,i);
      R = tmpRA*tmpRA;
      tmpRA = pixel1_Y-tex1Dfetch(tex_y_obs,i);
      R += tmpRA*tmpRA;
      tmpRA=tex1Dfetch(tex_z_obs,i);
      R += tmpRA*tmpRA;
      R=sqrt(R);
      //compute bin index
      binFloat = (R-Rstart)*pre1;
      binFloor=(int)binFloat;
      w2=binFloat-binFloor;
      w1=1.0-w2;
      //verify the bin is in the subset of the pulse
      //data this thread can access
      if(binFloor+1 < num_bins && binFloor > 0){
         binFloor += i*EXTERNAL_PHD_CACHE;
         f0=tex1Dfetch(tex_f0,i);
         //read the bins
         tmpRB=pulse_data[(int)(binFloor)];
         tmpRC=pulse_data[(int)(binFloor+1.0f)];
         tmpRF=(float)tmpRB.x; //bin1.real
         tmpRG=(float)tmpRB.y; //bin1.imag
         tmpRD=(float)tmpRC.x; //bin2.real
         tmpRE=(float)tmpRC.y; //bin2.imag
         //compute f1, the non phase corrected sum
         tmpRD = w1*tmpRF + w2*tmpRD;
         tmpRE = w1*tmpRG + w2*tmpRE;
         //populate the phase corrector (f2)
         tmpRF = pre2*f0*(R);
         tmpRH = static_cast<int>(tmpRF / pi);
         tmpRG = tmpRF - static_cast<float>(tmpRH) *
            pi;
         tmpRF = __tanf(tmpRG);
         tmpRG = tmpRFfloat*tmpRF + 1.0f;
         f2.x = (2.0f - tmpRG) / tmpRG;
         f2.y = (2.0f * tmpRF) / tmpRG;
         //sum phase corrected result to the image
         pixel0.x += tmpRD*f2.x-tmpRE*f2.y;
         pixel0.y += tmpRE*f2.x+tmpRD*f2.y;
      }
      /*** End Second Pixel Computation***/
   }
   //write back to global memory for pixel
   image_data[(int)(k+0+j*image_height)].x
      =(float)pixel0.x;
   image_data[(int)(k+0+j*image_height)].y
      =(float)pixel0.y;
   image_data[(int)(k+1+j*image_height)].x
      =(float)pixel1.x;
   image_data[(int)(k+1+j*image_height)].y
      =(float)pixel1.y;
}
```


Figure 1.15: Tiled Backprojection Kernel with Two Pixels Per Thread, Kernel Code

```
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(image_width / threads.x, image_width / threads.y);
grid.y /= 2;
```

Figure 1.16: Tiled Backprojection Kernel with Multiple Pixels Per Thread, Host Code

Gflop/s. Additional performance is obtained by noting that the latency required to transfer pulse data from host to device can be largely overlapped with the processing of that data. This increases the throughput to 202.1 Gflop/s. Revisiting the parallelization scheme and increasing the number of pixels per thread permits the tile size to be increased without increasing the size of the thread block. This results in register usage that increases as a sublinear function of tile size, and additional active blocks to be scheduled on each streaming multiprocessor at a time. This occupancy increase the throughput to 221.3 Gflop/s.

1.6 Acknowledgments

This work was completed with funding provided by the following grants: NETS0963812 and NETS1115194.

References

[1] 2008. NVIDIA Introduction to CUDA.

[2] 2009. *NVIDIA Programming Guide - Version 2.2.*

[3] R.S. Ledley. 1976. Introduction to computerized tomography. *Comput. Biol. Med. 6*: 239-246.

[4] Alan Dl Cenzo. 1966. A Comparison of Resolution for Spotlight Synthetic-Aperture Radar and Computer-Aided Tomography. *Proceedings of the IEEE, VOL 74, NO. 6*: 1966

[5] Curtis H. Casteel and LeRoy A. Gorham Jr. and Michael J. Minardi and Steven M. Scarborough and Kiranmai D. Naidu and Uttam K. Majumder. A Challenge Problem for 2D/3D Imaging of Targets from a Volumetric Data Set in an Urban Environment. *Proc. of SPIE Vol. 6568 65680D-1*

[6] Lars M. H. Ulander and Hans Hellsten and Gunnar Stenstrom. 2003. Synthetic-Aperture Radar Processing Using Fast Factorized Back-Projection. *IEEE Transactions on Aerospace and Electronic Systems, Vol. 39 No. 3*: 760-776

[7] Mita D. Desai and W. Kenneth Jenkins. 1992. Convolution Backprojection Image Reconstruction for Spotlight Mode Synthetic Aperture Radar. *IEEE Transactions on Image Processing, Vol. 1 No. 4*: 505-517.

[8] Leonid Brekhovskikh. 2003. Fundamentals of ocean acoustics. *3rd. Springer Verlag*

[9] Shibdas Bandyopadhyay. 2009. SAR implementation on the GPU using CUDA.

[10] William Chapman, Sanjay Ranka, Sartaj Sahni, Mark Schmalz, "Parallel Processing Techniques for the Processing of Synthetic Aperture Radar Data on FPGAs." October 2009.

[11] William Chapman, Sanjay Ranka, Sartaj Sahni, Mark Schmalz, "Parallel Processing Techniques for the Processing of Synthetic Aperture Radar Data on FPGAs." October 2009.

[12] William Chapman, Sanjay Ranka, Sartaj Sahni et. all, "Parallel Processing Techniques for the Processing of Synthetic Aperture Radar Data on GPUs." December 2011.