Auto-optimization of a Feature Selection Algorithm

Didem Unat[1], Han Suk Kim[1], Jürgen P. Schulze[2], Scott B. Baden[1]
[1]Department of Computer Science and Engineering
[2]California Institute for Telecommunications and Information Technology
University of California San Diego,
La Jolla, California 92093, USA
Email: {dunat, hskim, jschulze, baden}@ucsd.edu

***Abstract*—Advanced visualization algorithms are typically computationally expensive but highly data parallel which make them attractive candidates for GPU architectures. However, porting algorithms on a GPU still remains a challenging process. The Mint programming model addresses this issue with its simple and high level interface. It targets the users who seek real-time performance without investing in significant programming effort. In this work, we present automatic CUDA parallelization and optimizations of the Harris interest point detection algorithm with Mint. Mint generates highly optimized CUDA C from annotated C source and performs several optimizations. For 4 well-known datasets in volume rendering, on Tesla C1060 the Mint-generated kernels run under a second and deliver on average 10 times the performance of OpenMP running with 4 threads on a Nehalem processor.**

I. Introduction

One of the challenges in computer visualization is how to render data in real-time. Interactive rendering requires at least 25 frames per second and sporadic long computations should not exceed one second; otherwise users can sense the discontinuity in the interaction. On the other hand, many advanced visualization techniques employ more complex algorithms, with the goal of finding meaningful information in the 3D datasets. However, advanced algorithms are often computationally expensive and the number of voxels in 3D data further raises the computational cost. Graphical Processing Units (GPUs) are an attractive means for accelerating such computations. They have been employed successfully in data parallel applications and are commonly used to accelerate rendering algorithms.

However, an outstanding difficulty is the steep learning curve encountered when programming GPU architectures and in the complexity of the required performance programming techniques. A directive-based programming model for GPU programming is a promising approach to address this issue because the user can avoid learning low level programming by expressing the parallelism at a high level. In this paper, we describe our experience with a directive-based approach to GPU programming, called Mint [1]. Mint is simple, compact, and comes with a source-to-source translator that generates optimized CUDA C from traditional C source.

The Mint optimizer targets stencil methods, an important problem domain with a wide range of applications. Stencil methods arise in finite different solvers and image processing. In previous work [1], Unat et al. introduced Mint and demonstrated its effectiveness in treating a variety of commonly used finite difference kernels. We demonstrate the applicability of Mint to another field: computer visualization. We consider the Harris interest point detection algorithm [2], which extracts information about the visually interesting features in 3D volume datasets. The algorithm computes a local feature with convolution and selects corners of an object and areas with high intensity as "interest points".

The Harris point detection algorithm is representative of wide range of computer vision algorithms, including pointwise and convolution computations [3], [4]. These algorithms constitute the cores of many important applications such as object recognition and motion detection. Moreover, feature selection algorithms are commonly applied in computer visualization to control colors and opacities of objects in the data [5], [6], [7], [8]. Thus, in demonstrating our approach for the Harris algorithm, we broaden the scope of applications to which the Mint programming model applies.

In this paper, we demonstrate the effectiveness of the Mint translator both in terms of programmer's productivity and performance of the generated-code. We discuss the details of the optimization steps carried out automatically by the translator. Mint greatly improves the productivity of the programmer since the programmer does not need to learn CUDA or GPU architecture. Secondly, Mint delivers much higher performance than OpenMP which requires almost the same level of programming effort. For 4 different 3D volume datasets running on a Tesla C1060, we demonstrate that the translator delivers 6.8 to 11.9 times the performance of OpenMP running with 4 threads on a quad-core Nehalem processor. On a Fermi device, Tesla C2050, the average speedup reaches 23x over OpenMP.

The remainder of this paper is organized as follows: Sec-II provides an overview of the Mint programming model. Sec-III gives background on the feature selection algorithm and Sec-IV describes the Mint implementation. Sec-V discusses the translation and optimization steps of the feature selection algorithm. Sec-VI presents performance results and an evaluation, and the last section presents related work and a summary.

II. Mint Overview

Simplicity has been our principal design goal for the Mint model. Mint employs just five directives: 1) `parallel` to indicate the start of an accelerated region, 2) `for` to mark the succeeding nested loops for GPU acceleration, 3) `copy` to express data transfers between the host and device, 4)

`barrier` and 5) `single` to synchronize and handle serial regions.

The indispensable directive of the model is the `for` directive since it indicates the most time consuming part of the code, that benefits most from acceleration. If used with the `nest` clause, Mint will create multi-dimensional kernels to parallelize the specified loop nest. This capability of Mint is crucial; it enables the user to employ higher dimensional CUDA thread blocks which are required to use the device effectively. We introduce two additional clauses to support the `for` directive for locality and thread workload management. The `tile` clause specifies how the iteration space of a loop nest is to be subdivided into *tiles*. In the CUDA context, a tile corresponds to the number of data points computed by a thread block. Another important clause is `chunksize`. It allows the programmer to easily manage the mapping of workload to threads and create "tiny" or "fat" threads. This clause is particularly helpful when combined with on-chip memory optimizations because it enables re-use of data. Fig 1 depicts how the tile and chunksize clauses are used to establish the size of a CUDA thread block. These clauses allow non-experts to incrementally tune the code for GPU acceleration without explicitly implementing each variation.

.

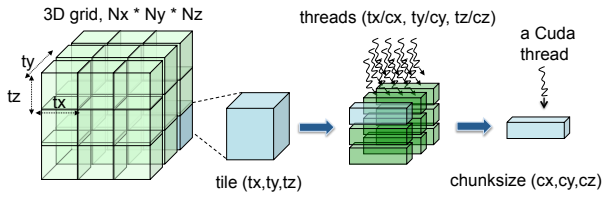


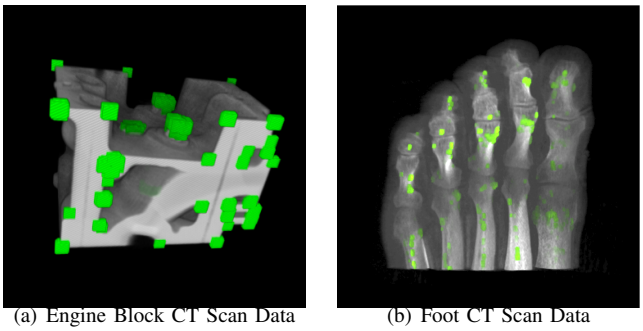
Fig. 1. Tiling and chunking of a 3D grid

Lastly, Mint helps the user manage the separate memory spaces. However, the Mint programmer specifies transfers at a high level and does not need to worry about the storage management. More information about the Mint model can be found in [1].

III. Feature Selection Algorithm

Feature selection is a very important step in many vision algorithms. Features are points of interest and interesting information about the image resides on the feature points; corners or edges of an object. In order to apply advanced algorithms, such as object recognition or motion detection, it is essential to have a good feature detection algorithm.

On the other hand, in volume visualization, where data is stored in a 3D structured grid, feature detection algorithms can provide useful information for advanced image processing, such as transfer function manipulation [6], [7], [8]. We introduce an extension of the Harris interest point detection algorithm to 3D datasets, as a representative example for feature selection algorithms for volume datasets.



(a) Engine Block CT Scan Data (b) Foot CT Scan Data

Fig. 2. Feature selection for the Engine Block CT Scan from General Electric, USA and the Foot CT Scan from Philips Research, Hamburg, Germany. Identified as corners by the Harris corner detection algorithm are Green squares at the corners of the engine block and around the joints in the foot image.

A. Background

The Harris interest point detection algorithm [2] extracts a set of features from an image by scoring the importance level of each pixel in the image. Positive score values correspond to interest points. A pixel is selected as a point of interest if there is a large change both in the X-axis and Y-axis. For example, corners of an object or high intensity points get a large positive score. On the contrary, the algorithm assigns a score close to zero to homogenous regions and large negative scores to edges.

The basic idea of computing the score is to measure the change E around a voxel (x, y, z):

$$
\begin{aligned}
& E(x,y,z) \\
& = \Sigma_{u,v,w} g(u,v,w) |I(x+u, y+v, z+w) - I(x,y,z)|^2 \\
& = \Sigma_{u,v,w} g(u,v,w) |xI_x + yI_y + zI_z|^2 \\
& = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} g \otimes I_x^2 & g \otimes I_x I_y & g \otimes I_x I_z \\ g \otimes I_x I_y & g \otimes I_y^2 & g \otimes I_y I_z \\ g \otimes I_x I_z & g \otimes I_y I_z & g \otimes I_z^2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \\
& = \begin{bmatrix} x & y & z \end{bmatrix} M(x,y,z) \begin{bmatrix} x & y & z \end{bmatrix}^T
\end{aligned} \tag{1}
$$

where g is a Gaussian convolution kernel. This equation says that the matrix M in Eq. 1 defines the changes at point (x, y, z), and the three eigenvalues of this matrix characterize the changes on each principal axis. Namely, if the change is large on all three axes, i.e., all three eigenvalues are large, the point is classified as a corner. On the other hand, for those points with a small change, i.e., homogeneous areas, M has small eigenvalues. If M has one or two large eigenvalues, it indicates that area is an edge.

The biggest challenge to employ this type of algorithm in practice, is its high cost because it requires convolution operation for every data point (voxel). Real-time rendering performance, e.g. 30 frames per second or more, is critical in visualization for the sake of interactivity. As we will discuss in Sec. VI, without the help of many-core architectures, it is impossible to compute this algorithm in real-time.

Fig. 2 shows results of the 3D Harris interest point detection algorithm. Fig. 2(a) shows an engine block; we highlighted the features the algorithm identified with green squares. All the corners and the two cylinders on the top were detected as features. Fig. 2(b) shows a foot CT scan data, which does not have distinct corners. However, the algorithm successfully identifies the tips and joints of the toes as features.

IV. Mint Implementation

The Mint program that implements the main loop of the 3D Harris interest point detection algorithm appears in Listing 1. (For the sake of clarity, we omit some details). Note that this program can also be compiled by a standard C compiler because the compiler will simply ignore the pragmas. Hence, the OpenMP implementation of the algorithm can be obtained by using simple string substitution.

The code in Listing 1 computes the convolution and obtains a Harris score for each voxel. In this example, the window size of the Gaussian convolution is set to 5: we compute the convolution as the weighted sum of a 5x5x5 patch around a point (x, y, z). Line 1 copies the *data* voxel array from the host memory to device memory. The last two arguments of the `copy` directive indicate the dimensions of the image. Line 2 copies the weight vector to the device. Line 4 indicates the start of the accelerated region. This program accelerates only one nested loop. However, as in OpenMP, a parallel region in Mint can have several parallel `for` loops. The Mint `for` directive in line 7 enables the translator to parallelize the loop-nest on lines 8 through 48. The `nest(3)` clause specifies that the 3 outmost loops can run in parallel. The 3 inner loops sweep the 5x5x5 window and will not be parallelized. The translator subdivides the input grid into 3D tiles for locality, as shown in Fig. 1. The `chunksize` clause specifies the workload of a thread. In this program, a thread is assigned to 64 iterations in the outmost loop. The for-loop clauses are optional. In the absence of these clauses, the compiler will choose default values. Lastly, line 50 copies the Harris scores back to the host memory.

V. Translation and Optimization

We have developed a fully automated translation and optimization system for the Mint programming model. The translator is built on top of the ROSE compiler framework [9] developed at Lawrence Livermore National Laboratory.

A. Mint Baseline Translator

The input to the Mint translator is C source code annotated with Mint pragmas. We refer to the translation without any optimizations as the baseline translation. The baseline translator creates CUDA kernels by outlining the candidate parallel for-loops in the parallel region. In the example program provided in Listing 1, Mint transforms lines 8-48 into a kernel. The translator moves the body of the loop into a newly-generated CUDA kernel and replaces the removed statements with a kernel launch. Variables that need to be passed become arguments to the kernel call. The baseline translator also inserts the code computing thread IDs into the newly-created kernel and replaces the original for-loop indices with the thread IDs. Mint uses these IDs to assign work to CUDA threads.

```
1  #pragma mint copy(data, toDevice, width, height, depth)
2  #pragma mint copy(w, toDevice, 5, 5, 5)
3
4  #pragma mint parallel default(shared)
5  {
6   // main loop
7  #pragma mint for nest(3) tile(16,16,64) chunksize(1,1,64)
8   for (i = 3; i < depth - 3; ++i) {
9    for (j = 3; j < height - 3; ++j) {
10    for (k = 3; k < width - 3; ++k) {
11
12     float Lx = 0.0f;
13     float Ly = 0.0f;
14     float Lz = 0.0f;
15     float LxLy = 0.0f;
16     float LyLz = 0.0f;
17     float LzLx = 0.0f;
18
19     for (l = i - 2; l <= i + 2; ++l) {
20      for (m = j - 2; m <= j + 2; ++m) {
21       for (n = k - 2; n <= k + 2; ++n) {
22
23        // gradient in x direction
24        float dx = (data[l][m][n+1] - data[l][m][n-1]);
25        // gradient in y direction
26        float dy = (data[l][m+1][n] - data[l][m-1][n]);
27        // gradient in z direction
28        float dz = (data[l+1][m][n] - data[l-1][m][n]);
29
30        const float weight=w[l-i+2][m-j+2][n-k+2];
31
32        // gaussian convolution sum
33        Lx += weight * dx * dx;
34        Ly += weight * dy * dy;
35        Lz += weight * dz * dz;
36        LxLy += weight * dx * dy;
37        LyLz += weight * dy * dz;
38        LzLx += weight * dz * dx;
39       }
40      }
41     }
42     // compute cornerness metric, Harris Scores
43     harrisScores[i][j][k] = ...//computing determinant
44             -sensitivity_factor*(Lx + Ly + Lz)*
45              (Lx + Ly + Lz)*(Lx + Ly + Lz);
46    }
47   }
48  }
49 }//end of parallel region
50 #pragma mint copy(harrisScores,fromDevice,width,height,depth)
51 ...
```

Listing 1. shows a part of Mint program for the 3D Harris interest point detection algorithm.

B. Preprocessing and Stencil Analyzer

The output of the baseline translator is CUDA code that makes all memory references through device memory. When Mint optimizations are not enabled, the translated code does not take advantage of on-chip memory: it only uses device memory. In order to optimize for on-chip memory, Mint analyzes the structure of the stencils and their neighboring relations because the shape of the stencil affects ghost cell loads (halos) and the amount of shared memory needed. This process would be very cumbersome if managed explicitly by the programmer. Our analyzer automatically derives this information from the array access patterns and passes it to the optimizer. The optimizer then applies various on-chip memory optimizations by using both shared memory and registers.

The analyzer first determines the stencil coverage by examining the pattern of array subscripts with a small offset

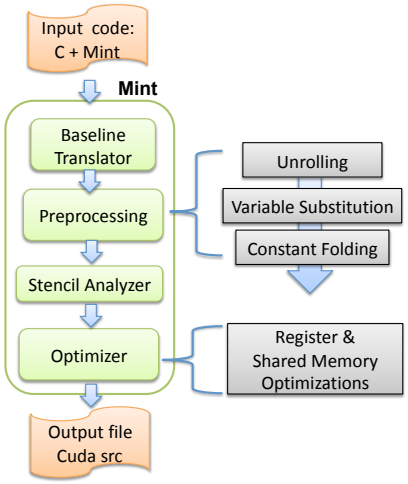


Fig. 3. Compilation Flow

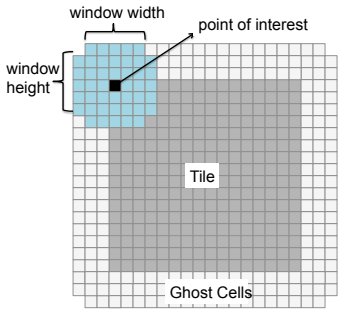


Fig. 4. A tile and its respective ghost cells in shared memory. The black point is the point of interest. The blue region around the black shows the coverage of the Gaussian convolution.

from the central point. That is, the index expressions are of the form $i \pm c$, where i is an index variable and c is a small constant. However, in Listing 1, the array indices are relative to the Gaussian convolution loops (lines 19-21). The indices are not a direct offset of the main loops (lines 8-10). This makes it difficult for a compiler to analyze the nearest neighbor relations between the points. We implemented a preprocessing step to allow Mint to handle such cases so that the stencil analyzer can effectively collect the stencil pattern appearing in the computation.

The Mint translator preprocesses the CUDA kernels generated by the baseline translator in order to make indices more explicit to the stencil analyzer. In the Harris algorithm, an index to the *data* array is a function of l, m and n which are functions of i, j, and k. In order to eliminate the l, m and n indices from the loop body, Mint rewrites the references to arrays in terms of i, j and k. First, it determines the unrolling factor and recursively unrolls the loops. The unrolling factor is the difference between the upper and lower bounds of the loop: the window size. In this example, it is 5. Mint completely unrolls such loops only if the relation between the indices is based on a small constant. After unrolling, the translator replaces the instances of l, m and n with i, j and k respectively. This process introduces expressions such as $i \pm c_1 \pm c_2$, that involve the index variable and a number of constants. To simplify the index expressions, we apply constant folding. This optimization converts, if possible, the index expressions, into the form $i \pm c$, where c is a small constant. After preprocessing, the stencil analyzer can detect the stencil patterns appearing in the loop body. Fig. 3 summarizes the key components of the compilation steps.

C. Optimizer

We have incorporated a number of optimizations into our optimizer that we have found to be useful in optimizing stencil methods written in CUDA. On-chip memory, i.e shared memory and registers, can greatly reduce the number of global memory accesses [10]. The optimizer chooses the most frequently referenced array(s) as the candidate(s) for registers and shared memory. For the Harris interest point algorithm, the input data that stores the voxels is a good candidate for on-chip memory because it is referenced several times. For example, in a 3D Harris interest point algorithm with 5x5x5 Gaussian convolution, there are 6 references to the voxel array in the inner Gaussian loop (2 per axis). This translates into 125*6 references per thread per Gaussian convolution if each thread is assigned to compute one Harris score. Mint uses on-chip memory, in particular shared memory, to buffer these accesses.

The Mint optimizer lets a thread block load a tile of data and ghost cells into shared memory. Each thread is responsible for a single load into shared memory with some threads also loading ghost cells. Fig. 4 illustrates the references for a 2D case. A thread block reads a tile[1] and three neighboring ghost cells on each side into shared memory. If a thread is assigned to compute the Harris score for the black point, then the blue area is the coverage of the convolution. In a 3D case, we would need six more tiles to cover the ghost cells. Since shared memory is a scarce resource, the optimizer keeps two tiles for the slowest varying dimension in addition to the center tile in shared memory. The rest of the references are through global memory.

The optimizer detects the shareable references and the ghost cells needed by a thread block. It automatically determines which thread should be loading which ghost cells and inserts conditional loads in the generated kernel. For example, in a 3D Harris interest point algorithm with 5x5x5 Gaussian convolution, the optimizer finds 450 shareable references and 135 ghost cells for a 16x16x1 tile and automatically generates the necessary code to handle global memory loads. As a result, by using shared memory, Mint eliminates 450 references per thread out of 750 memory references. A CUDA thread still performs remaining 300 references through global memory.

The reuse of data in shared memory can be increased further through the `chunksize` clause. The programmer can easily

[1]The tile size is a configurable parameter and can be set through the `tile` clause.

Dataset	Size	Number of Voxels	Convolution $3 \times 3 \times 3$				Convolution $5 \times 5 \times 5$			
			Serial	OpenMP	Mint-1	Mint-2	Serial	OpenMP	Mint-1	Mint-2
Engine	$256 \times 256 \times 256$	16,777,216	3.298	0.916	0.0924	0.039	15.375	3.927	0.3896	0.169
Lobster	$301 \times 324 \times 56$	5,461,344	1.079	0.339	0.0312	0.012	5.012	1.466	0.1323	0.057
Tooth	$94 \times 103 \times 161$	1,558,802	0.307	0.094	0.0103	0.003	1.427	0.426	0.0439	0.017
Cross	$66 \times 66 \times 66$	287,496	0.059	0.018	0.0024	0.001	0.264	0.072	0.0106	0.004

TABLE I
PERFORMANCE FOR VARIOUS VOLUME DATASETS WITH DIFFERENT CONVOLUTION SIZE. RUNNING TIMES ARE SHOWN IN SECONDS. MINT-1 REFERS TO THE RESULTS ON TESLA C1060. MINT-2 REFERS TO THE RESULTS ON TESLA C2050.

assign more work to a thread by setting a chunking factor. In the example provided in Listing 1, the chunksize is set to 64 in the outmost loop. That is, a thread computes 64 Harris scores in the slowest varying dimension of a 16x16x64 tile. The optimizer inserts a loop in the generated kernel so that each thread computes more scores. The optimizer uses a buffer to hold 3 tiles and rotates their content in the loop until it processes all 64 iterations.

VI. PERFORMANCE RESULTS

We next demonstrate the effectiveness of Mint by comparing the performance of Mint-generated CUDA, serial CPU and OpenMP implementations. Table I lists the running times in seconds for 4 volume datasets with different convolution sizes.

We used four well-known volume datasets in volume rendering. All four datasets have a single byte at each point in the 3D structured grid. The values range from 0 to 255, indicating the opacity of each point. The first dataset, *Engine*, is CT scan data from General Electric, USA and the second dataset, *Lobster*, is also a CT scan, which is released in the VolVis distribution of SUNY Stony Brook, NY, USA. The *Tooth* data is scanned with the GE Industrial Micro CT scanner. Finally, the *Cross* data is an artificial dataset created by Ove Sommer, in the Computer Graphics Group of the University of Erlangen, Germany.

We obtained the performance results on an Nvidia Tesla C1060 GPU with 4GB device memory and an Nvidia Tesla C2050 GPU with 2.5GB device memory. The serial and OpenMP codes were compiled with gcc 4.3.3 and run on a quad-core Intel Xeon Nehalem processor. The *OpenMP* results were obtained by 4 OpenMP threads. We used the version CUDA v3.2 of the CUDA toolkit. All computations were run in single precision.

The convolution size affects the accuracy of the feature selection algorithm. Larger windows result in more accurate solution but longer running times. For example, the serial implementation of the *Engine* dataset takes 3.3 sec if 3x3x3 patches are used in the convolution. However, it takes over 15 sec to detect the features when the Gaussian convolution uses 5x5x5 patches. The GPU acceleration of the feature selection algorithm allows the users to detect features much faster without sacrificing accuracy. Since all the Mint-generated kernels run under 1 sec, Mint achieves real-time feature selection.

We consider that both Mint and OpenMP require a similar amount of programming effort but Mint delivers a substantial performance improvement because it enables the annotated code to be accelerated on the GPU. On Tesla C1060 (Mint-1), Mint delivers 6.8 to 11.9 times the performance of OpenMP running with 4 threads on a multicore architecture. The speedup is larger for large datasets (over 10x) because we can effectively occupy all the stream processors on the GPU device and hide the memory latency better. On the Fermi-based GPU (Mint-2), Mint delivers 18 to 25.7 times the performance of OpenMP. Both Fermi and Tesla results were obtained with the same tile and chunksize parameters without modifying the translator.

Another benefit of using Mint is the programmer's productivity. The programmer has to make modest number of modifications in the input code to enable GPU acceleration. For example, we have simply inserted 5 lines of Mint code into the original C implementation of the Harris interest point algorithm. Compared to the 389 lines of the original code, this is negligible. Moreover, the programmer can easily change the convolution size in the input program and regenerate the CUDA code with Mint. It would be cumbersome for the programmer to implement the CUDA variants of the algorithm using different convolution sizes because each variant requires a different number of ghost cell loads and sharing between threads. The translator automatically determines the communication between threads with the help of the stencil analyzer in the preprocessing step and applies optimizations accordingly.

A. Optimization Levels

Next we discuss the impact of the Mint optimizer on the performance. Fig. 5 shows the performance improvements as a result of cumulative optimizations for 3x3x3 convolution applied to 4 datasets. *OpenMP 4* indicates the performance of the OpenMP implementation running with 4 threads. *Baseline* refers to the performance of the codes generated by the Mint baseline translator. This variant resolves all array references through device memory. *Register* turns on the register optimization that uses registers to accommodate a part of the global memory references. *Shared* uses shared memory in addition to the registers to further improve the reuse of data by buffering memory accesses. Lastly, *chunksize* indicates the performance when a chunking factor is set for the nested-loop in the input program. In this variant, each thread computes a number of elements in the outmost loop as opposed to a single element as in previous variants. We set the chunking factor to 64 to obtain results for this optimization. In all Mint variants, loops are annotated with `nest(3)` to create multi-dimensional thread blocks.

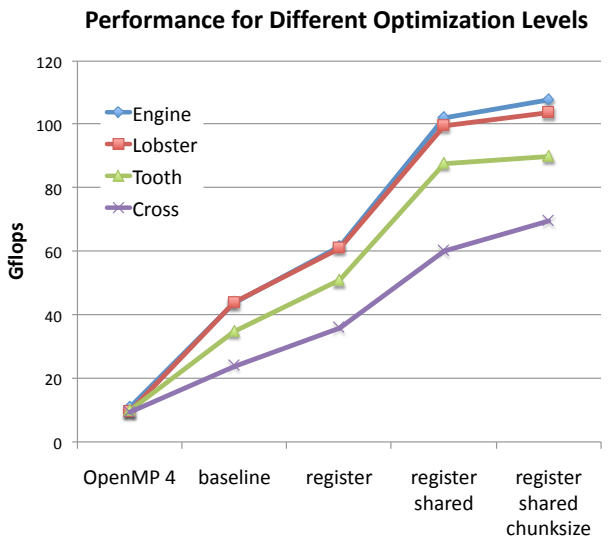


Fig. 5. Effect of the Mint optimizer on the performance on Tesla C1060. Convolution 3x3x3.

The baseline Mint code outperforms the OpenMP code running on 4 CPU cores and provides on average a 3.5x speedup. When the optimization flag is turned on, Mint improves the performance still further. As shown in Fig. 5, both register and shared memory optimizations substantially improve the performance of the baseline translation because they reduce the redundant memory accesses to the device memory. We observed performance improvements in the range of 40-50% with register optimization and 140-160% when combined with shared memory optimization. The chunksize optimization is effective but not as effective as on-chip memory optimizations. The main reason for the modest performance improvement is that assigning more elements to a thread highly increases the register usage per thread. Since the number of registers is limited on the device, this optimization may lead to fewer concurrent thread blocks. However, we still observe the benefit of applying this optimization on all test data. The overall optimizations provide on average a 2.6x speedup for four 3D volume datasets over the baseline translation. The speedup ranges from 2.5 to 2.9.

VII. Related Work

In order to facilitate CUDA programming, several source-to-source translators have been proposed [11], [12]. OpenMPC [12] takes a directive-based approach and supports an extended OpenMP syntax for GPUs. However, one of the limitations of the model is that OpenMPC only parallelizes the outmost loop of a loop-nest causing severe performance penalties. Similar to Mint, the PGI compiler [11] can parallelize loop-nest and applies on-chip memory optimizations. By comparison, Mint can use the on-chip memory resources more effectively because Mint uses domain-specific knowledge to reduce the large optimization space. This results in improved performance for the selected application domain.

Micikevicius [10] focused on order-k space stencils of wave equation and examined the single GPU optimizations of stencil computation. We greatly benefitted from his work and integrated the optimizations into the Mint translator. Our contribution is to make the translation and optimization processes entirely automated. In addition, we implemented a broader set of optimizations, such as unrolling and constant folding to improve our stencil analyzer. Consequently, Mint lifts the burden of CUDA programming from the user while still offering competitive application performance.

VIII. Conclusion

We have studied auto-parallelization and optimization of the 3D Harris interest point algorithm with Mint. Mint simplifies the view of the GPU hardware while providing competitive performance. On an Nvidia Tesla C1060 the Mint-generated kernels achieve 10 times the performance of OpenMP. We incorporated a number of optimizations in the translator including unrolling, constant folding and on-chip memory optimizations. Although we discuss with a single feature detection algorithm, the convolution is widely used in computer vision algorithms. We believe that the applicability of Mint is much broader and expect that Mint to be used as a tool for non-experts to get started with GPU programming while avoiding a lot of low level programming.

References

[1] D. Unat, X. Cai, and S. Baden, "Mint: Realizing CUDA performance in 3D stencil methods with Annotated C," in *International Conference on Supercomputing*, ICS'11, 2011.

[2] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151, 1988.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004. 10.1023/B:VISI.0000029664.99615.94.

[4] P. Kube and P. Perona, "Scale-space properties of quadratic feature detectors," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 18, pp. 987 –999, Oct. 1996.

[5] B. Lorensen, C. Bajaj, L. Sobierajski, Machiraju, and J. Lee, "Visualization viewpoints: The transfer function bake-off," *IEEE Computer Graphics and Applications*, vol. 21, 2001.

[6] H. Pfister, B. Lorensen, W. Schroeder, C. Bajaj, and G. Kindlmann, "The transfer function bake-off," in *VIS '00: Proceedings of the conference on Visualization '00*, (Los Alamitos, CA, USA), pp. 523–526, IEEE Computer Society Press, 2000.

[7] J. Kniss, G. Kindlmann, and C. Hansen, "Multidimensional transfer functions for interactive volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 3, pp. 270–285, 2002.

[8] H. S. Kim, J. P. Schulze, A. C. Cone, G. E. Sosinsky, and M. E. Martone, "Dimensionality reduction on multi-dimensional transfer functions for multi-channel volume data sets," *Information Visualization*, vol. 9, no. 3, pp. 167–180, 2010.

[9] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan, "Treating a user-defined parallel library as a domain-specific language," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pp. 324–, IEEE Computer Society, 2002.

[10] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, ACM, 2009.

[11] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 43–50, ACM, 2010.

[12] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *SIGPLAN Not.*, vol. 44, pp. 101–110, February 2009.