3D Recursive Gaussian IIR on GPUs and FPGAs

A Case Study for Accelerating Bandwidth-Bounded Applications

Jason Cong, Muhuan Huang and Yi Zou
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA

***Abstract*—GPU devices typically have a higher off-chip bandwidth than FPGA-based systems. Thus typically GPU should perform better for bandwidth-bounded massive parallel applications. In this paper we present our implementations of a 3D recursive Gaussian IIR on multi-core CPU, many-core GPU and multi-FPGA platforms. Our baseline implementation on the CPU features the smallest arithmetic computation (2 MADDs per dimension). Since this application is clearly bandwidth bounded, we show that the difference on the memory subsystems on different platform requires different bandwidth optimization techniques. Our implementations on the GPU and FPGA platforms show a 26X and 33X speedup respectively over the optimized single-thread code on the CPU.**

I. INTRODUCTION

Nowadays, normal desktop or server CPU (e.g., those using FSB protocols) has an off-chip memory bandwidth below or around 10GB/s. GPU device typically has a very high off-chip memory bandwidth. For example, the Nvidia Telsa compute cards C1060 has 102GB/s peak off-chip bandwidth. The off-chip bandwidth of FPGA-based systems, is often lower. For example, the SDRAM bandwidth in the Xilinx XUPV5 board is less than 1GB/s. The aggregate off-chip memory bandwidth of the 4-FPGA system we use (Convey HC-1) is around 80GB/s. In the medical image processing, the three-dimensional images we process have a large memory footprint that can not be stored on-chip. The ratio between computation and data access is very small. We would normally expect the GPU to perform better simply because of its better bandwidth. However, this is not always the case. This paper presents a case where the multi-FPGA system with a lower bandwidth, delivers a better performance for a bandwidth-bounded application. GPU computation model requires a large amount of threads to obtain good utilization of the hardware while hiding the latency. When each thread needs to maintain some on-chip working-set (in registers or scratchpads) to capture data reuse, we must reduce the number of active parallel threads to resolve the fitting issue. The latency in the data access and computation can not be hidden when the number of threads is low. FPGA-based computation does not require a very large number of parallel threads as it also explores deep pipelining in addition to data parallelism. This translates to a better balance between data reuse and data-parallel computation in this application case.

Our case study to accelerate a bandwidth-bounded application is Gaussian smoothing. Gaussian smoothing, which convolves an image with a Gaussian function, is an important image processing step to blur the image or reduce the noise. Gaussian smoothing also serves as the critical computation kernel in various other numerical algorithms in computer vision and medical imaging. The Demons algorithm [1], a popular medical image registration algorithm, uses Gaussian smoothing to smooth the deformation field. Fluid registration [2], [3], a class of image registration algorithms that better support large deformations, uses Gaussian smoothing to smooth the velocity field.

Gaussian smoothing essentially performs

$$\widetilde{A} = G_\sigma \otimes A \tag{1}$$

```
// Smoothing over x direction (stride 1 direction)
for( i = 0; i < m; i++)
  for( j = 0; j < n; j++)
  {
    for( k = 0; k < p; k++)
      u_tmp[k] = u[i][j][k];
    GaussianSmoothing1D(u_tmp);
    for( k = 0; k < p; k++)
      u[i][j][k] = u_tmp[k];
  }
// Smoothing over y direction (stride p direction)
    ...
// Smoothing over z direction (stride p*n direction)
    ...
//Scaling & Normalization
for( i = 0; i < m; i++)
  for( j = 0; j < n; j++)
    for( k = 0; k < p; k++)
      u[i][j][k]* =  c3;
```

Fig. 1: 3D Gaussian IIR Computation

where A is the input image, G_σ is a Gaussian function and $\otimes$ is the convolution operator.

There are many algorithms and implementations of Gaussian smoothing. Roughly these can be divided into three categories: FFT-based convolution, direct convolution, and recursive impulse infinite response (IIR) filtering. The FFT-based method operates on the frequency domain. The complexity of the FFT-based convolution is $O(NlogM)$, where N is the number of pixels/voxels and M is the radius of the convolution kernel. Direct convolution performs weighted sum operation for each output pixel/voxel. The complexity is $O(NM)$ because the number of operations per output pixel in the direct convolution method grows proportionally with the kernel size. The computation involved in the recursive IIR does not depend on the size of Gaussian kernels. The complexity is $O(N)$.

In this paper, the baseline implementation is a Gaussian recursive IIR proposed by Alvarez and Mazorra in [4]. They expliot the relationship between Gaussian smoothing and a heat equation PDE to provide a very fast smoothing algorithm. We try to use the Gaussian smoothing operation inside a fluid registration application which requires the use of a large σ (standard deviation of the Gaussian distribution). Gaussian IIR is the method among the three categories that features the smallest computation.

II. REVIEW OF THE ALGORITHM

The Gaussian kernel processes each 3D voxel in a separable manner, where the filter kernel can be applied in three dimensions separately. The computation in each dimension performs in a forward and backward fashion, and thus we need to conduct sweeping in six directions.

For an 1D case, each sweeping for a typical IIR works like

$$y(n) \approx a * x(n) + b * y(n-1) \tag{2}$$

where $y(n)$ denotes the new signal sequence that is generated by the IIR, and $x(n)$ means the input signal sequence.

```
//Backward:
u_tmp[0] = u_tmp[0] * c1;
for(i = 1; i < N-1; i++)
 u_tmp[i] = u_tmp[i] + u_tmp[i-1] * c2;
//Forward:
u_tmp[N-1] = u_tmp[N-1] * c1;
for(i = N-2; i >= 0; i--)
u_tmp[i] = u_tmp[i] + u_tmp[i+1] * c2;
```

Fig. 2: 1D Gaussian IIR

Figure 1 shows the code for the 3D Gaussian smoothing process. The size of the image is $m \times n \times p$. The algorithm contains two steps. In the first step, we perform sweeping for all six directions (or for three dimensions x, y and z respectively). In the second step, image value is simply scaled by a constant. Because the final normalization step is typically fused with subsequent computation steps in the image processing pipeline, we ignore the normalization step in our run-time measurements.

We illustrate 1D Gaussian smoothing in Figure 2. In backward computation, a voxel value with a larger index depends on the voxel value with a smaller index; while in the forward computation, the converse is true, i.e., a voxel value with a smaller index depends on the voxel value with a larger index.

Coefficients $c1$, $c2$ and $c3$ in Figures 1 and 2, are derived based on the σ of the Gaussian filter. We have $a = 1$ and $b = c2$ for Eq. 2. $c1$ is used for setting up the boundary condition, and $c3$ is used for scaling. The values are calculated beforehand. More details can be seen in [4].

III. Recursive Gaussian IIR on CPUs

A. Reducing The Bandwidth Pressure

For a reasonably sized image coming from the medical imaging domain (e.g., like 256^3), a 2D slice of the 3D image can fit the working set of L2. We perform loop merging for four out of six directions (or equivalently, we first perform a series of 2D Gaussian smoothing in a sequence, then work on the remaining dimension). In this way, the pressure on bandwidth is reduced, because the revised procedure then only need to read and write the 3D image object twice. Note that a direct implementation following Figures 1 and 2 shall read and write the 3D image object three times.

B. Parallel Implementation on Multi-Core CPUs

We start from the code that we optimized in the preceding subsection, and add OpenMP pragmas at the outer-most loops. Effectively, we are parallelizing different 2D Gaussian smoothing, and also parallelizing groups of 1D Gaussian smoothing (for the remaining dimension).

IV. Recursive Gaussian IIR on GPUs

A. Reuse 1D Working Set Or Not

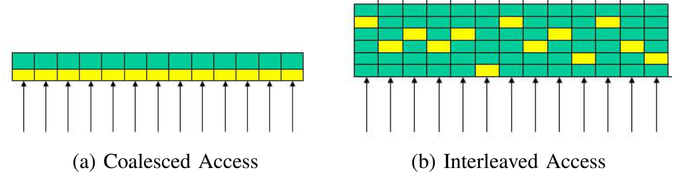
Reference [5] claims that recursive Gaussian performs poorly on GPU for 2D image smoothing as it only has one dimension of parallelism. For a 3D image, the situation is much better because we have two parallel dimensions. We design the parallel algorithm so that each thread is responsible for one 1D Gaussian IIR. However, if each thread keeps the temporary working set (a row of 256 elements for a 3D image of size 256^3) in the on-chip shared memory, it will limit the possible parallelism we can get. For a 3D image of size 256^3 in a single precision format, the maximum parallelism we can get in one stream multiprocessor(SM) is limited to 16 (16KB/256/4). (We are using a previous-generation Nvidia Telsa C1060 with 16KB on-chip shared memory in each SM.) The CUDA programming model

```
Step 1: Transpose:
  Read data from off-chip to scratchpad
  Transpose in scratchpad
  Write transposed data to off-chip
Step 2: Computation:
  Read data from off-chip
  Perform IIR
  Write data to off-chip
Step 3: Transpose:
  Read data from off-chip to scratchpad
  Transpose in scratchpad
  Write transposed data to off-chip
```

Fig. 3: Computation With Transpose

```
  Read data from off-chip to scratchpad
  Perform IIR on the scratchpad data
  Write scratchpad data to off-chip
```

Fig. 4: Computation Without Transpose



(a) Coalesced Access (b) Interleaved Access

Fig. 5: Coalesced Access vs. Interleaved Access

needs many more threads to fill up the pipeline and hide the latency. If we do not exploit this data reuse in 1D IIR, we can execute more parallel threads, although we shall end up using 2X bandwidth than that with data reuse.

We tested the approach that exploits this data reuse and the approach that does not. In the platform that we use (Telsa C1060), the approach that does not reuse the data actually runs faster. The situation may change in the latest Nvidia hardware because the Fermi GPU dedicates a 4X larger on-chip scratchpad memory.

B. Eliminating Transpose Operations

Recall that we need to do a sweeping of the 3D array in all six directions. For four directions out of six, we can get coalesced data access very easily because the coalescing direction is perpendicular to the dependence direction. For the other two directions, the coalescing direction is parallel with the dependence direction. One typical implementation is to transpose the image, then apply the IIR on a transposed dimension, and then transpose it back. Each transpose step needs an additional sweeping of the 3D image. Our implementation does not perform explicit transpose. On each stream multiprocessor, we first copy a 2D tile into on-chip memory in a coalesced fashion; we then perform computation on the tile and then write data back in a coalesced fashion. Overall, our optimized CUDA implementation of Gaussian IIR needs to read and write a 3D object six times. Figure 3 shows the pseudo code of the implementation with transpose, and Figure 4 shows the pseudo code of the implementation that does not use transpose. The approach in Figure 4 is better, because it has fewer external memory accesses compared to that in Figure 3.

Coalescing is one of the essential techniques we should use to improve performance. However, in our case, the parallel data access that is required for the core computation may come with a large stride when the dependent direction is parallel with the coalescing direction. Because of this, we use the scratchpad as the intermediate buffer to make the coalesced data access easier. The scratchpad is not used to capture reuse due to concerns discussed in previous subsection.

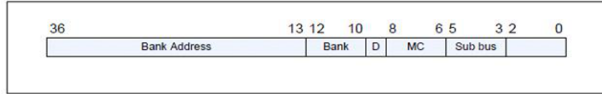


Fig. 6: Binary Interleave

V. RECURSIVE GAUSSIAN IIR ON FPGAS

The multi-FPGA platform Convey HC-1 uses an interleaved memory scheme. Different FPGAs access the off-chip memory using a shared memory model. The system employs an on-board crossbar to realize the interconnection. At a high-level, the interleaved memory scheme can support the parallel data access shown in Figure 5b. The particular interleave scheme is called binary interleave. The binary interleave scheme takes out the 3rd to 12th bits (counted from the LSB) of the virtual address to determine the bank the address falls in. The system features 1024 banks.

The HC-1 platform has four user FPGAs. Each FPGA is presented with 16 external memory access ports.

A. Overview of the Design

For the FPGA implementation, we use the scheme described in Figures 1 and 2 and exploit the 1D reuse. We need to read and write the 3D object three times.

Each processing element (PE) is a dedicated hardware unit that computes 1D IIR. Each PE is composed of two components. The first component is the address generator hardware unit that keeps sending memory access requests into one memory access port. In parallel with the address generator, another component–the main task-level pipeline includes the backward processing unit and forward processing unit. The backward processing unit reads memory responses, performs computation, and writes data into the on-chip BRAM (ping-pong buffer). The forward processing unit reads data from the ping-pong buffer, and sends requests to another memory access port. The backward processing unit and the forward processing unit work in parallel. The core computation component in the backward processing unit and the forward processing unit is a floating point MADD operator. The components contain some additional control logics for data streaming and loop pipelining. Note that there is a strong data dependency within the backward processing and forward processing, and the floating point operation typically has a multi-cycle latency. If we simply pipeline the loop in the dependence direction, we can not achieve a one-cycle pipeline initial interval (II). So we instead compute a group of 1D IIR in an interleaved fashion.

We realize eight PEs on each FPGA. Our multi-FPGA design incorporates 32 PEs and utilizes all the memory access ports available in the platform.

B. Distributing Memory Requests

Binary interleave as shown in Figure 6 is used for our custom FPGA design implementation. We can quickly see that this binary interleave scheme may result in bank conflicts if parallel access uses a stride that is a power of two. A simple remedy is to enlarge the array size by 1. E.g., we can enlarge an image size of 256^3 to $256 * 257 * 257$. In this manner, we resolve the bank conflicts in a way similar to prime number interleave. For all the three dimensions, we can get an interleaved access with no bank conflict. This simplifies our hardware design, because we can design a single piece of hardware to support the sweeping for all six directions.

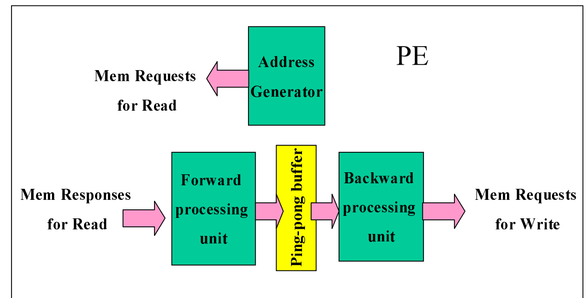


Fig. 7: One PE in Our FPGA Design

TABLE I: Performance on Multi-Core

Image	1thread	2threads	4threads	8 threads
$256 * 256 * 64$	0.179	0.091	0.046	0.027
$256 * 256 * 128$	0.359	0.181	0.092	0.052
$256 * 256 * 256$	0.719	0.363	0.186	0.104

C. Packing/Unpacking for Wider Data Access

The hardware platform supports memory requests for 8-bit, 16-bit, 32-bit and 64-bit accesses. It does not support the cache-line transfers that typical systems do. Our computation uses the single-precision floating point. For two out of three dimensions, we rearrange (loop interchange and loop tiling) the loop structure in a way that enables packing and unpacking to utilize the 64-bit data channels. This means that we use a single request to get two 32-bit contiguous data. For the remaining dimension, we still use a 32-bit data channel and rely on two requests to read/write data. To conserve area and build a static pipeline, we set the II of the backward processing unit and forward processing unit to 2. The backward processing unit and forward processing unit are now capable of processing two new 32-bit data every two cycles. (Effectively, the computation throughput is unchanged, but the external bandwidth is improved as we generate fewer requests). To do alignment for 64-bit access correctly, we change the padding scheme from $256 * 257 * 257$ to $256 * 257 * 258$.

VI. EXPERIMENT RESULTS

The baseline code is written in C. We compile the code using gcc -O3. We parallelize the code using OpenMP. The hardware platform is a dual socket system that has two Intel Xeon E5405 CPUs and 8GB system memory. Each socket is a quad-core with 12MB L2. Our CUDA code is compiled using CUDA toolkit 3.2 and tested using Nvidia Telsa C1060 which has 30 stream multiprocessors or 240 stream processors. Our FPGA platform is the Convey HC-1 equipped with four Xilinx Virtex5 XC5LX330 FPGAs. Our FPGA design is first described using C code which is heavily restructured and optimized for hardware synthesis. Verilog RTL is further obtained using the high-level synthesis tool AutoPilot version 2010.a.3. ISE 11.5 is used to obtain the final configuration bitstream. Four FPGAs share the same bitstream, but work on different data. Note that the hardware components that we compare are not the most recent, but they have been in marketplace for roughly the same number of years.

We test three data-sets, with size $256*256*64$, $256*256*128$ and 256^3. Our code for the CPU and GPU platforms can support images with a larger size as well. However, because we allocate a fixed size for the ping-pong buffer used in our FPGA implementation, currently the FPGA implementation can only support images up to 256^3.

A. Performance on Multi-Core

Table I shows the performance of parallel OpenMP code on an Intel multi-core platform. We see that the program scales to 4-core, but

TABLE II: Performance of GPU and FPGA Implementation

Image	GPU	Speedup	FPGA	Speedup
$256*256*64$	0.0070	25.6X	0.0055	32.5X
$256*256*128$	0.0139	25.8X	0.0109	32.9X
$256*256*256$	0.027	26.6X	0.0216	33.3X

TABLE III: Power and EDP comparison for 3D Gaussian IIR

	CPU	GPU	FPGA
Power	2*80W	200W	4*23W
Energy	8.0X	2.8X	1X
EDP	40.3X	3.4X	1X

saturates at 8-core as the program gets bandwidth bounded. Measured execution times are all in seconds.

B. Performance of GPU and FPGA Implementations

Table II shows the performance of our GPU implementation and FPGA implementation. Speedup shown in the table is compared against a single-threaded version. The GPU can deliver 26X speedup while our multi-FPGA implementation can deliver 33X speedup. Compared to a parallel version that runs on a two-socket (8-core) server configuration, the speedup of the GPU implementation is around 4X, and our multi-FPGA implementation delivers around 5X speedup. Note that CPU, GPU and FPGA implementations all use a single-precision floating point, and the correctness of our implementation is fully validated.

We investigated the reason why the FPGA design is faster, and why the GPU implementation can not use a similar scheme to obtain a better performance. The peak external bandwidth of our GPU is around 100GB/s, and the peak external bandwidth of our multi-FPGA system is 80GB/s. We get around 30% efficiency of the external memory system on either platform. In our GPU implementation, we read/write the 3D array six times, which generates 2X traffic compared to the FPGA implementation. So why can't we use an implementation that only reads/writes 3D array three times in a GPU? Our FPGA implementation separates address generation logic from the main computing pipeline. This can effectively prefetch the required data so that data access and computation are overlapped. We also have a pipeline parallelism between the forward processing and backward processing. To achieve a similar effect and hide the data access latency and computation latency, CUDA requires a large number of threads. External data access takes several hundred cycles. Suppose we use only 16 threads to read/write data in the off-chip memory; we will always need to wait for this long latency. To alleviate this issue, we can use more threads to read and write the data. An improved version first reads the data into the on-chip shared memory using more threads, then starts computation, and then writes data back using more threads. This improves the performance; however, the computation is not overlapped with data access because we need explicit synchronization of threads to guarantee the proper memory ordering. Also in the computation part, we still can not the hide the latency of flow-dependency (read after write) because the available parallelism is restricted. The latency of a RAW dependency is 24 cycles according to CUDA programming manuals.

C. Power, Energy and EDP Comparison

The TDP (thermal design power) of the Intel Xeon CPU is 80W. The TDP of the Tesla C1060 is 200W. We run the *xpa* tool from Xilinx to estimate the power consumption of our FPGA design. Table III compares the power consumption, energy consumption and the energy-delay product (EDP) of the three platforms. Because TDP is the maximum power allowed, we use the full-loaded (8-thread) execution time to compute the energy and EDP of the CPU platform. For this application, the energy of the multi-socket CPU platform, and the GPU implementation is 8X, 2.8X of the multi-FPGA platform. The EDP of the multi-socket CPU platform and GPU implementation is 40X, 3.4X of the multi-FPGA platform respectively.

VII. Related Work

A separable 2D convolution based on Deriche's algorithm [6] is part of the CUDA SDK. A 3D convolution based on that algorithm should be straightforward to develop. The implementation uses explicit transpose to simplify the code reuse.

There are various ways to implement Gaussian filters. A CUDA implementation of the Demons algorithm is presented in [7]. In their implementation, Gaussian smoothing uses the separable convolution method, where three convolutions for three dimensions are performed in a sequential fashion. We need to use a large sigma ($\sigma = 4.5$) in our target application fluid registration; thus we prefer a recursive approach.

An FPGA-based anisotropic diffusion filtering filter for a 3D image is presented in [8]. The core computation step used in anisotropic diffusion filtering is Gaussian smoothing. However, their approach can only support a convolution radius of up to 4, and does not work in our setting, because our fluid registration application requires a much larger σ where direct convolution is not feasible to implement on FPGAs. We are not aware of any prior work that implements a 3D Gaussian IIR on FPGA hardware.

VIII. Conclusions and Future Work

This paper presents the GPU and FPGA implementation of 3D recursive Gaussian smoothing. We show that our multi-FPGA design is around 30% faster than the GPU implementation using less than half of the power consumption. This illustrates a need for customizable computing, even for the case of bandwidth-bounded application where GPUs are considered well-suited for.

Currently we are working to integrate the implementation of 3D Gaussian filter into an FPGA-based fluid registration design.

IX. Acknowledgements

This work is partially funded by the Center for Domain-Specific Computing (NSF Expedition in Computing Award CCF-0926127), and grants from Nvidia Corp. and Mentor Graphics Corp. under the UC Discovery program.

References

[1] J.-P. Thirion, "Non-rigid matching using Demons," in *Proc. CVPR*, Jun. 1996, pp. 245 –251.

[2] E. D'Agostino, F. Maes, D. Vandermeulen, and P. Suetens, "A viscous fluid model for multimodal non-rigid image registration using mutual information," in *Proc. MICCAI*, 2002, pp. 541–548.

[3] I. Yanovsky, A. D. Leow, S. Lee, S. J. Osher, and P. M. Thompson, "Comparing registration methods for mapping brain change using tensor-based morphometry," *Medical Image Analysis*, vol. 13, no. 5, pp. 679–700, October 2009.

[4] L. Alvarez and L. Mazorra, "Signal and image restoration using shock filters and anisotropic diffusion," *SIAM J. Numer. Anal.*, vol. 31, pp. 590–605, April 1994.

[5] X. Wang and B. Shi, "GPU implemention of fast gabor filters," in *Proc. ISCAS*, 30 2010.

[6] R. Deriche, "Fast algorithms for low-level vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, pp. 78–87, January 1990.

[7] P. Muyan-Ozcelik, J. Owens, J. Xia, and S. Samant, "Fast deformable registration on the GPU: A CUDA implementation of demons," in *Proc. ICCSA*, 30 2008.

[8] O. Dandekar, C. Castro-Pareja, and R. Shekhar, "FPGA-based real-time 3D image preprocessing for image-guided medical interventions," *Journal of Real-Time Image Processing*, vol. 1, pp. 285–301, 2007.