AUTOMATIC TUNING MATRIX MULTIPLICATION PERFORMANCE ON GRAPHICS HARDWARE

BY

Changhao Jiang and Marc Snir

Department of Computer Science
University of Illinois Urbana Champaign
201 N. Goodwin Avenue
Urbana, IL 61801-2302
{cjiang, snir}@cs.uiuc.edu

Technical Report UIUC DCS-R-2005-2558
Department of Computer Science, UIUC
April, 2005

Automatic Tuning Matrix Multiplication Performance on Graphics Hardware [*]

Changhao Jiang, Marc Snir
Department of Computer Science
University of Illinois Urbana Champaign
201 N. Goodwin Avenue
Urbana, IL 61801-2302
{cjiang, snir}@cs.uiuc.edu

April 25, 2005

Abstract

Graphics hardware's performance is advancing much faster than the performance of conventional microprocessor. In order to utilize the tremendous computing power of these systems, it is critical to tune software to graphics hardware's architectural features. The frequent changes in GPUs' architecture and performance characteristics makes it very desirable for such tuning to be automated.

This paper implements an automatic tuning system to generate high-performance matrix-multiplication implementation on graphics hardware. The automatic tuning system uses a parameterized code generator to generate multiple versions of matrix multiplication, whose performances are empirically evaluated by actual execution on the target platform. An ad-hoc search engine is employed to search over the implementation space for the version that yields the best performance. In contrast to similar systems on CPUs, which utilize cache blocking, register tiling, instruction scheduling etc. tuning strategies, this paper identifies and exploits several tuning strategies that are unique for graphics hardware. These tuning strategies include optimizing for multiple-render-targets, SIMD instructions with data packing, overcoming limitations on instruction count and dynamic branch instruction. The generated implementations have comparable performance with expert manually tuned version in spite of the significant overhead incurred due to the use of the high-level BrookGPU language. As the first attempt in automatic generation of numerical libraries for graphics hardware, the results from this paper are encouraging.

[*]This work is supported by DARPA contract NBCHC-02-0056 and NBCH30390004.

1 Introduction

In the past decade, graphics hardware, a.k.a graphics processing unit (GPU)[1], has enjoyed a faster growth than what Moore's law dictates. By utilizing extensive parallel and vector processing units, modern graphics hardware dramatically outperforms the most advanced CPU. As a result, a growing interest has been raised in performing general purpose GPU computation, namely GPGPU. GPGPU's ultimate goal is to enable the GPU as a powerful coprocessor to CPU and offload computationally intensive tasks. GPU algorithms for dense matrix multiplication [1, 10, 12], FFT [15], database operations [9], sparse matrix conjugate gradient solver [4], ray tracing [6, 17], etc. have been studied and demonstrated to work on graphics hardware.

In order for general purpose computing to fully utilize the power of graphics hardware, it is critical to tune software to cater to the underlying architecture. Tuning software performance for a particular hardware architecture usually requires detailed knowledge of that architecture. However this requirement is difficult to meet for graphics hardware due to the following reasons: First, most GPU vendors do not release their products' architectural internal details, such as cache organization, rasterization algorithm. Second, programmers have only indirect control of the hardware through a vendor supplied driver, which dynamically loads, compiles, optimizes and executes the application supplied programs. The optimizations done by the dynamic compiler inside the driver are transparent to the programmer. Third, graphics hardware evolves fast. Every six months, GPU vendors introduce a new generation of graphics cards. A well tuned program for a particular generation of architecture may turn out to perform badly on its successor generation due to changes in the underlying architecture.

The difficulty in tuning performance for a fast-evolving hardware architecture makes self-adaptive software desirable. Automatic software tuning for general purpose processors has been studied for some years. Previous research in this field centered around automatic generation of high-performance numerical routines, such as dense and sparse matrix operations [3, 11, 20], sorting [13], FFT [8], signal processing [18], by improving software's spatial/temporal locality, instruction scheduling and dynamic algorithm selection to cater to modern processors' deep memory hierarchy and pipeline design.

Automatic tuning for graphics hardware presents new challenges. First, graphics hardware uses a non-traditional programming model that complicates the mapping of algorithms to the hardware. Second, as graphics hardware is vastly different from general purpose processors, new tuning strategies are needed. Since performance tuning for modern CPU's has been well studied, programmers are familiar with common optimization techniques such as cache blocking, register tiling, software pipelining, loop unrolling, etc. However, these techniques rarely directly work for GPUs. Third, due to the fact that graphics hardware's architectural details and machine parameters

[1] In this paper, "graphics hardware" and "GPU" are used interchangeably.

are usually withheld by vendors, the use of performance models either to prune search space of automatic tuning or to replace search is more difficult to realize on graphics hardware.

To our knowledge, this paper is the first attempt to implement an automatic tuning system to generate numerical libraries for graphics hardware. More specifically, it studies automatic generation of high-performance matrix multiplication on graphics hardware, as matrix multiplication is the most important building block for a variety of numerical libraries. In contrast to ATLAS [20], which utilizes register tiling, cache blocking and instruction scheduling to achieve high performance on pipelined processor with deep memory hierarchy, our approach automatically tunes matrix multiplication to graphics hardware's unconventional architecture features, such as SIMD instruction with swizzling and smearing, multiple-render-targets, limited instruction count, limitation on branch instruction, varying shader models, etc. Our automatic tuning system is capable to generate matrix multiplication implementations with comparable performance to expert manually tuned version despite the significant overhead incurred due to the use of a high level language.

The remainder of this paper is organized as follows: section 2 introduces related research. Section 3 introduces some special features of modern graphics hardware architecture. Section 4 describes algorithms for matrix multiplication on graphics hardware. Section 5 presents in details the automatic tuning system for matrix multiplication on graphics hardware. Section 5 explains and analyzes the performance data. The paper is concluded by section 7 with future research.

2 Related Work

In this section, we introduce some representative automatic tuning systems for microprocessors in the field of dense/sparse matrix operations, signal processing and sorting. We also briefly survey previous research on matrix multiplication on graphics hardware.

PHiPAC [3] is an early prototype of an automatic matrix multiplication generation system. ATLAS [20] extends PHiPAC's idea to all of the other dense matrix kernels that constitute the Basic Linear Algebra Subprograms (BLAS). Both projects employ parameterized code generators that can generate multiple versions of matrix multiplication code according to input tuning parameter values. These tuning parameters control different software transformations that affect L1-cache blocking, register tiling, instruction scheduling. The generated code's performance is empirically evaluated by actual execution. A search engine is then used to search over the implementation space for the version that yields the best performance. An alternative approach to empirical-search based tuning is to use analytical model to determine the best tuning parameter values [21].

FFTW [8] is the first automatic tuning system to generate one or multi-dimensional complex Fourier transformations. It employs a high-level description of execution plan for decomposing large Fourier transform into smaller specially optimized kernels, named "codelet". A dynamic programming based search process is performed at runtime, when input transform size is known, to find the best execution plan. Extending FFTW's idea to more general signal processing, the Spiral [18] system is built on top of a symbolic algebra system. It allows users to enter customized transforms in an interpreted environment using a high-level tensor notation, and uses a novel search method based on genetic algorithms.

Sorting and sparse matrix operations are two examples of applications which need to be tuned not only to the architecture but also to input data's characteristics. Li et al. [13] use a genetic algorithm and a classifier system to produce a hierarchically-organized hybrid sorting algorithm that adapts to input data characteristics, and has better performance than carefully tuned commercial sorting libraries. Sparsity project [11] automatically tunes sparse matrix operations to both the architecture and the sparse matrix's non-zero structure. It combines traditional techniques such as loop transformations with data structure transformations and optimization heuristics that are specific to sparse matrices. It provides a novel framework for selecting optimization parameters, such as block size, using a combination of performance models and search.

As one of the most important building blocks for numerical libraries, dense matrix multiplication on graphics hardware has attracted great attention since the appearance of programmable graphics hardware. Larsen et al. [12] first presented a single-pass matrix-multiplication algorithm for graphics hardware. Moravánszky [1] and Hall et al. [10] introduced two algorithms, which extended Larsen's algorithm to utilize graphics hardware's SIMD instruction with swizzling and smearing by data packing. Fatahalian et al. [7] thoroughly studied the performance efficiency of previously proposed algorithms on a variety of graphics hardware and reached the conclusion that due to the limit of cache-to-processor bandwidth, it is not possible to fully utilize the tremendous computing power of graphics hardware without changing the underlying architecture. We will describe in more details the above algorithms in section 4.

3 Graphics Architecture Features

In this section, we introduce the graphics hardware's special features that are not found in conventional microprocessors and are relevant to our work. Readers interested in deeper treatment of graphics hardware architecture are referred to [2, 14] and to vendor specifications of graphics hardware products.

Most modern graphics hardware has multiple vertex processors and fragment processors. Figure 1 depicts a conceptual view of a graphics hardware with sixteen fragment processors and six vertex processors[2]. Vertex processors perform geometric

[2]Note that figure 1 ignores many graphics related components. It does not represent any real

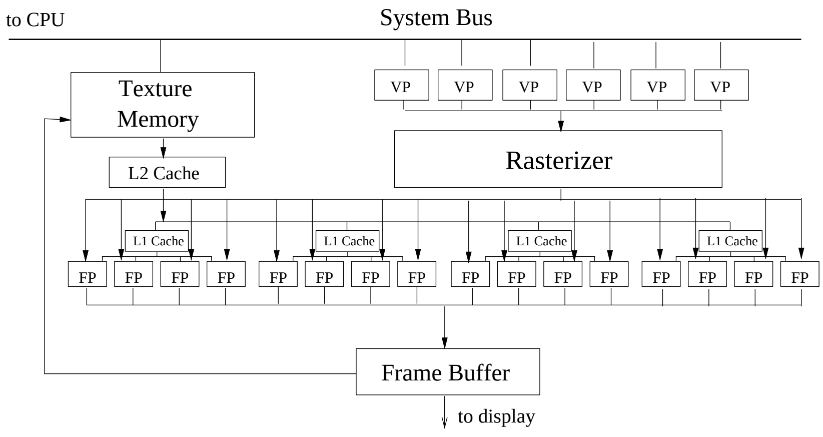


Figure 1: Architecture Model of Graphics Hardware

transformations and lighting operations on geometric primitives. After vertices have been projected to screen space, the rasterizer calculates fragment[3] information by interpolating vertex information. Then rasterizer assigns fragment-rendering tasks to fragment processors. A fragment processor renders one fragment at a time. After a fragment has been rendered, the fragment processor writes the final color information into the fragment's designated location in the frame buffer for display.

The graphics hardware's memory subsystem, namely texture memory, is mainly designed to support texture mapping operations. Since texture mapping only requires fragment processors to read from the texture memory, most modern GPUs do not support write operations by fragment processors to the texture memory. Fragment processors can only perform writes to the frame buffer. If a program needs to store intermediate results to texture memory, it can either copy the intermediate results from the frame buffer to texture memory, or use a render-to-texture technique, which allows rendering results in the frame buffer to be used as input texture for further computations. Since texture mapping is performed in the fragment processors, most modern GPUs do not allow vertex processor to access texture memory [4].

In figure 1, four fragment processors share one L1 cache. The L2 cache is shared by all sixteen fragment processors. Data organization in graphics hardware's cache, namely texture cache, is also designed to improve spatial locality of texture mapping operation. Optimizations for temporal locality of texture mapping are implemented in the rasterizer by rasterizing fragments in some special order. As cache organization and rasterization algorithms used in GPU products are usually considered as commercial secrets, there is little public knowledge about their internal details.

graphics hardware architecture but only serves to facilitate the understanding of general purpose computing on GPU.

[3]In graphics terminology, "fragment" refers to screen element before shading, "pixel" refers to screen element after shading.

[4]Latest GPUs start to add the support for vertex processors to access texture memory.

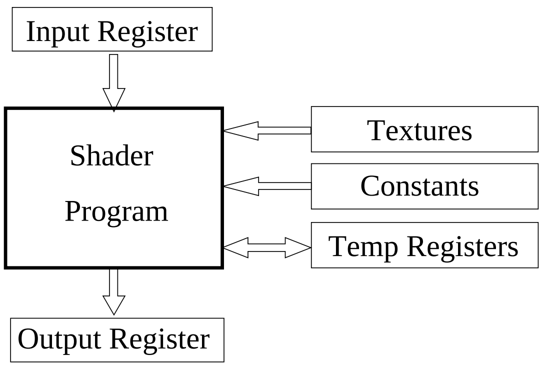


Figure 2: Programming model for GPU

Due to vertex processors' inability to access texture memory and the rasterizer's lack of programmability, most GPGPU applications rely on fragment processors to perform intensive computation. A program executed by a fragment processor is called "fragment program" or "fragment shaders"; these are used interchangeably in the context of this paper. Each execution of a "fragment program" renders one fragment. Therefore, one can consider a GPU as a stream processor, which performs the same kernel function (fragment program) on streams of data elements (fragments).

The programming model for fragment processors is illustrated in figure 2. A fragment program reads input data from input registers filled by the rasterizer. It can read a number of constant values set by the host application, read from texture memory, and read and write a number of temporary registers. After the execution, the result in the output registers is written into corresponding positions in the frame buffer for display.

We describe below several additional features of fragment processors that are not found in conventional microprocessors and that are relevant to our work.

SIMD instructions with swizzling and smearing. Fragment processors support four-way SIMD instructions. Each register has four components corresponding to four color channels of a pixel (RGBA). Color channels can be permuted, which is called "*swizzling*", and can be replicated, which is called "*smearing*". In the following code, register R1 is used with "*swizzling*", register R0 is used with "*swizzling*" and "*smearing*".

```
R2=R1.abgr * R0.ggab
```

Branch instruction. Early graphics hardware either does not support shaders with branches, or supports branches indirectly through predicated instructions or loop-unrolling. Latest graphics hardware starts to support dynamic branch instructions. However, using dynamic branch instructions can cause expensive performance penalties.

Instruction count. Most graphics hardware has limit on the static number of instructions a shader can contain. With branch instructions, it is possible that dynamic instruction count is vastly higher than static instruction count. Some graphics hardware may have limit on dynamic instructions executed by a shader.

Outputs per shader. A shader is only able to output to designated pixels in the frame buffer. With the introduction of multi-render-targets support in latest graphics hardware, a shader is capable to write to a limited number of auxiliary buffers in addition to the frame buffer.

4 GPU Algorithms for Matrix Multiplication

In this section, we present some matrix multiplication algorithms for GPU. To begin with, we show the naïve three nested loop algorithm for multiplying two matrices on CPU (assuming matrix C is initialized to zero).

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
         for (k=0; k<L; k++)
            c[i][j] += A[i][k] * B[k][j]
```

Larsen et al. [12] first described an algorithm to map the above code onto GPU. Basically, they propose to store matrix A and matrix B as two textures and to compute the result matrix C in the frame buffer. The shader program fetches one row from texture A and one column from texture B, computes the dot product, and stores the result into the frame buffer.

There are several caveats to this scheme. First, it fails to utilize the SIMD instructions of the fragment processor. Second, no data reuse is exploited. As matrix multiplication performs $O(n^3)$ operations on $O(n^2)$ elements, exploiting data reuse can significantly increase the computation to memory access ratio, thus resulting in better register and cache usage and improved performance. Third, on fragment processors that do not support dynamic branch instructions, the dot-product computation needs to be fully unrolled, which can easily exceed the instruction count limit when the matrix is large.

To address those problems, Moravánszky [1] and Hall et al. [10] proposed two multi-pass algorithms with data packing. Multi-pass techniques essentially use strip-mining loop transformations to decompose the k-dimension loop into two nested loops. The shader calculates a partial sum for an element of C. The outer-most loop accumulates partial sums into the correct result for a C element. Data packing is used to pack four elements of a matrix into four color channels of a texel (texture element) so that each memory access operation can load/store four matrix elements instead of just one element in Larsen's algorithm. By packing four elements in one register, the fragment processors are able to execute SIMD instructions.

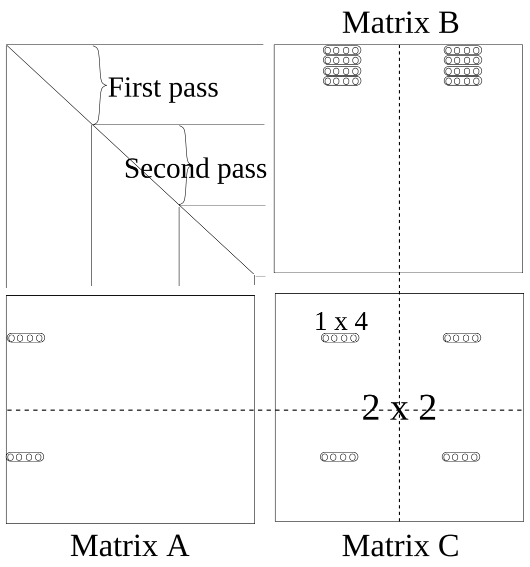


Figure 3: Matrix multiplication with MRT and Data packing

However, Hall [10] and Moravánszky [1] propose different data packing schemes. Hall uses 2×2 scheme, which packs four elements from four sub matrices of the original matrix. Whereas Moravánszky uses 1×4 scheme, which packs four consecutive elements into one texel.

The 2×2 scheme allows each element loaded from memory to be used twice in the shader. Thus, each execution of the shader reads from two rows of matrix A and two columns of matrix B, and produces four elements of matrix C. The 1×4 scheme reads from one row of matrix A and four columns of matrix B to generate four elements for matrix C. The shader performs a few 1×4 vector by 4×4 matrix products. Hence, elements from matrix A are used four times, whereas elements from matrix B are not reused.

Data packing not only enables SIMD instruction but also improves data reuse in the GPU. In this paper, we propose another technique that can further improve data reuse beyond the previous two algorithms. The technique is based on multiple-render-targets (MRT), which is supported in the latest graphics hardware. MRT allows a shader to write multiple results. One of the results is written to the frame buffer, the others are written to a number of auxiliary buffers. Figure 3 illustrates a multi-pass matrix multiplication algorithm with the 1×4 data packing scheme and 2×2 MRT scheme.

MRT based matrix multiplication algorithms naturally extend data-packing based algorithms. The idea is to divide matrix C into $m \times n$ blocks of sub-matrices. One of them will be assigned to the frame buffer, the other sub-matrices are distributed to auxiliary buffers. $a \times b$ data-packing based algorithms effectively performs strip-mining loop transformation on the i and j loops by factors a and b. The $m \times n$ MRT based matrix multiplication further strip-mines the resulting i and j loops by m and

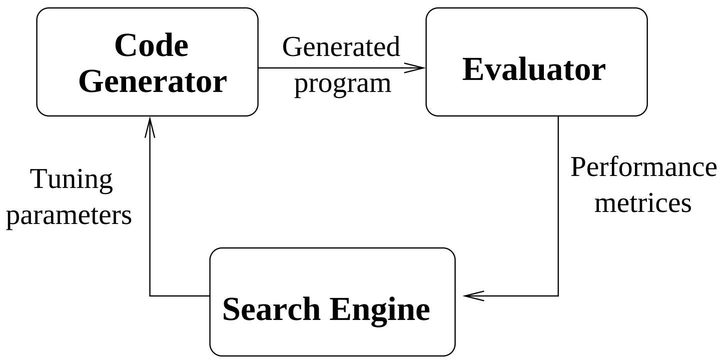


Figure 4: Components of automatic tuning

n. With MRT, elements loaded from matrix A can be reused n times further after data packing, elements loaded from matrix B can be reused m times further after data packing.

5 Automatic Tuning System

Typically, automatic tuning approach involves three components as shown in figure 4. A code generator inputs the values of tuning parameters and outputs the program version that is specified by these parameters. An evaluator empirically evaluates the performance metrics of the generated code and feeds back the metrics to search engine. A search engine searches over the implementation space by controlling the tuning parameter values fed into the code generator according to some search strategy. We will elaborate our tuning system with regarding to figure 4.

5.1 Code Generator

The code generator encapsulates several degrees of freedom in restructuring and transforming program to generate different implementations of the same algorithm. In designing our code generator, we adopt similar strategy as ATLAS [20]. We focus on tuning the kernel routine for multiplying 1024×1024 matrices. Input matrices are first divided into blocks of 1024×1024 sub matrices. Then the matrix multiplication is performed in terms of multiplying the block matrices with the tuned kernel routine. Matrices of size not multiples of 1024 will result in clean-up code, which can be executed either by the CPU or by the GPU with code generated similarly. We choose the particular value of 1024 because it yields the best performance.

The other issue to address in designing our code generator is what programming language generated programs should be coded into. There are three levels of programming language that can be used to program graphics hardware: assembly level shading languages such as the "ARB_fragment_program" extension to OpenGL, high-level shading languages such as the Cg language [16] from nVidia, and high-level general purpose languages such as BrookGPU [5].

Assembly level shading languages require programmers to explicitly manipulate registers and arithmetic instructions. Assembly programs are encoded in string variables in C/C++ programs that use OpenGL or Direct3D commands to maintain graphics pipeline states and load/execute the assembly program.

High level shading languages like Cg and HLSL allow shaders to be written in C-like language. Similar to assembly program, the C-like shading programs are encoded in string variables in C/C++ programs that use OpenGL or Direct3D and the high level shading language's runtime library to explicitly control the graphics hardware pipeline and to load/compile/execute the high-level shading code.

High-level general purpose languages go one step further by hiding the graphics hardware characteristics. Programs written in the BrookGPU language are first source-to-source translated into C++ programs containing fragment programs coded in Cg that appear as string variables and wrapper C++ code for setting up and maintaining graphics pipeline states. From this point, on top of the BrookGPU's runtime library, the generated C++ program will execute just as a normal graphics program written in C++ with fragment programs encoded in Cg.

We decided to generate programs in the highest level language, specifically the BrookGPU language, mainly for two reasons. First, generated code should be portable to various architectures, even future architectures that are not yet defined. Generating high level program will permit fundamental changes in hardware and graphics API as long as the compiler and runtime library for the high level language keep up with those changes. Whereas, code generated in assembly language or Cg language is tied to particular generation of hardware and may need to be modified to utilize new features of the hardware or graphics API. Second, implementing the code generator is a very tedious and error-prone process. The generator is easier to debug when its output is high-level code. The downside of this decision is that the code compiled from BrookGPU is less efficient than manually generated code. One can hope that as high level languages for GPUs and their associated compilers and run-time libraries mature, the performance penalty for the use of high level languages will shrink or disappear, as it happened with conventional processors.

Our code generator is implemented in the Python script language to generate BrookGPU programs according to input tuning parameter values.

5.2 Tuning Strategies and Parameters

Tuning strategies are heuristics of restructuring or transforming program to improve the overall performance of the generated code. They have associated tuning parameters that control various aspects of the generated code and embody the tuning strategy. Usually, the optimum values of these tuning parameters are platform-specific and therefore can not be determined a priori. This leads to the need for empirical evaluation based search. In this subsection, we describe the tuning strategies and their associated tuning parameters for our automatic tuning system.

5.2.1 Tuning Multi-Render-Targets

Today's most advanced GPU offers up to three auxiliary buffers in addition to the frame buffer known as multiple render targets (MRT). The MRT strategy can help improve data reuse and therefore reduce the number of memory accesses and improve performance. However, MRT necessitates the copying of intermediate results stored in auxiliary buffers into texture memory for further passes. Furthermore, MRT requires more temporary registers by the shader, which reduces the performance of the fragment processors. Hence the optimal scheme to decompose matrices to use MRT needs to be tuned to the target platform.

[**mrt_w**, **mrt_h**]: The matrix C is divided into $mrt\_w \times mrt\_h$ sub-matrix blocks to utilize MRT. The valid values for these two parameters are limited by the number of auxiliary buffers supported in hardware. Since latest hardware supports up to 3 additional buffers, the possible values of these two parameters range over 8 cases, which have the product of mrt_w and mrt_h less or equal to 4.

5.2.2 Tuning Data Packing

This is the strategy of utilizing SIMD instructions with data packing. As introduced in section 4, the two data-packing schemes 1×4 and 2×2 have different advantages and disadvantages. Our automatic tuning system relies on actual execution to decide which one is better on target platform.

[**mc_w**, **mc_h**]: Tuning parameters "mc_w" and "mc_h" decide how to pack consecutive elements into one texel. $mc\_w \times mc\_h$ block of elements are packed into one texel. As there are only four available channels (RGBA) for each texel, the product of mc_w and mc_h must be less or equal to 4. Hence, there are totally 8 cases.

5.2.3 Tuning Number of Passes

It would be nice to have a long shader that calculates the resulting matrix C in one pass, which can eliminate the expensive texture-copy or render-to-texture operation for intermediate results. However, due to fragment processor's limit on instruction count and temporary registers, a shader can not be too long. Even within the valid range of instruction count limit, longer shader may perform worse than shorter shader. As a result, the number of k-loop iterations to be executed in a shader needs to be tuned to the target platform.

[**np**] Tuning parameter "np" determines how many iterations in k-dimension loop are executed by the fragment shader. We observed from experiments that np larger than 256 is either not supported by hardware or has already started to suffer from

performance penalty. Hence, in our tuning system, we limit the range of *np* from 1 to 256.

5.2.4 Tuning for Branch Instruction

Latest graphics hardware adds support for dynamic branch instruction. This allows implementing a loop-based shader without having to fully unroll it as is the case for earlier generation of graphics hardware, which does not have dynamic branch instructions. Using loop-based shader could help reduce static instruction count. However, as branch instructions come with an expensive performance penalty, whether to use branching or loop unrolling needs to be tuned on actual hardware.

[**unroll**] Tuning parameter "*unroll*" decides whether or not to use branch instruction to implement a loop-based shader. The valid values of *unroll* are either 0 or 1. If *unroll* equals 1, the inner loop of the generated code will be fully unrolled.

5.2.5 Other Tuning Parameters

[**compiler**] BrookGPU resorts to either "cgc" or "fxc", which are compilers from nVidia's Cg Toolkit and Microsoft's DirectX9 respectively, to compile Cg program into assembly fragment program. Since these two compilers might perform different optimizations, the generated code might execute differently. We use a tuning parameter "*compiler*" to determine which compiler to use to compile shader. The valid values of *compiler* are either "cgc" or "fxc".

[**profile**] This tuning parameter originates from the options of shader models in interfacing with graphics hardware. Currently, there are two popular graphics API's, namely Direct3D and OpenGL. They provide somewhat equivalent functionalities through different programming API. BrookGPU is able to use either of them as the back end API to interact with GPU. For both Direct3D and OpenGL, there are several shader profiles. Specifically, Direct3D has four shader profiles, "$ps20$", "$ps2a$", "$ps2b$" and "$ps30$". OpenGL has three profiles "arb", "$fp30$", "$fp40$". The profiles provide different capabilities to shader programs. For example, "$fp40$" and "$ps30$" support dynamic branch instruction. Also, different profiles have different limits on instructions count and number of temporary registers. We use a tuning parameter "$profile$" to choose among back-ends and shader models. The valid values of $profile$ are "$ps20$", "$ps2a$", "$ps2b$", "$ps30$", "arb", "$fp30$" and "$fp40$".

5.3 Performance Evaluator

The performance evaluator uses MFLOPS (million floating point operations per second) as the performance metric to evaluate the quality of generated code. Graphics hardware typically supports fused multiply-add instruction, which allows a *multiply* and an *add* to complete in one instruction. In order to compare with matrix multiplication performance on a CPU, we consider this single instruction operation as two floating operations. The performance evaluator returns zero MFLOPS for invalid generated programs, e.g. programs that exceed the instruction count limit.

5.4 Search Engine

The search engine is responsible for searching over the implementation space to find the version with the best performance. As optimization in multi-dimensional discrete space is generally an NP-hard problem, hence there is no general algorithm that can solve this discrete optimization without exhaustive search. In our case, exhaustive search over all possible versions would require $8 \times 8 \times 256 \times 2 \times 2 \times 7 = 458752$ evaluations. If each evaluation takes 10 seconds, the whole search would take 53 days, which may not be acceptable.

We implement an ad-hoc search algorithm specifically for our tuning problem. We employ two techniques to limit our search to around four hours without sacrificing too much performance. The first technique is to employ some problem specific heuristics to prune the search space of tuning parameters. The second technique is to search tuning parameters in some predetermined order to effectively decompose the high-dimensional search space into multiple lower-dimensional spaces.

5.4.1 Space Pruning

According to the symmetric property of mc_w and mc_h parameters, we impose an additional constraint that $mc\_w \leq mc\_h$. Since the matrix size is 1024×1024, we also limit mc_w and mc_h to powers of two. Now we have only four possible cases $(mc\_w, mc\_h) \in \{(1,1),(1,2),(1,4),(2,2)\}$. Similarly mrt_w and mrt_h can be limited to four cases: $(mrt\_w, mrt\_h) \in \{(1,1),(1,2),(1,4),(2,2)\}$.

Parameter np decides the number of iterations in k-loop to be executed in the shader. Intuitively, as np increases, the performance will first improve due to fewer number of passes. When np exceeds some optimum value, the instruction count issue and excessive use of temporary registers start to outweigh the benefits of fewer passes.

The search problem essentially boils down to finding the maximum value of a unimodal function. Theoretically, the best algorithm has complexity of $O(log(n))$. However, for our particular problem, since the np value range is rather small and we believe the optimum np value should be near power of two values, we designed algorithm 1 for finding optimum np value. The idea of algorithm 1 is to exponentially increase stride and evaluate the performance at corresponding np until either the end of interval is reached or the performance is less than the minimum of the previous two evaluated performances. The procedure is then recursively invoked on both sides of the best np found in the exponential search until the length of interval is less or equal to a predetermined threshold. Its theoretical worst case complexity complies with the following recursion.

$$f(n) = f(\frac{n}{2}) + f(\frac{n}{4}) + log(n)$$

Solving this recursion gives the algorithm's worst case complexity of $O((log_2 n)^{\frac{\sqrt{5}+1}{2}})$. In our tuning system, it is often the case that the loop at step 8 of algorithm 1 is exited prematurely because performance goes below the previously evaluated two values. Therefore, algorithm 1 practically has better performance than generic $O(log(n))$ algorithms for our problem.

Algorithm 1 Finding Optimum *np*

```
Input: $start$ – starting value of $np$ in the interval
       $length$ – length of the interval
       $direction$ – left(-1) or right(1)
Output: update global variable storing the best $np$

procedure $find\_np(start, length, direction)$
1: if ($length \leq threshold$) return;
2: Initialize $p$, $last\_two$, $max\_mflops$, $best\_np$
3: repeat
4:     Evaluate $mflops$ at $np = start + direction * p$
5:     if ($mflops > max\_flops$)
6:         update $max\_mflops$, $best\_np$
7:     exponentially increase stride $p$
8: until out of range or performance $\leq min$(last_two).
9: $find\_np(best\_np, left\_size, left)$
10: $find\_np(best\_np, right\_size, right)$
11: return;
```

5.4.2 Search in Phases

In addition to space pruning, search in phases can further reduce the search space by decomposing the high dimensional space into several lower dimensional spaces. The assumption is that the optimal values of some tuning parameters are independent of each other, so that we can search the best values for some tuning parameters while fixing the others. Formal proof of independence relationship between parameters of multi-variate function is difficult. In our case, from experiment results, we speculate *np* parameter is independent of mc_* and mrt_* parameters to some extent, therefore we decouple the nested search for *np* and mrt_*, mc_* into a sequential search. Algorithm 2 describes the search order of tuning parameters we use in our tuning system. The search for *np* parameter is further divided into two stages. In step 4, only power of two values are searched. In step 8, after mc_* and mrt_* are determined, algorithm 1 is applied to pin down the best *np*.

After applying the above two techniques, the typical evaluation running time reduces to around 4 hours.

Algorithm 2 Search Order of Tuning Parameters

```
1: For each compiler value
2:     For each profile value
3:         For each unroll value
4:             Search np in power of two values
5:             For each mc_* value
6:                 For each mrt_* value
7:                     Evaluate Performance
8:             Recursively search np in both sides of
               best np found in step 4.
```

6 Performance Evaluation

We run the automatic tuning system on four graphics cards. Their configurations are given in table 1. The host CPU and operating system are 2.6Ghz Pentium 4 and Windows XP.

	G6800U	G6800G	QF3400	G5800U
Model Name	GeForce 6800 Ultra	GeForce 6800 GT	Quadro FX 3400	GeForce FX 5800 Ultra
Pixel Processor	16	16	16	4
Core Frequency	400 MHz	350 MHz	350 MHz	500 MHz
Mem Frequency	1100 MHz	1000 MHz	900 MHz	1000 MHz
Mem Width	256 bit	256 bit	256 bit	128 bit
Bandwidth	35.2GB/s	32.0GB/s	28.8GB/s	16GB/s
Driver	6693	7568	6176	6693
GPU	NV40	NV40	NV45GL	NV30
DirectX	9.0c	9.0c	9.0c	9.0c
OpenGL	1.5.2	2.0.0	1.5.1	1.5.2

Table 1: Four GPU platforms

We conducted experiments only on nVidia cards. We did not test on ATI cards mainly because no ATI cards truly support 32-bit floating point. The most advanced ATI cards, ATI Radeon X800 XT only supports 24-bit floating point data operations in pixel processors.

We benchmarked the performance of multiplying two matrices of size 1024×1024, whose elements are randomly generated. Timing operations performed by a GPU is difficult because the GPU and the CPU work asynchronously. We work around this problem by measuring the time from the start of the multiplication until one element of the result matrix is read from the GPU to the CPU. This potentially could involve an overhead of moving large matrices between GPU and CPU and the serial overhead of setting up graphics pipeline states. In order to reduce the impact of this overhead, we force the GPU to perform the same matrix multiplication operation ten times and use the average as the execution time. Our experiments show that the overhead is typically below 10% of measured performance, and the error range of measured performance is below 3%.

6.1 Manually Tuned Implementation

Fatahalian et al. [7] thoroughly studied the performance efficiency of matrix multiplication algorithms on a variety of graphics hardware and presented two hand-tuned implementations as the most efficient implementations. They included these two implementations in GPUBench [19], which is a benchmark suite designed to analyze performance of programmable graphics processor for GPGPU. To test the effectiveness of our automatic tuning approach, we compare the performance of our automatic tuned version with these two expert hand-tuned implementations.

Table 2 summarizes the high level structure of these two implementations in terms of our tuning parameters described in section 5.2. We use the same names "NV_Single" and "NV_Multi" as in [7] to refer to these two implementations.

	mrt_*	mc_*	np	unroll	profile	compiler
NV_Single	1×1	2×2	128	1	fp30	NA
NV_Multi	1×1	1×4	6	1	arb	NA

Table 2: High level structure of the two implementations

It is important to note that these two implementations are implemented in C++ and OpenGL API with fragment program written in carefully crafted assembly code. Whereas, our automatic tuning system generates high level BrookGPU code. The generated BrookGPU code is first translated into C++ code with fragment programs encoded in Cg language, which in turn will be compiled into lower level assembly code by "cgc" or "fxc" compiler according to chosen shader model. The graphics pipeline in generated BrookGPU code is transparently managed by BrookGPU language's runtime library to provide a high level generic programming model. As we will see later, the difference in implementation level has significant impact on the performance.

6.2 Experiment Results

In this subsection, we present the experiment results. We first compare the performance of our automatically generated matrix multiplication implementations with the manually tuned versions on four platforms. Then we study the sensitivities of the tuning parameters to overall performance. In all figures shown in the subsection, Y axis represents MFLOPS (million floating point operations per second).

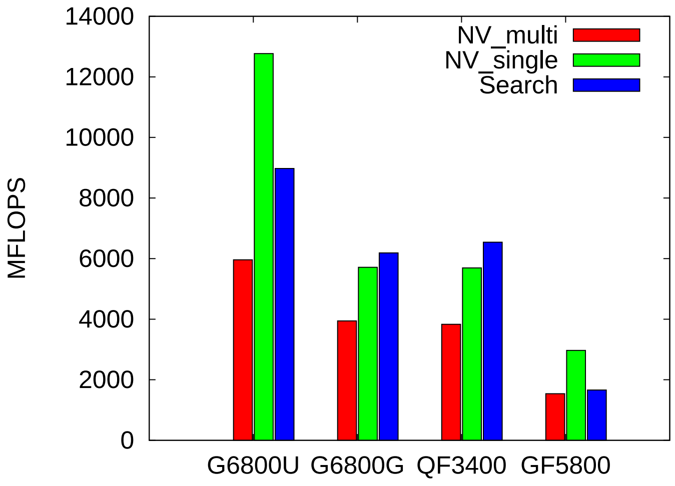


Figure 5: Performance on four platforms

6.2.1 Automatic Vs. Manual

Figure 5 shows the performances of the two hand-tuned implementations and the automatically tuned version, which is denoted as "Search", on the four platforms. As we can see, "NV_multi" consistently performs the worst among the three implementations. Between "NV_Single" and "Search", on G6800U and G5800U, "Search" achieves 70%, 56% of the performance of "NV_Single". On G6800G and QF3400, "Search" achieves 8% and 15% speedup over "NV_Single" respectively.

This result might look surprising, because both of the hand-tuned implementations are within the search range of automatic tuning. The reason for the lower performance of "Search" is the overhead associated with using the high level BrookGPU language. We found some inefficiencies in the BrookGPU's runtime system and "cgc"/"fxc" compilers. For example, instead of using "render-to-texture" technique, BrookGPU's OpenGL backend uses the expensive copy operation to move intermediate results from the frame buffer to texture memory. Also in dealing with array-indexing operation, BrookGPU seems to generate auxiliary instructions to map index values to texture coordinates. The addition of extra instructions compared to carefully crafted assembly code would hurt performance. We also suspect that "cgc" and "fxc" compilers' register allocation strategy is not optimum for some cases. For instance, when a loop is unrolled, occasionally the compiler fails to reuse registers across unrolled iterations of the loop, which greatly increases the pressure on registers and limits the ability of unrolling loop to improve performance.

In order to roughly measure the performance overhead of using high level BrookGPU language, we compare the performance of "NV_Single" with the performance of its counterpart implementation in BrookGPU. We force the code generator to generate implementations in BrookGPU with the same *mrt_∗*, *mc_∗*, *unroll*, *profile* values

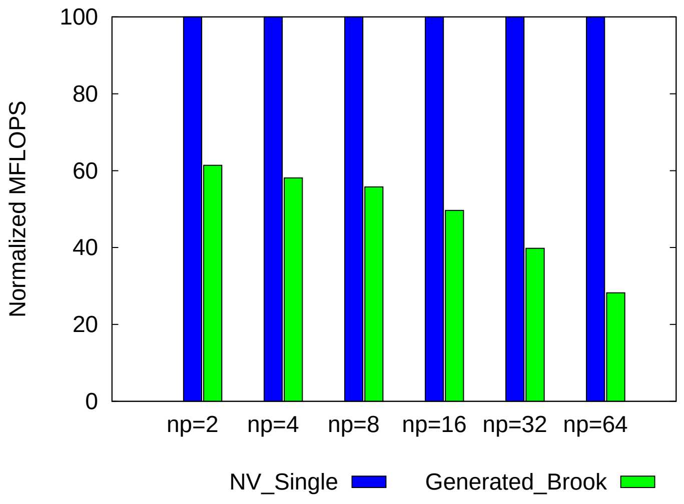
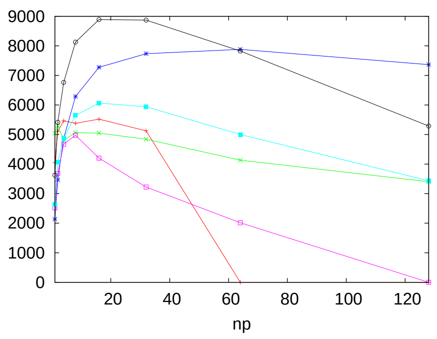


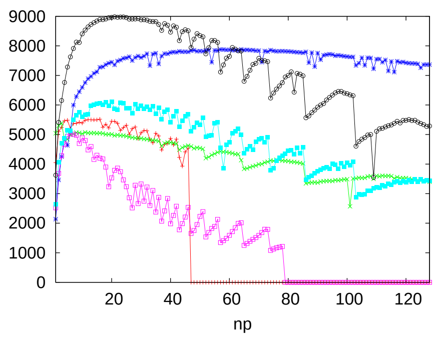
Figure 6: Performance penalty associated with runtime library and compiler optimization

as "NV_Single" 's corresponding values in table 2. For *compiler* parameter, since "NV_Single" is implemented in assembly fragment code, there is no corresponding *compiler* value for it. We use the *compiler* value with the best performance. We vary the *np* tuning parameter from 2 to 64 in power of two values. We choose this range because "NV_Single" does not support $np = 1$ case and larger *np* in power of two values will exceed the instruction limit. Figure 6 , which is based on data collected on G6800U platform, shows the relative performance of the "NV_Single" and its counterpart implementation in BrookGPU. As can be observed, due to the overhead of using high level BrookGPU language, the generated BrookGPU version never reaches more than 60% of the performance of "NV_Single". As *np* increases, the relative overhead also increases. We don't fully understand the reason. We suspect the reason has to do with the added array-indexing instructions, which increase the dynamic instruction count and the use of more active registers.

If we take into account the performance overhead due to using the high level BrookGPU language, the performance achieved in figure 5 is satisfactory. On two platforms, the automatic tuned version can even outperform the hand-tuned version in spite of the significant overhead. This is mainly because "NV_Single" was specially tuned for graphics hardware similar to "GeForce 6800 Ultra" graphics card. When changing to other platforms, the performance of "NV_Single" is far from optimum. This testifies the benefit of automatic tuning system to adapt to changing underlying architecture. Since BrookGPU system is still a prototype research project, we believe its implementation has good potential for improvement.



(a) **Power of two values.**



(b) **All values**

Figure 7: Sensitivity of *np* parameter.

6.2.2 Parameter Sensitivity

In this subsection, we present the sensitivities of the tuning parameters described in section 5.2 to the overall performance.

Figure 7(a) shows the performance curves over power of two *np* values. These curves are searched in step 4 of algorithm 2. Different curves correspond to fixing the other parameters to different values. All curves have a single maximum. Figure 7(b) shows the performance curves over *np* ranging from 1 to 128. These curves are searched in step 8 of algorithm 2. As can be observed that there are some performance drops off the original curves in 7(a) at some particular *np* values. The performance curves recover from those drops gradually to the original curves. We don't understand the underlying reason for these performance drops, however, since the dropping points are not in power of two values, in most cases algorithm 1 can still find the global optimum as if the curve is a unimodal function.

Figure 8, which is based on data collected on G6800U platform, shows the sensitivities and interaction of mrt_* and mc_* parameters. As described in section 5.4.1, both mrt_* and mc_* range over $\{(1,1),(1,2),(1,4),(2,2)\}$. For each combination of mrt_* and mc_*, we tested five *np* values at $\{2,4,8,16,32\}$. On G6800U platform, $mc = 2 \times 2$ can achieve 2X to 2.5X speedup over $mc = 1 \times 1$. $mrt = 2 \times 2$ can further achieve 10% speedup over $mrt = 1 \times 1$. The optimum mrt_* and mc_* values are platform dependent.

For the "*unroll*" parameter, our experiments show that $unroll = 0$ is almost always better than $unroll = 1$. The reason is that for profiles that do not support branch instruction, the *fxc* and *cgc* compilers automatically unrolls the loop even if *unroll* is set to zero. For profiles that support branch instruction, the compilers determine whether or not to unroll the loop based on the length of the shader even if *unroll* is set to zero. Hence, generating high-level code with explicit loop-unrolling does not benefit performance in both cases.

For the *profile* parameter, we find in all of the four platforms we tested that

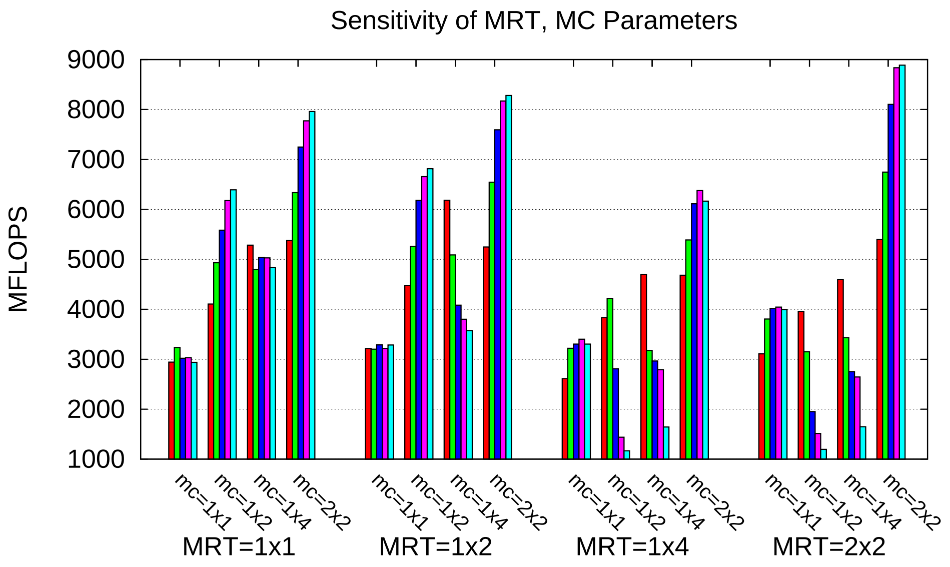


Figure 8: Sensitivity of MRT and MC parameters

profiles supporting more capabilities generally perform better than profiles supporting fewer capabilities. For example, for "DirectX" back end, performance increases in the order of "ps20", "ps2b", "ps2a", "ps30". For "OpenGL" back end, performance increases in the order of "arb", "fp30" and "fp40".

For the *compiler* parameter, we find both fxc and *cgc* generate codes of equivalent quality on all platforms.

7 Conclusion and Future Research

As graphics hardware advances and changes so rapidly, the ability of automatic tuning software to graphics hardware's architectural features will be essential for achieving good performance across a wide variety of architectures. In this paper, we present an automatic tuning system that can generate high-performance matrix multiplication implementation with comparable performance to hand-tuned version on a variety of graphics hardware. This paper identifies and employs some tuning strategies which are unique to graphics hardware. To our knowledge, it is the first attempt in automatic generation of high-performance numerical libraries for graphics hardware, and the results are encouraging.

For future research, similar automatic tuning approaches can be applied to generate broader library routines such as FFT, sorting, and linear algebra. Also, as BrookGPU is still a prototype research project, we believe it has good potential for improvement, which in turn can significantly benefit building similar automatic tuning systems for graphics hardware.

References

[1] Ádám Moravánszky. Dense matrix algebra on the gpu. 2003. http://www.shaderx2.com/shaderx.PDF.

[2] K. Akeley. Reality engine graphics. In *Computer graphics and interactive techniques*, pages 109–116. ACM Press, 1993.

[3] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *International Conference on Supercomputing*, Vienna, Austria, July 1997.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. *Proceddings of SIGGRAPH*, August 2004.

[6] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *ACM SIGGRAPH/EUROGRAPHICS Graphics hardware*, pages 37–46, 2002.

[7] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *ACM SIGGRAPH/EUROGRAPHICS Graphics hardware*, 2004.

[8] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation",.

[9] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, pages 215–226. ACM Press, 2004.

[10] 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, 2003.

[11] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[12] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55. ACM Press, 2001.

[13] X. Li, M. J. Garzaran, and D. Padua. Optimizing sorting with genetic algorithms. In *CGO'05*, pages 99–110. IEEE Computer Society, 2005.

[14] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. Infinitereality: a real-time graphics system. In *SIGGRAPH*, pages 293–302. ACM Press/Addison-Wesley Publishing Co., 1997.

[15] K. Moreland and E. Angel. The FFT on a gpu. In *ACM SIGGRAPH/EUROGRAPHICS Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[16] nVidia Corporation. NVIDIA Cg Toolkit. http://developer.nvidia.com/object/cg_toolkit.html.

[17] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *SIGGRAPH*, 21(3):703–712, July 2002.

[18] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, February 2004.

[19] Stanford Univ. Graphics Lab. GPU benchmark suite. http://graphics.stanford.edu/projects/gpubench.

[20] 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.

[21] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. A. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *PLDI*, pages 63–76, 2003.