Auto-tuning a High-Level Language Targeted to GPU Codes

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, John Cavazos
Computer and Information Sciences, University of Delaware, Newark, DE 19716
{sgrauerg,xulifan,rsearles,sudhee,cavazos}@udel.edu

ABSTRACT

Determining the best set of optimizations to apply to a kernel to be executed on the graphics processing unit (GPU) is a challenging problem. There are large sets of possible optimization configurations that can be applied, and many applications have multiple kernels. Each kernel may require a specific configuration to achieve the best performance, and moving an application to new hardware often requires a new optimization configuration for each kernel.

In this work, we apply optimizations to GPU code using HMPP, a high-level directive-based language and source-to-source compiler that can generate CUDA / OpenCL code. However, programming with high-level languages may mean a loss of performance compared to using low-level languages. Our work shows that it is possible to improve the performance of a high-level language by using auto-tuning. We perform auto-tuning on a large optimization space on GPU kernels, focusing on loop permutation, loop unrolling, tiling, and specifying which loop(s) to parallelize, and show results on convolution kernels, codes in the PolyBench suite, and an implementation of belief propagation for stereo vision. The results show that our auto-tuned HMPP-generated implementations are significantly faster than the default HMPP implementation and can meet or exceed the performance of manually coded CUDA / OpenCL implementations.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Performance

Keywords

Auto-tuning, GPU, CUDA, OpenCL, Optimization, Belief Propagation

1. INTRODUCTION

The utilization of parallel programming on architectures such as the graphics processing unit (GPU) leads to significant speedup over a sequential CPU implementations for many algorithms. In particular, there is a large body of work utilizing NVIDIA's CUDA architecture to obtain a speed-up using GPUs, e.g., audio/video processing, medical imaging, and black hole simulation [18].

In order to achieve the best performance possible, we may need to use particular code transformations, such as loop tiling and unrolling, permutation, and fusion / fission. However, constant tweaking is required to determine which transformations give the best performance, and the code resulting from these transformations is brittle. A kernel may be optimized for the current architecture, but the same code might give poor performance on a different architecture.

In this paper, we use HMPP Workbench, a directive-based compilation framework targeted to GPUs. The tool allows us to insert pragmas to transform sequential code and generate CUDA and OpenCL code that can potentially match or exceed the performance of manually tuned GPU kernels.

To find a good set of transformations to apply to input code on a given architecture, we use auto-tuning, a process by which we optimize the code with several applicable transformation configurations and then pick the optimized version of the code with the best performance. Specifically, we use auto-tuning with HMPP transformations to convert programs written in sequential C code to generate parallel CUDA / OpenCL code that gives improved performance over the default HMPP configuration. In this work, we use this approach to obtain optimized versions of 2D and 3D convolution kernels, codes in the PolyBench [1] suite, and an implementation of belief propagation for stereo vision. The main contributions of this paper are: (1) showing that auto-tuning applied to GPU kernels written in a directive-based language can be used to effectively parallelize and optimize a variety of codes, in many cases meeting or exceeding the performance of hand-written GPU programs, (2) showing how particular transformations affect performance and describing the best transformation configuration found for each kernel, and (3) showing that the best transformations are kernel and architecture specific.

We give a background of GPU computing in Section 2, describe directive-based programming and the HMPP Workbench in Section 3, discuss our experiment setup in Section 4, show optimized HMPP results on 2D/3D convolution, PolyBench kernels, and belief propagation for stereo vision in Section 5, describe related work in Section 6, and discuss conclusions and future work in Section 7.

2. GPU COMPUTING

The graphics processing unit (GPU) has recently evolved from specialized hardware to a powerful co-processor to the CPU. The most powerful GPUs can perform more FLOPS

```
#pragma hmppcg unroll 2, contiguous
for (i = 0; i < N; i++)
{
        B[i] = A[i];
}
```

```
for (i=0; i < N/2; i++)
{
        B[2*i] = A[2*i];
        B[2*i + 1] = A[2*i + 1];
}
```

(a) Transformation using 'contiguous' unroll, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

```
#pragma hmppcg unroll 2, split
for (i=0; i < N; i++)
{
        B[i] = A[i];
}
```

```
for (i = 0; i < N/2; i++)
{
        B[i] = A[i];
        B[i + N/2] = A[i + N/2];
}
```

(b) Transformation using 'split' unroll, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

```
#pragma hmppcg tile i:2
for (i=0; i < N; i++)
{
        B[i] = A[i];
}
```

```
for (i=0; i < N/2; i++)
{
        for(i_2 = 0; i_2 < 2; i_2++)
        {
                B[2*i + i_2] = A[2*i + i_2];
        }
}
```

(c) Transformation using tiling, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation

Figure 1: Output code generated by transforming loop on left with HMPP pragmas for unrolling and tiling.

and have greater memory bandwidth than the most powerful CPUs [28]. The development of the CUDA and OpenCL programming environments allow the programmer to run multiple threads in parallel to utilize the parallel processing power of the GPU without the use of a graphics API.

In the CUDA / OpenCL environment, the parallel threads are organized into 1D, 2D, or 3D structures called thread blocks, and each thread block is placed in a 1D or 2D grid of thread blocks. The number of threads in a thread block is limited to a value determined by the particular GPU. Threads within a thread block execute on the same GPU multiprocessor as part of a 32-thread chunk known as a warp, have access to a common space of fast, on-chip shared memory, and can synchronize with each other.

The programming guide from NVIDIA [28] suggests various optimizations when programming on the GPU, including adjusting the thread block size to increase multiprocessor occupancy, using registers and shared memory rather than high-latency global memory for data accesses, and organizing the threads in a way that favors coalesced global memory accesses.

3. DIRECTIVE-BASED GPU PROGRAMMING

In this work, we use directive-based GPU programming using the HMPP Workbench from CAPS Entreprise to generate GPU code and explore a large space of possible optimizations. The HMPP compiler framework is shown in Figure 2. This tool allows the programmer to generate CUDA / OpenCL kernels from loops written in sequential C code via pragmas placed before the kernel and in the command to run the kernel [11]. A project to make HMPP directives an open standard called OpenHMPP [8] is underway.

There are benefits to developing applications in a high-level directive-based language as compared to developing in low-level languages, such as OpenCL and CUDA. For example, high-level languages preserve the serial implementation of the code. The focus is on highlighting the parallelism in the code as opposed to developing a platform-specific implementation of the parallelism. A particular advantage of a directive-based approach is that it eases the interaction between domain scientists and programmers. However, programming with high-level languages may mean a loss of performance when compared to programming in low-level languages.

In addition to pragmas to specify parallelization, HMPP provides pragmas to drive code transformations for optimization. These include pragmas for loop permutation where the loops are re-ordered, tiling / unrolling of loops at any factor with varying parameters, and fusion / distribution of computations in loops. Figure 1 shows the input and output for tiling and loop unroll transformations using the 'contiguous' and 'split' unroll schemas using factor 2. When using the 'contiguous' option to unroll a loop that sequentially steps through an array of size N, the array accesses in the first iteration of the unrolled loop are to indices 0 and 1. When using the 'split' unrolling schema, the accesses in the first iteration are to indices 0 and N/2, which helps maintain memory coalescence across threads. To determine the behavior of the remaining iterations that occur when the unroll factor does not perfectly divide the array, HMPP provides 'remainder' and 'guarded' pragmas. The default 'remainder' option generates a remainder loop that is processed after the unrolled loop, while the 'guarded' option inserts guards in the unrolled loop so only the intended computations are run in the final iteration.

In addition to CAPS, Cray and PGI offer directive-based GPU programming. Also, the cuda-lite framework [38] shares some similarities with the HMPP Workbench since both utilize code annotations to simplify the process of optimizing GPU code. However, the cuda-lite framework operates at a lower level, with the annotations placed within a CUDA kernel, while the pragmas in directive-based GPU program-

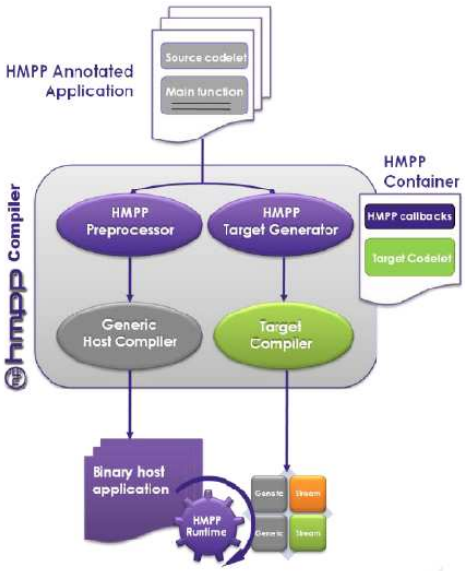

Figure 2: HMPP framework used for turning a sequential C program with specific pragmas to an optimized GPU program. (graphic courtesy of CAPS Entreprise)

ming are placed in high-level sequential code.

4. EXPERIMENTAL SET-UP

Our experiments explore the use of HMPP code transformation pragmas to perform auto-tuning to optimize GPU kernels written using directives in a high-level language. Using HMPP directives, we can perform auto-tuning on a large optimization space on CUDA and OpenCL kernels. In this work, we focus on loop permutation, loop unrolling, loop tiling, and specifying which loops to parallelize.

We run the experiments on a cluster of nodes with NVIDIA C2050 GPUs. This GPU is based on the GF100 (Fermi) architecture. It contains 14 multiprocessors with 32 processors each for a total of 448 parallel processors. Each multiprocessor contains 32768 registers and 64 KB which is split between shared memory and L1 cache. The programmer can allocate 16 KB shared memory and 48 KB L1 cache or 48 KB shared memory and 16 KB L1 cache. In addition to the GPUs, each node of our experimental setup contains two Intel 5530 Quad core Nehalem CPUs clocked at 2.4 GHz with 8 MB cache.

We start with optimizing 2D and 3D convolution kernels, go on to optimize the codes in the PolyBench suite, and finally work on optimizing belief propagation for stereo vision. In each of the codes, we apply 'contiguous' and 'split' loop unrolling using the 'remainder' and 'guarded' options for the remainder loop. We also explore tiling and permutation as appropriate and look at which loops are best to parallelize. The contiguous and split unrolling schemas along with the tiling schema are described and shown in Section 3.

The transformed codes are generated using a python script that contains a list of unroll, tiling, permutation, and parallelization transformations to use for a particular kernel. Running the script generates and compiles codes with every

```
%(permutePragma)
%(unrollTilePragma_iLoop)
%(parallelNoParallelPragma_iLoop)
for (i = 0; i < NI; i++)
{
      %(unrollTilePragma_jLoop)
      %(parallelNoParallelPragma_jLoop)
      for (j = 0; j < NJ; j++)
          {
              c[i][j] *= p_beta;
              %(unrollTilePragma_kLoop)
              %(parallelNoParallelPragma_kLoop)
              for (k = 0; k < NK; k++)
                {
                  temp = p_alpha * a[i][k] * b[k][j];
                  c[i][j] += temp;
                }
          }
}
```

Figure 3: Framework for GEMM kernel in PolyBench to insert transformations.

combination of input transformations. In order to generate the annotated codes, the script uses a file with the source for the code with specified locations to place each pragma for code transformation. Figure 3 shows a portion of this file for the PolyBench GEMM kernel. The code contains a place to put a pragma to permute the loops as well as places to insert pragmas for unrolling / tiling each loop and to specify whether or not to parallelize each target loop. We go on to run each transformed program to determine the best input configuration.

The measured run-times include the running time of each kernel and the overhead involved in calling the kernel (possibly multiple times) but not the time to transfer data from the CPU to the GPU and vice versa. All the computations used for auto-tuning are performed on 32-bit floating point values and utilize thread block dimensions of 32 x 8 to allow for full GPU multiprocessor occupancy.

5. EXPERIMENTAL RESULTS

In this section, we present our auto-tuning HMPP results on 2D convolution and 3D convolution, the PolyBench benchmark suite, and belief propagation as applied to stereo vision. In many of the results, the auto-tuned HMPP results meet or exceed the performance of hand-written CUDA and OpenCL versions of the codes.

5.1 2D Convolution

```
for (int i = 0; i < DIM_0 - 1; i++)
  for (int j = 0; j < DIM_1 - 1; j++)
    B[i][j] =
         C_11 * A[i-1][j-1]+C_12 * A[i+0][j-1] +
         C_13 * A[i+1][j-1]+C_21 * A[i-1][j+0] +
         C_22 * A[i+0][j+0]+C_23 * A[i+1][j+0] +
         C_31 * A[i-1][j+1]+C_32 * A[i+0][j+1] +
         C_33 * A[i+1][j+1];
```

Figure 4: 2D Convolution Kernel.

First, we look at optimizing the 2D convolution kernel shown in Figure 4 using code transformations in HMPP. Our experiments are performed using CUDA and OpenCL with arrays of size 4096 x 4096. In our initial experiment, we reverse the loop order via permutation and find that the alter-

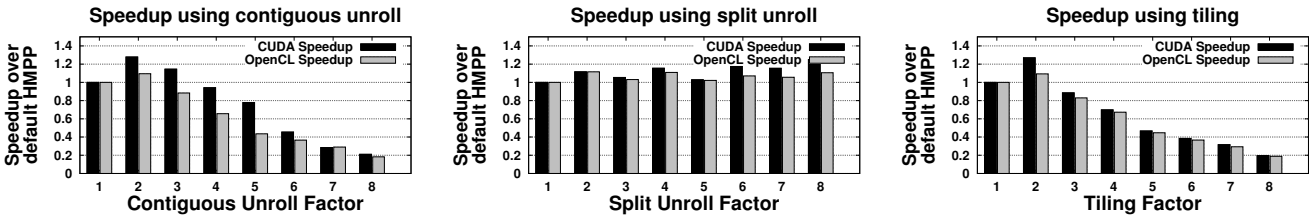

Figure 5: Speedup in 2DCONV using (from left to right) 'contiguous' unrolling, 'split' unrolling, and tiling with factors 1-8 in the inner loop. The loops are in the initial order and outer loop is not unrolled.

nate loop order significantly increases the runtime, causing us to keep the initial loop order. Also, experiments involving loop unrolling showed that using the 'guarded' option for remainder behavior gave similar or better results than the 'remainder' option, causing us to use the 'guarded' option in all ensuing experiments on this kernel.

Our next experiments look at tiling and unrolling the inner loop at factors 1-8, using both the 'contiguous' and 'split' schemas for unrolling. The results for these experiments are shown in Figure 5. The CUDA results show that applying 'contiguous' unrolling or tiling with factor 2 gives the best results, but the speedup corresponding to these schemas drops off with larger factors while there continues to be a speedup at larger factors in split unrolling. The pattern is a little different in the OpenCL results. Applying 'contiguous' unroll or tiling with factor 2 gives some speedup, but the best speedup occurs with an unroll factor of 8 using the 'split' schema.

The likely explanation for run-time improvement when using a contiguous unroll or tiling factor of 2 is that the convolution kernel is structured such that contiguous loop unrolling / tiling allows particular data to be shared across unrolled iterations, decreasing the number of memory accesses. However, contiguous unrolling weakens the capability to perform coalesced memory accesses, particularly in higher unroll factors. This causes a drop-off in performance with higher unroll factors. Meanwhile, the 'split' unroll schema does not provide data sharing between iterations, but does allow the kernel to maintain coalesced memory accesses, preventing this performance drop-off.

The results of the best found configurations for this kernel are also shown with results on other kernels in Figures 8 and 9 and described in Table 3.

5.2 3D Convolution

Next, we look into optimizing the 3D convolution kernel shown in Figure 6 using 256 x 256 x 256 arrays.

Our initial experiments use each of the 6 possible loop orderings via permutation without any tiling or unrolling. Loop order can influence memory coalescence on the GPU, and memory coalescence or lack of it can be a major factor in the run-time of a GPU kernel. To simplify the discussion of loop order, we refer to the outer 'i' loop as loop '1', the middle 'j' loop as loop '2', and the inner 'k' loop as loop '3'.

In this kernel, there are three nested loops which can theoretically be parallelized, but HMPP only supports parallelizing up to two nested loops. By default, the first two loops are parallelized, and a non-optimal loop permutation order can lead to non-coalesced memory accesses and a longer runtime.

```
for (int i = 0; i < DIM_0 - 1; i++)
  for (int j = 0; j < DIM_1 - 1; j++)
    for (int k = 0; k < DIM_2 - 1; k++)
        B[i][j][k] = C_11 * A[i-1][j-1][k-1]+
                     C_13 * A[i+1][j-1][k-1]+
                     C_21 * A[i-1][j-1][k-1]+
                     C_23 * A[i+1][j-1][k-1]+
                     C_31 * A[i-1][j-1][k-1]+
                     C_33 * A[i+1][j-1][k-1]+
                     C_12 * A[i+0][j-1][k+0]+
                     C_22 * A[i+0][j+0][k+0]+
                     C_11 * A[i-1][j-1][k+1]+
                     C_32 * A[i+0][j+1][k+0]+
                     C_13 * A[i+1][j-1][k+1]+
                     C_21 * A[i-1][j+0][k+1]+
                     C_23 * A[i+1][j+0][k+1]+
                     C_31 * A[i-1][j+1][k+1]+
                     C_33 * A[i+1][j+1][k+1];
```

Figure 6: 3D Convolution Kernel.

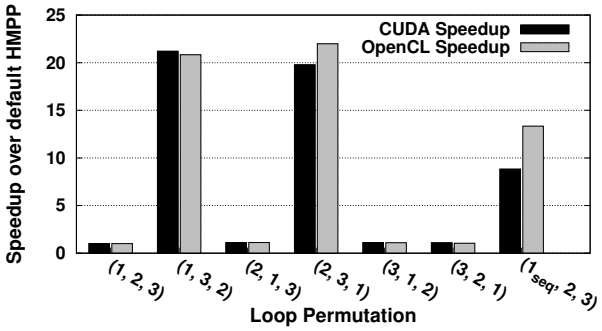

Figure 7: Speedup over default HMPP in 3DCONV using different permutations with default parallelization and initial permutation with 2nd and 3rd loops parallelized.

The results of our initial experiments are shown in Figure 7 and show a significant speedup over the default (1,2,3) configuration in the permutations where the inner parallelized loop is loop '3' in the initial code.

Next, we keep the default (1,2,3) loop ordering and use an alternate method to determine loop parallelization, inserting pragmas to set the first loop to 'noParallel' and the subsequent two loops to 'parallel', causing the final two loops to be parallelized while the first loop remains sequential. The runtime of this configuration is shown alongside the results of each permutation in Figure 7 and show that the CUDA and OpenCL results using this configuration are slower than the (1,3,2) and (2,3,1) permutations but faster than the other permutations.

Additional experiments explore tiling and unrolling each

Pragma	Experimental Parameter Values in Codes
permute	Depends on kernel. Different Ordering of loops.
unroll	Unroll factors 1 through 8 using 'contiguous' and 'split' options.
tile	Tiling factors 1 through 8.
parallel / noParallel	Depends on kernel. Determines which loops are parallelized for GPU processing.
remainder / guarded	Used each option with loop unrolling. 'Remainder' option allows generation of remainder loop. The 'guarded' option avoids this via guards in unrolled loop.

Table 1: Transformations applied to each input kernel for optimization.

loop at various factors in the best permutations. In the CUDA experiments, the speedup of the (1,3,2) permutation goes from 21.2x to 27.2x when unrolling loop '3' by a factor of 4 using the 'contiguous' schema and 'guarded' remainder option. This represents the best found CUDA speedup. In the OpenCL experiments, the best found configuration uses the (2,3,1) permutation without any unrolling or tiling. The speedup of this configuration is 22x over the default. The results of these configurations are shown alongside other codes in Figures 8 and 9 and described in Table 3.

5.3 The PolyBench Benchmark Suite

The PolyBench collection of benchmarks contains codes for linear algebra, linear algebra solvers, data-mining, and stencils, all with static control parts [1]. We convert a number of codes in this benchmark suite to CUDA / OpenCL using HMPP pragmas. These codes are described in Table 2. For the ATAX and BICG codes, we manually performed loop distribution on the inner loop nests before running the experiments to allow greater parallelism than in the initial code structure. We will automate this process in future work via the 'distribute' HMPP pragma.

In the current set of experiments, we look at permutation, unrolling, tiling, and specifying which loop(s) to parallelize with the parameters shown in Table 1. An exhaustive search of all possible transformations within this space is not feasible, as many of these kernels contain a number of loops, so a sampling of this space is utilized. No particular formula determines the sampling as every kernel is different. We try and include the transformations that will retrieve the best results based on results of other kernels. In the future, we may be able to improve results by doing a more exhaustive 'local search' around the best optimizations found in these experiments. The array dimensions used in our experiments, running time of a sequential CPU implementation using these dimensions, and number of HMPP-transformed versions of each converted PolyBench code are given in Table 2. Also, the cluster described in Section 4 is used for the CPU running times.

We constructed hand-written CUDA and OpenCL kernels to compare with the optimized HMPP results. These kernels are tuned to maximize parallelism and coalescence of memory accesses but are not heavily optimized. For these initial experiments, we used 32-bit float data. The resulting speedup of the best HMPP-transformed version of these codes over the default HMPP configuration and the hand-coded results are in Figures 8 and 9 alongside the convolution results. The default HMPP and best found configuration running times are shown in Table 2, and the best found HMPP configuration for each code is described in Table 3. The results show that both the CUDA and OpenCL default HMPP configurations are faster than CPU implementations on 12 of the 15 kernels, in many cases by a large margin. In addition, the best found HMPP configurations from auto-tuning beat the CPU result on all 15 kernels and are often significantly faster than the default HMPP configuration. Section 5.5 discusses these results in more detail.

Our hand-written GPU kernels as well as the default and best found optimized HMPP-annotated codes are distributed as the PolyBench/GPU 1.0 package, available at `http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU`.

5.4 Experiments Using Doubles

To see if we can generalize our auto-tuning results on 32-bit floats to 64-bit doubles, we take the best transformations found for floats and apply these transformations to the same codes using doubles. We also ran manual implementations of the kernels using doubles and normalized the results to the running time of the default HMPP configuration using doubles. The results when using doubles for CUDA and OpenCL are shown on the right of Figures 8 and 9. These results show that the configurations that give speedups on floats also give speedups using doubles on most of the kernels, though not always to the same extent. Specifically, the geometric mean of the speedup using these configurations over default HMPP on CUDA (OpenCL) is 2.73x (2.62x) using floats and 2.24x (2.24x) using doubles.

5.5 HMPP Auto-Tuning Results Discussion

Figures 8 and 9 show results from our auto-tuning experiments on convolution and PolyBench codes using HMPP on a C2050 (Fermi) GPU. The running times of the sequential CPU, default HMPP, and best found HMPP-transformed configurations are given in Table 2. Each auto-tuned code beats the performance of the default code produced by the HMPP compiler, in some cases by a large margin. All the auto-tuned HMPP CUDA codes beat the default HMPP CUDA configuration by over 13 percent, with the speedup ranging from 1.135x for the SYRK kernel to 49.7x for the ATAX kernel. In addition, the average performance of the auto-tuned kernels show a geometric mean speedup of 2.73x compared with 2.02x using hand-written kernels. When running the best auto-tuned configuration using 64-bit doubles, there continues to be a speedup in most of the codes. The geometric mean of the speedup is 2.24x over default HMPP.

The results are similar using OpenCL, though some of the speedups are lower, particularly on kernels where the best found configuration involves loop unrolling. One possible explanation for this is that CUDA codes are compiled with an Open64-based compiler in CUDA 4.0 while OpenCL codes are compiled with an LLVM-based compiler. Thus, the LLVM-based compiler may be doing a better job optimizing some of the non-transformed HMPP codes. Still, auto-tuned HMPP targeting OpenCL gives a speedup over default HMPP on many of the codes. The geometric mean of the OpenCL speedup using the auto-tuned results is 2.62x and 2.24x over default HMPP using floats and doubles, re-

Code	Description	Size of Array Dimension(s)	Number of Versions	CPU Runtime	CUDA Runtime on Fermi		OpenCL Runtime on Fermi	
					Default	Best	Default	Best
2DCONV	2D Convolution	4096	198	0.160	0.00355	0.00277	0.00338	0.00294
2MM	2 Matrix Multiply (G=AxB)	2048	163	188	0.766	0.639	0.745	0.742
3DCONV	3D Convolution	256	137	0.360	0.202	0.00743	0.204	0.00926
3MM	3 Matrix Multiply (E=AxB; F=CxD; G=ExF)	512	116	1.40	0.0165	0.0139	0.01580	0.01578
ATAX	Matrix Transpose and Vector Multiplication	4096	130	0.506	0.497	0.0100	0.587	0.00959
BICG	BiCG Sub Kernel of BiCGStab Linear Solver	4096	490	0.386	0.510	0.0184	0.608	0.0188
CORR	Correlation Computation	2048	152	71	6.88	2.64	6.89	2.38
COVAR	Covariance Computation	2048	145	70	6.92	2.50	6.91	2.35
FDTD-2D	2D Finite Difference Time Domain Kernel	2048 w/ 500 time-steps	177	24.5	1.17	1.02	0.886	0.876
GEMM	Matrix Multiply C = αA x B + βC	512	254	0.347	0.00601	0.00506	0.00549	0.00545
GESUMMV	Scalar, Vector and Matrix Multiplication	4096	679	0.0321	0.0379	0.0241	0.0377	0.0254
GRAMSCHM	Gram-Schmidt Process	2048	725	190	7.49	6.51	7.54	7.51
MVT	Matrix Vector Product and Transpose	4096	125	0.23	0.0190	0.00954	0.0189	0.00918
SYR2K	Symmetric rank−2k operations	2048	178	22.8	15.70	12.81	15.70	12.92
SYRK	Symmetric rank−k operations	1024	123	1.81	0.465	0.409	0.472	0.408

Table 2: Kernel description, size of each array dimension, number of HMPP-transformed versions generated, and runtime (in seconds) of sequential CPU implementation and of default HMPP and best found HMPP-transformed configurations on C2050 (Fermi) using floats on 2D/3D convolution and parallelized PolyBench codes.

Code	HMPP CUDA	HMPP OpenCL
2DCONV	Unroll 2nd loop using 'contiguous' and 'guarded' options with factor 2	Unroll 2nd loop using 'split' and 'guarded' options with factor 8
2MM	Unroll 3rd and 6th loops using 'split' and 'guarded' option w/ factor 3	Unroll 3rd and 6th loops using 'split' and 'guarded' options w/ factor 4
3DCONV	Permute loop-nest order from (1,2,3) to (1,3,2); unroll loop 2 using 'contiguous' and 'guarded' options w/ factor 4	Permute loop-nest from (1,2,3) to (2,3,1) ordering
3MM	Unroll 3rd, 6th, and 9th loops using 'split' and 'guarded' options w/ factor 3	Unroll 3rd, 6th, and 9th loops using 'contiguous' and 'guarded' options with factor 8
ATAX	Reverse order of 2nd nested loop set (4th and 5th loops) using permutation and tile 1st and 2nd loop w/ factor 4	Reverse order of 2nd nested loop set (4rd and 5th loops) using permutation and tile 1st and 2nd loops w/ factor 2
BICG	Reverse order of 1st loop set (2rd and 3rd loops) and tile 2nd, 3rd, 4th, and 5th loops w/ factor 2	Reverse order of first loop set (2rd and 3rd loops) and tile 2nd loop w/ factor 4
CORR	Parallelize 8th loop rather than 7th loop using noParallel/-parallel pragmas and tile 9th loop w/ factor 4	Parallelize 8th loop rather than 7th loop using noParallel/parallel pragmas and unroll 9th loop using 'contiguous' and 'remainder' options w/ factor 2
COVAR	Parallelize 6th loop rather than 5th loop using noParallel/parallel pragmas and unroll 7th loop using 'split' and 'guarded' options w/ factor 4	Parallelize 6th loop rather than 5th loop using noParallel/parallel pragmas
FDTD-2D	Unroll 4th loop w/ factor 2 and 8th loop with factor 4, each using 'split' and 'guarded' options	Unroll 4th and 8th loops using 'split' and 'remainder' options w/ factor 2
GEMM	Unroll 3rd loop using 'split' and 'guarded' options with factor 3	Unroll 3rd loop using 'contiguous' and 'guarded' options with factor 8
GESUMMV	Tile 1st loop w/ factor 4 and tile 2nd loop w/ factor 2	Tile 1st loop w/ factor 4 and tile 2nd loop w/ factor 3
GRAMSCHM	Unroll 5th and 6th loops using 'split' and 'guarded' options w/ factor 3	Unroll 5th and 6th loops using 'contiguous' and 'guarded' options w/ factor 4
MVT	Tile 1st, 2nd, and 4th loops w/ factor 2	Tile 1st loop w/ factor 2
SYR2K	Unroll 5th loop using 'split' and 'remainder' options with factor 3	Unroll 2nd loop using 'split' and 'guarded' options with factor 2 and 5th loop using 'split' and 'guarded' options with factor 4
SYRK	Unroll 5th loop using 'split' and 'guarded' options with factor 2	Unroll 2nd and 5th loops using 'split' and 'remainder' options with factor 2

Table 3: Best found transformations on C2050 (Fermi) for convolution and PolyBench kernels

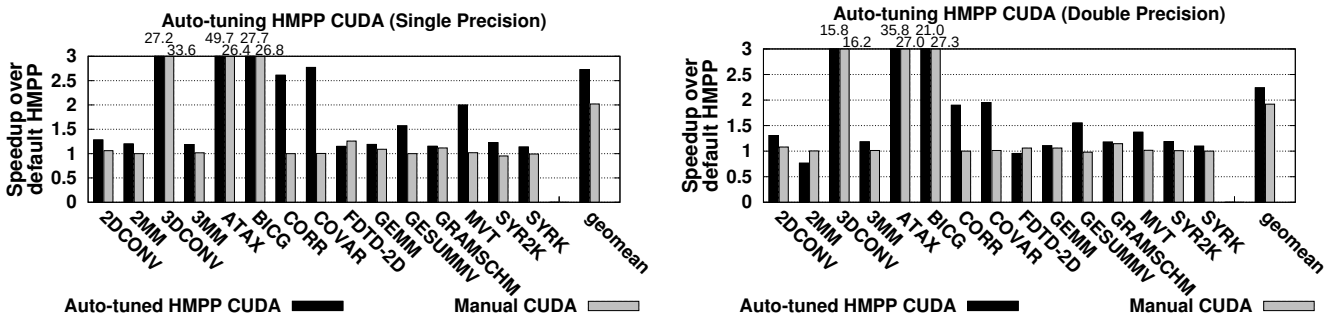

Figure 8: Speedup of auto-tuned HMPP-generated CUDA kernels and hand-written CUDA codes over default HMPP CUDA on 2D/3D convolution and PolyBench codes using single and double precision.

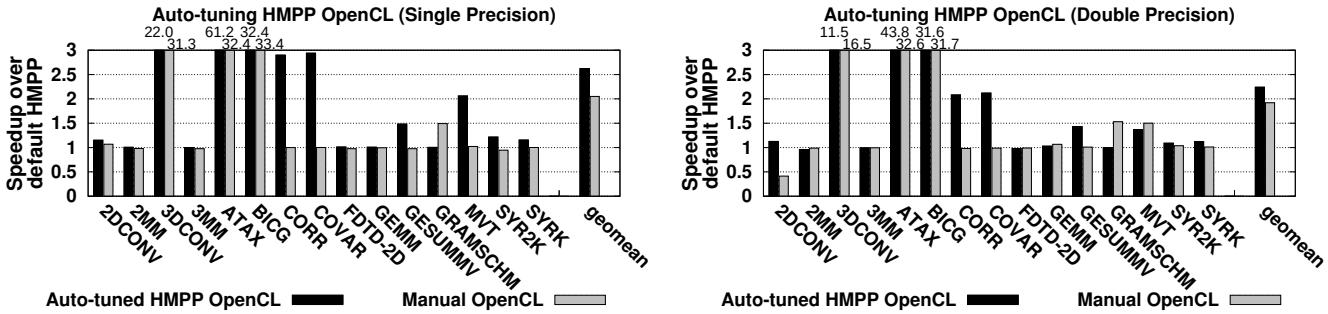

Figure 9: Speedup of auto-tuned HMPP-generated OpenCL kernels and hand-written OpenCL codes over default HMPP OpenCL on 2D/3D convolution and PolyBench codes using single and double precision.

spectively.

The results in Table 3 show some pattern of what transformations should be applied. It is important to use the best loop permutation to maximize memory coalescence. For example, we get a speedup of over 20x compared to the default on the 3DCONV, ATAX, and BICG kernels by using permutation. The programmer should consider what loops are parallelized because the default configuration may not be optimal. In the CORR and COVAR kernels, we get over 2.5x speedup by specifying loops to parallelize. For loop unrolling, the results show that applying unrolling to the innermost loop in a loop nest often gives the best speedup. In the 2DCONV, 2MM, 3MM, GEMM, SYR2K, and SYRK kernels, the best configuration on CUDA and OpenCL involve loop unrolling on the inner loop in particular loop nest(s).

Still, the best found transformations vary across kernels and programming environments, showing a need to experiment with different optimizations and parameter values due to different characteristics of each kernel and programming environment.

5.6 Experiments With Different Architectures

We go on to explore how the best found optimization configurations can vary across GPU architectures. We take the best found CUDA HMPP optimization configuration for each kernel on the C2050 (Fermi) GPU and run them on the NVIDIA GTX 280 (Tesla), which uses the GT200 architecture and has 240 cores, and the NVIDIA 9800GT, which uses the G92 architecture and has 112 cores. The resulting runtime is compared with the running time of the best CUDA HMPP configuration found using auto-tuning on each architecture. The results for each kernel are shown in Figure 10.

Some of the configurations that worked well on the Fermi did not perform as well on the Tesla or 9800 GT. Using the best found transformed kernels on the Fermi and running them on the Tesla or 9800 GT resulted in a slowdown compared to the default HMPP configuration on 4 out of the 15 kernels on the Tesla and 5 of the 15 kernels on the 9800 GT. Still, some of the best optimization configurations, particularly permutation and parallelization of particular loops, did result in speedups across all architectures. For example, in the cases where permutation or parallelization is used in the best found optimization configuration on the Fermi, those configurations also result in a speedup over the default HMPP on the Tesla and 9800 GT GPU architectures. Still, the best optimization configuration found on a specific target architecture, T_1, is often significantly faster than using the best configuration from an alternate architecture, T_2, and using it on T_1. On the left of Figure 10, the geometric mean speedup over the default HMPP configuration for all the kernels on the 'Best Fermi on Tesla' configurations is 1.69x while the speedup of the auto-tuned 'Best Tesla' is 2.39x. On the right of Figure 10, the geometric mean of the speedup of the 'Best Fermi on 9800GT' configurations is 1.34x while the speedup of the auto-tuned 'Best 9800GT' is 1.71x. These results show that the best transformations are not only specific to the kernel, but seem to be specific to the GPU architecture.

5.7 Optimizing Belief Propagation

We go on to use HMPP to optimize belief propagation

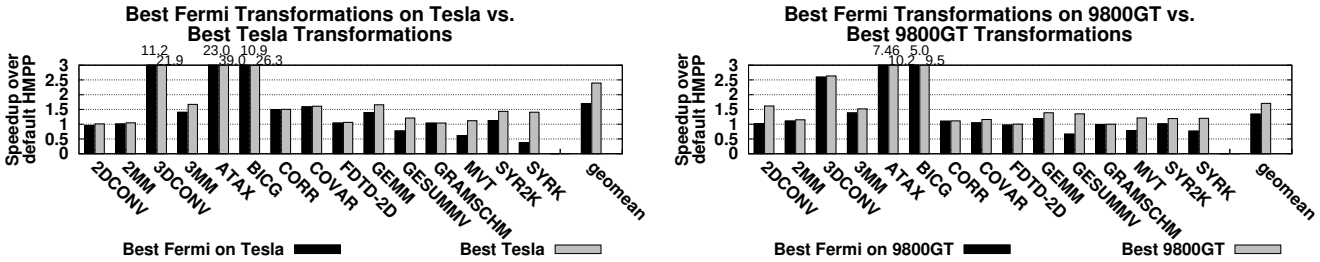

Figure 10: Left: Results of running best CUDA HMPP transformations found on C2050 (Fermi) on GTX 280 (Tesla) vs. best configurations found using auto-tuning on GTX 280 (Tesla); Right: Results of running best CUDA HMPP transformations found on C2050 (Fermi) on 9800 GT vs. best configurations found using auto-tuning on 9800 GT.

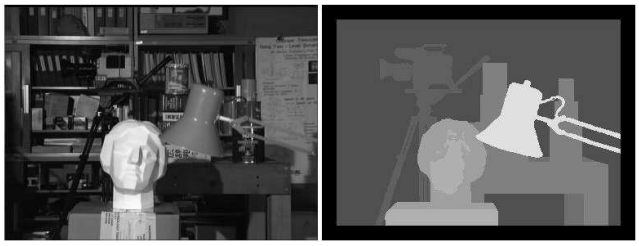

Figure 11: Reference image in Tsukuba stereo set and ground-truth disparity map.

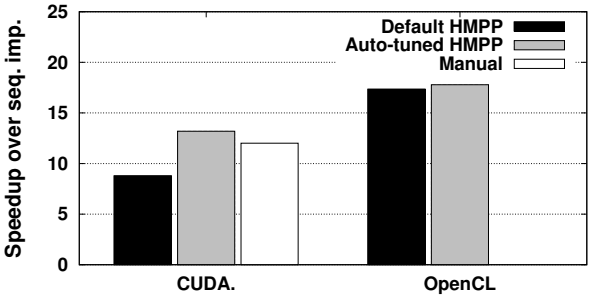

Figure 12: Speedup over sequential implementation of default and optimized HMPP belief propagation on CUDA and OpenCL and of manual implementation on CUDA.

as applied to stereo vision. The input to the algorithm is a stereo set of two images, and the output is a computed disparity map between the images. In Figure 11, we show an image in the Tsukuba stereo set used in our experiments and the corresponding ground-truth disparity map.

We begin with the sequential code corresponding to the work by Felzenszwalb [12] available online. We adjust the implementation to maximize the number of coalesced memory accesses and add HMPP pragmas to run the program on the GPU. Each step of the algorithm is parallelized, as the computations in every step can be performed on each image pixel simultaneously. We profile the default parallel HMPP implementation and find that an iterative 'message-passing' step dominates the runtime, so we focus on optimizing that step. The targeted code repeats for a given number of iterations and consists of message values corresponding to each disparity at each pixel being passed to neighbors and updated. Additional details about each step of the algorithm can be found in Felzenszwalb's work.

Our experiments use the Tsukuba stereo set with image dimensions of 384 x 288 and 15 possible disparity levels. The implementation is run using a single level with 250 message-passing iterations, with the optimized implementation determined in the same manner as in Section 5.3 using the transformations shown in Table 1. The CUDA and OpenCL results of the initial and best optimized HMPP implementations are in Figure 12 alongside results of a manual CUDA implementation developed by Grauer-Gray et al. [15]. The speedup is relative to the initial sequential CPU implementation. Each CUDA and OpenCL implementation performs at least seven times faster than the CPU implementation, with the optimized HMPP implementation giving a greater speedup. In the CUDA results, the manual implementation performs better than default HMPP but worse than our optimized HMPP implementation, showing how HMPP optimizations can improve the performance of a program to outperform a manual GPU implementation.

6. RELATED WORK

There are a number of library generators that automatically produce high-performance kernels, including FFT [13, 29, 36], BLAS [34, 37, 5, 14, 16], Sparse Numerical Computation [19, 33, 26, 4, 23], and domain specific routines [3, 7, 24]. Recent research [21, 22, 17] expands automatic code generation to routines whose performance depends not only on architectural features, but also on input characteristics. These systems are a step toward automatically optimizing code for different architectures. However, these prior works have been largely focused on small domain-specific kernels.

Automatic code generation and optimization for different architectures has been explored in many studies. The research related to loop optimizations includes work by Wolf et al. [35], Kisuki et al. [20], and Stephenson et al. [32] that look at such optimizations on the CPU. Wolf et al. looked at how to best combine particular transformations on the architecture, Kisuki et al. focused on retrieving the optimal tiling size and unroll factor simultaneously, while Stephenson et al. used supervised learning to predict the optimal unroll factor of any loop. These works focused on optimizing for single-core CPUs. In our work, we optimize codes for many-core GPU architectures.

There is also related work on GPU code optimization by Ryoo et al. [31]. They looked at a number of manual optimizations of CUDA kernels, including tiling, pre-fetching, and full loop unrolling within CUDA kernels. Liu et al. [25] looked at varying thread block dimensions and loop unrolling as optimizations in CUDA kernels. Baskaran et al. [2] did an extensive study of loop unrolling within CUDA kernels, but they were concerned with improving the performance of a single application.

Choi et al. [6] developed an auto-tuning framework for sparse matrix-vector multiplication on the GPU. Nukada et al. [27] optimized 3D-FFT of varying transform sizes on GPUs using auto-tuning. These two related works applied auto-tuning to specific applications.

Another related work is the CUDA-CHILL project [30], which can translate loop nests to high performance CUDA code. The developers of this project provide a programming language interface that uses an embedded scripting language to express transformations or recipes. Their work only supports CUDA transformations and it is necessary to program the script, which is not as convenient as using a directive-based approach like HMPP.

Cui et al. ([10] and [9]) introduced the EPOD framework built on Open64 that can encapsulate algorithm-specific optimizations into patterns, and these patterns can be reused in other similar codes. It targets multiple platforms including multi-core CPUs and NVIDIA GPUs. One of the works is focused on generating optimization scripts for BLAS3 GPU kernels. Our work differs because it uses a directive-based tool where the transformations are placed in program code rather than an outside script and also shows results for specific transformations and on a greater variety of codes.

Our work differs from much of the previous GPU work because it does not require analysis and transformations using initial GPU kernels, instead utilizing directives placed in C code to apply the optimizations and generate the kernels. In addition, this work focuses on transformations at a higher level than presented in the related work, looking at transformations of the loop(s) to be parallelized rather than loops within the kernel.

7. CONCLUSIONS AND FUTURE WORK

The problem of determining the best set of optimizations to apply to a kernel to be executed on a GPU has been extensively studied. In this work, we developed optimized GPU kernels using auto-tuning with the directive-based HMPP Workbench. We found that we were able to significantly improve the running time over the default HMPP implementation as well as match or even exceed the performance of manually written CUDA / OpenCL code on kernels in the PolyBench suite and on the belief propagation application as applied to stereo vision. Using code transformations available in HMPP, we were able to quickly generate different versions of code spanning a large space of optimizations.

In the future, we plan to extend this work by looking at more large applications and more optimizations, including distribution / fusion. We also plan to experiment with additional GPU architectures, including AMD GPUs, as well as other many-core architectures.

8. ACKNOWLEDGMENTS

This work was funded in part by the U.S. National Science Foundation through the NSF Career award 0953667 and the Defense Advanced Research Projects Agency through the DARPA Computer Science Study Group (CSSG).

9. REFERENCES

[1] Polybench v2.0. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[3] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of High-performance Parallel Programs for a Class of ab initio Quantum Chemistry Models. *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):276–292, February 2005.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. on Graphics*, 22(3), July 2003. (Proc. SIGGRAPH 2003).

[5] C. Chen, J. Chame, and M. Hall. Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy. In *In Proc. of the International Symposium on Code Generation and Optimization (CGO)*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[7] R. Choy and A. Edelman. Parallel MATLAB: Doing It Right. *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):331–341, February 2005.

[8] O. Consortium. Openhmpp. http://www.openhmpp.org.

[9] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for blas3 on gpus. *Parallel and Distributed Processing Symposium, International*, 0:255–265, 2011.

[10] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. In *Proceedings of the the International Symposium on Code Generation and Optimization (CGO)*, pages 107 –118, april 2011.

[11] C. enterprise. *Hmpp:hybrid compiler for manycore applications-workbench user guide*. CAPS enterprise, 2010.

[12] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. *Int. J. Comput. Vision*, 70:41–54, October 2006.

[13] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999*

conference on Programming language design and implementation, pages 169–180, New York, NY, USA, 1999. ACM Press.

[14] K. Goto and R. A. van de Geijn. On Reducing TLB Misses in Matrix Multiplication. In *Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note 9.*, 2002.

[15] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan. Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction. In *Pattern Recognition in Remote Sensing (PRRS 2008), 2008 IAPR Workshop*, pages 1–4, 2008.

[16] J. D. Hall, N. A. Carr, and J. C. Hart. Cache and Bandwidth Aware Matrix Multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois, Apr. 2003.

[17] S.-C. Han, F. Franchetti, and M. Püschel. Program Generation for the All-pairs Shortest Path Problem. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 222–232, New York, NY, USA, 2006. ACM Press.

[18] W.-m. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[19] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[20] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 237, 2000.

[21] X. Li, M. J. Garzarán, and D. Padua. A Dynamically Tuned Sorting Library. In *In Proc. of the International Symposium on Code Generation and Optimization (CGO)*, pages 111–124, 2004.

[22] X. Li, M. J. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. In *In Proc. of the International Symposium on Code Generation and Optimization (CGO)*, pages 99–110, March 2005.

[23] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2):110–140, 2003.

[24] C. Lin and S. Z. Guyer. Broadway: a Compiler for Exploiting the Domain-specific Semantics of Software Libraries. *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):342–357, February 2005.

[25] Y. Liu, Z. E. Z., and S. Xipeng. A cross-input adaptive framework for gpu program optimizations. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[26] J. Mellor-Crummey and J. Garvin. Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. *Int. J. High Perform. Comput. Appl.*, 18(2):225–236, 2004.

[27] A. Nukada and S. Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[28] NVIDIA. *Nvidia cuda programming guide: Version 3.2*. NVIDIA Corporation, 2010.

[29] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):232–275, February 2005.

[30] G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 136–150. Springer Berlin / Heidelberg, 2011.

[31] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[32] M. Stephenson and S. P. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the the International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.

[33] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *Journal of Physics Conference Series*, 16:521–530, Jan. 2005.

[34] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[35] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[36] J. Xiong, J. Johnson, R. W. Johnson, and D. A. Padua. SPL: A Language and a Compiler for DSP Algorithms. In *Proc. of the International Conference on Programming Language Design and Implementation*, pages 298–308, 2001.

[37] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In *Proc. of Programing Language Design and Implementation*, pages 63–76, June 2003.

[38] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. W.m.w.: Cuda-lite: Reducing gpu programming complexity. In *In: LCPC2008. Volume 5335 of LNCS*, pages 1–15. Springer, 2008.