A Code Motion Technique for Accelerating General-Purpose Computation on the GPU*

Takatoshi Ikeda, Fumihiko Ino, and Kenichi Hagihara

Graduate School of Information Science and Technology
Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{ikeda,ino,hagihara}@ist.osaka-u.ac.jp

Abstract

Recently, graphics processing units (GPUs) are providing increasingly higher performance with programmable internal processors, namely vertex processors (VPs) and fragment processors (FPs). Such newly added capabilities motivate us to perform general-purpose computation on GPUs (GPGPU) beyond graphics applications. Although VPs and FPs are connected in a pipeline, many GPGPU implementations utilize only FPs as a computational engine in the GPU. Therefore, such implementations may result in lower performance due to highly loaded FPs (as compared to VPs) being a performance bottleneck in the pipeline execution. The objective of our work is to improve the performance of GPGPU programs by eliminating this bottleneck. To achieve this, we present a code motion technique that is capable of reducing the FP workload by moving assembly instructions appropriately from the FP program to the VP program. We also present the definition of such movable instructions that do not change the I/O specification between the CPU and the GPU. The experimental results show that (1) our technique improves the performance of a Gaussian filter program with reducing execution time by approximately 40% and (2) it successfully reduces the FP workload in 10 out of 18 GPGPU programs.

1. Introduction

The GPU (graphics processing unit) [8] is a processing unit designed for accelerating compute-intensive visualization tasks, such as three-dimensional (3-D) rendering applications [3]. Recently, GPUs are rapidly increasing their performance beyond the Moore's law [25]. For example, nVIDIA's GeForce 6800 provides approximately 120 GFLOPS at peak performance, which equals to six 5-GHz Pentium 4 processors [24].

In addition to this attractive performance, recent GPUs provide flexible programmabilities, such as programmable internal processors, branch capability, and single-precision 32-bit floating-point operations based on the IEEE standard [29]. These newly added programmabilities allow us to use GPUs not only for traditional graphics applications but also for compute-intensive, general-purpose applications.

There are many projects reporting experiences in accelerating general-purpose computation on the GPU (GPGPU) [1, 26]. From the viewpoint of implementation strategy, these prior projects can be classified into two groups: hardware- and software-based groups.

The former group is based on a hardware acceleration strategy that makes use of graphics-specific components of the GPU. For example, Lengyel et al. [20] have presented an algorithm for robot motion planning using rasterization hardware. Hoff et al. [15] also make use of interpolation-based polygon rasterization hardware to compute Voronoi diagrams. Takizawa and Kobayashi [30] solve the problem of data clustering using the Z-buffer depth comparison. The key difficulty in this strategy is to map CPU algorithms and data structures onto the hardware components.

The latter group is based on the stream programming model [18] that uses programmable components of the GPU, namely vertex processors (VPs) and fragment processors (FPs). Because current FPs provide higher arithmetic performance than VPs, many GPGPU implementations use only FPs as a computational engine in the GPU [26]. This software-based strategy is more flexible than the hardware-based strategy, because it requires us to simply translate CPU programs into GPU programs using graphics APIs. Due to this simplicity, it is used for various scientific prob-

*This work was partly supported by JSPS Grant-in-Aid for Scientific Research for Scientific Research (B)(2)(16300006) and on Priority Areas (16016254).

1-4244-0054-6/06/$20.00

lems, such as physical simulation [13, 21] and numerical computation [5–7, 10, 12, 16, 17, 19].

Thus, many projects report successful results using the GPU. However, prior projects mainly focus on showing the performance gain against the CPU. Accordingly, the efficiency of GPU implementations has not been investigated well. Note here that most implementations are based on the stream programming model that uses only FPs. In this model, FPs may become a performance bottleneck in the GPU, because FPs and VPs are connected in a pipeline. Therefore, it is better to offload FPs to VPs in order to stream more data through the pipeline for higher performance.

In this paper, we present a code motion technique that addresses the above mentioned problem for acceleration of GPU applications. To reduce the FP workload, our technique moves assembly instructions appropriately from the FP program to the VP program. This code motion is carried out without changing the I/O specification between the CPU and the GPU, preventing us from modifying CPU programs. Our technique is applicable to assembly programs running on the GPU that supports Vertex Shader (VS) 1.1 [23] and Pixel Shader (PS) 2.0 [23], which are standards for GPU design.

The rest of the paper is organized as follows. We begin in Section 2 by introducing GPU architecture with its programming strategy. We then present our code motion technique in Section 3. Section 4 shows some experimental results and Section 5 introduces related work. Finally, Section 6 concludes the paper.

2. GPGPU: General-Purpose Computation on the GPU

To describe the details of our technique, we first show a brief overview of GPGPU: the underlying architecture, assembly language, and programming strategy. In the following discussion, we assume VS 1.1 and PS 2.0 as the target standards for our technique.

2.1. GPU Architecture

The original task of the GPU is a rendering task, which computes pixels on the screen by projecting polygonal objects [3] (usually triangles) located in the 3-D space. In order to accelerate this compute-intensive task, the GPU is structured in a rendering pipeline consisting of two different programmable processors: VPs and FPs.

Figure 1 shows an overview of the GPU architecture. VPs transform 3-D triangles into 2-D triangles by projecting their vertices onto the screen from the viewing point. In other words, this geometric transformation computes the position of vertices on the screen, so that determines the

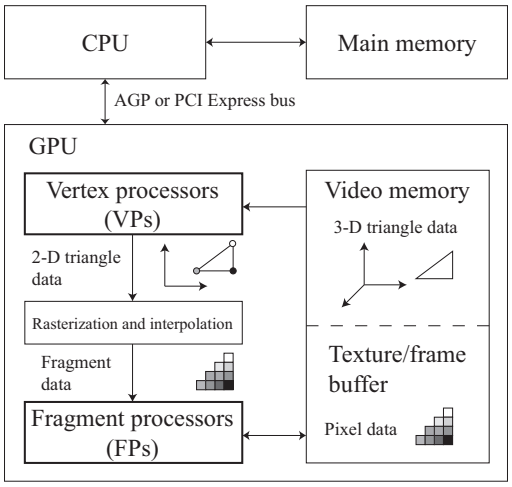

Figure 1. GPU pipeline architecture.

screen region in which FPs operate. Then, these 2-D triangles are rasterized into fragments for input to FPs. FPs take the responsibility for determining the final coloring of pixels. The details of VPs and FPs are as follows:

VP: VPs are based on a MIMD structure [11] in order to realize fast geometric transformation of vertices. Because this transformation is usually represented as a 4×4 transformation matrix, each processing element of VPs has a vector processing unit that is capable of rapid processing of 4×4 matrices.

The input to VPs, namely the polygonal data, must be transferred in advance from the main memory to the video memory. This data transfer is performed by a CPU program using graphics APIs such as DirectX [23] or OpenGL [28]. Then, the vertex data is set to input registers v# in VPs (see Table 1). By using other registers, VPs obtain transformation results, such as the vertex coordinates and colors, and then write them to output registers. Note here that output registers are not readable from VPs themselves.

FP: FPs are capable of rapid mapping of textures [3] (patterns) onto objects, aiming at increasing the realism of produced scenes. To do this, they obtain fragments from the rasterizer, and then execute some mathematical operations between the fragments and textures. FPs are based on a SIMD structure [11] that allows applying the same operation simultaneously to multiple fragments. Furthermore, as same as VPs, FPs support 4-length vector operations, because they deal with four-component RGBA data representing red, green, blue colors and opacity.

The fragment data is given by input registers v# and t# in FPs (see Table 1), which indirectly receive data from

output registers oD# and oT# in VPs, respectively. The texture data is read from the video memory using samplers s#, and the mapping results are written to buffers on the video memory through output registers oC# and oDepth.

In summary, an execution on the GPU can be represented as a pseudo code presented in Figure 2. The VP program is invoked on each vertex of 3-D triangles while the FP program is invoked on each fragment inside the 2-D triangles. The entire execution can be expressed as a doubly-nested loop, because rasterization inside the 2-D triangles generate fragments.

With regard to our code motion technique, the GPU architecture has three characteristics as follows.

- The input registers v# and t# in FPs hold linearly interpolated values of the output registers oD# and oT# in VPs, respectively. That is, because the former registers hold the attributes (for example, coordinates and colors) for a vertex, whereas the latter have those for a fragment, the input to FPs is automatically interpolated according to the relative position of fragments, as shown in Figure 1. This interpolation is automatically performed during rasterization. Moreover, registers v# (oD#) have lower precision than others. Each component in these registers is allowed to store 8 bits of unsigned data in the range [0,1], whereas registers t# (oT#) can store four 32-bits of signed data.

- VPs and FPs have different capabilities, because they do not process the same type of data: vertices and fragments. For example, FPs are capable of loading texture data from the video memory through samplers s#, but VPs do not have this capability. Thus, FPs have unique instructions and registers which VPs do not have.

- There is a limitation on the number of registers that pass data from VPs to FPs. This means that the information transmitted from a vertex to a fragment is limited by this number, because VP and FP programs are invoked on each vertex and fragment, respectively. In our target standards VS 1.1 and PS 2.0, there are eight t# (oT#) registers and two v# (oD#) registers.

2.2. Assembly Language for the GPU

Basically, there is no significant difference between the CPU assembly language and the GPU assembly language. Instructions require a few operands as follows.

```
add r1, r0, c0
```

In this example, two vectors in source registers r0 and c0 are added and stored into the destination register r1.

Table 1. Registers in VPs and FPs. The number of registers # is specified by standards. See [23] for details.

Unit	Type	Name	Read/Write
VP	Input register	v#	R
	Constant register	c#	R
	Temporary register	r#	R/W
	Address register	a0	R/W
	Output register	oPos, oFog, oPts oD#, oT#	W
FP	Input register	v#, t#	R
	Constant register	c#	R
	Temporary register	r#	R/W
	Sampler	s#	R
	Output register	oC#, oDepth	W

```
1: foreach vertex set do begin
2:     Execute the VP program on VPs; // specify a region
3:     Rasterize the region into fragments;
3:     foreach fragment do begin
4:         Execute the FP program on FPs;
5:     end
6: end
```

Figure 2. Pseudo code representing GPU execution. In this representation, our technique can be regarded as a loop-invariant code motion technique [2], because moving instructions from FPs to VPs is equivalent to moving them outside the inner loop.

Note here that instructions can be applied to the individual components of the vector data. To do this, we must specify the source/destination registers using masks: x, y, z, and w. For example, r0.xy specifies the first and the second components of register r0.

VP and FP programs have a limitation on program length. This limitation is specified by the number of instruction slots required for the program. For example, VS 1.1 and PS 2.0 have a maximum constraint of 128 slots and that of 96 slots for the VP program and the FP program, respectively. Most instructions occupy one instruction slot, but some arithmetic instructions require more slots.

Although the number of instruction slots does not represent the precise execution time, it provides useful information to compare two instructions in terms of execution time. Therefore, this number is useful to select the best combination of instructions, when all movable instructions cannot be moved due to the lack of computational resources.

```
1: mov    oPos,    v0    ; vertex coordinates
2: mov    oT0.xy,  v1    ; texture coordinates
```

(a)

```
    ; myPos = t0 (t0 is automatically set when invoked)
    ; leftPos = myPos + float2(-1.0f/W, 0.0f)
    ; rightPos = leftPos + float2(2.0f/W, 0.0f)
 1: mov    r0.z,    t0.y
 2: mov    r1.y,    c4.x              ; 0.0f
 3: mov    r0.y,    c4.x
 4: rcp    r0.w,    c3.x              ; -1.0f/W
 5: mov    r0.x,    -r0.w
 6: add    r1.x,    r0.w,    r0.w     ; 2.0f/W
 7: add    r0.xy,   r0,      t0
 8: add    r1.x,    r1.x,    r0.x
 9: add    r1.y,    r0.z,    r1.y
    ; set x_{n-1}, x_{n+1}, and x_n as texture data at
    ; leftPos, rightPos, and myPos, respectively
10: texld  r2,      r0,      s0
11: texld  r0,      r1,      s0
12: texld  r1,      t0,      s0
    ; convolution: y_n = sum_{k=-1}^{1} c_k x_{n+k}
13: mul    r2,      r2,      c0.x
14: mad    r1,      c1.x,    r1,      r2
15: mad    r0,      c2.x,    r0,      r1
16: mov    oC0,     r0                ; output y_n
```

(b)

Figure 3. Example of (a) VP and (b) FP programs performing 1-D convolution. W represents the texture width. See text for detail.

2.3. Programming Strategy for GPGPU

In modern GPUs, FPs provide higher performance than VPs, because rendering tasks usually need to process more fragments than vertices, as presented in Figure 2. Therefore, most GPGPU implementations use FPs as a computational engine in the GPU [26]. They regard FPs as a stream processor [18] and use VPs only for informing FPs of the working region. Thus, data is passed through VPs.

This strategy is called the stream programming model [18], which exploits the data parallelism inherent in the application by organizing data into streams and expressing computation as kernels that operate on streams. Streams here are usually stored as texture data, which can be fetched to FPs by samplers. A kernel is implemented as a FP program. The computed results can be transferred (readback) from the video memory to the main memory by using the graphics APIs mentioned in Section 2.1.

Table 2. Instructions available only on FPs.

Capability	Instruction
Texture reference	texkil, texld, texldb, texldp
Conditional selection	cmp, cnd
Dot product and add	dp2add

Figure 3 shows an example of VP and FP programs developed according to the stream programming model. These programs implement a 1-D convolution filter that smoothes a 2-D image. In this example, four vertices of a rectangle are given to VPs, and then VPs simply output two coordinates for each vertex: the coordinates on the screen (oPos) and those on the texture (oT0.xy). FPs receive an interpolated value (t0), namely the texture coordinates for a fragment. The FP program implements the convolution operation on each fragment, accessing its left and right neighbors. Because the texture coordinates are specified in the range [0,1], the relative coordinates of the left neighbor are given by (-1.0f/W, 0.0f), where W is the texture width.

In summary, many GPGPU implementations use the stream programming model due to its simplicity. However, VPs are not used well in this model, motivating us to move instructions from highly-loaded FPs to VPs.

3. Code Motion Technique

We now present our technique that aims at accelerating GPU programs by moving instructions from FPs to VPs. The problem here is that, given a pair of VP and FP programs, (1) which instructions are movable and (2) how these programs can be modified without changing the computational results.

3.1. Definition of Movable Instructions

To apply our technique to GPU programs, we have to make clear which instructions are allowed to move from FPs to VPs. This section presents the definition for such movable instructions. To define this, we basically focus on the three characteristics mentioned in the end of Section 2.1. In the following, we explain the definition using the example in Figure 3.

We define movable instructions such that they satisfy all of the following four conditions C1–C4:

C1. Executability: Movable instructions must be executable on VPs as well as on FPs. However, the instruction set available on VPs is not exactly the same as that available on FPs, as shown in Table 2. For example, the texld instruction cannot be moved to VPs, because VPs do not have a read capability to textures.

Therefore, in Figure 3(b), the texld instructions in lines 10–12 must be stay fixed in the FP program.

C2. Accessibility: As same as the limitations on instructions, some FP registers also have unique capabilities that VP registers do not have. Such unique registers are samplers s# and output registers oC# and oDepth. Samplers cannot be replaced by any register in VPs, because they are unique in terms of sampling a texture. Similarly, output registers are also unique in terms of having a write capability to the video memory. Thus, instructions accessing these unique registers cannot be moved to VPs. Note here that constant values are movable to VPs, because they are set by the VP/FP program, and thus are fixed before program execution. In Figure 3(b), the instructions in lines 10–12 and 16 must be in the FP program, because they access sampler s0 or output register oC0.

C3. Equivalency: Only linear operations can be moved to VPs, because FPs receive linearly interpolated data from VPs. Otherwise, namely if we move non-linear operations to VPs, we have wrong results due to the automated interpolation. Formally, this condition can be expressed as follows: computation C on FPs is movable if $C(L(\mathbf{V})) = L(C(\mathbf{V}))$, where L is the linear interpolation and $\mathbf{V}$ is the original output of VPs. In addition to this linearity of computation, we have to consider the precision of registers. That is, lower precision registers v# (oD#) cannot be used to pass high-precision data from VPs to FPs. Otherwise, we obtain wrong results due to round-off errors.

C4. Dependability: In addition to the GPU-oriented limitations mentioned above, we also have to consider data dependencies between instructions, because our code motion technique changes the execution order of instructions. Actually, Figure 2 indicates that moving an instruction to the VP program means that the instruction will be executed prior to all instructions in the FP program. Therefore, we must avoid the collapse of data dependencies in order to obtain correct results. Thus, an instruction is movable if all instructions that have a dependency on the instruction are also movable. In Figure 3(b), the instructions in lines 13–16 are not movable due to the dependencies from the unmovable texld instructions in lines 10–12.

According to the conditions mentioned above, our technique determines that the instructions in lines 1–9 are movable to VPs. These instructions intend to compute the coordinates in a texture in order to fetch the left and right neighbors. This address computation can be precomputed by VPs, because it satisfies condition C3 in addition to C1, C2, and C4. That is, the coordinates of the left and right neighbors (leftPos and rightPos) do not change whether they are computed for each fragment (after interpolation) or for each vertex (before interpolation).

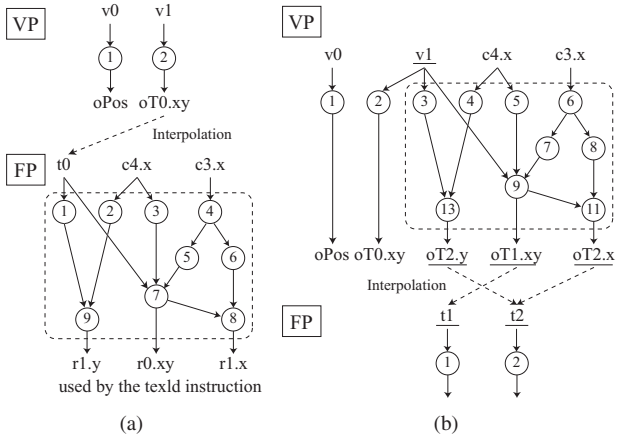

Figure 4. A portion of a directed acyclic graph representing data dependencies (a) before and (b) after code motion. A node ID represents the line number in the program. Underlined registers are modified to obtain correct results.

3.2. Code Motion Strategy

The next question is how instructions should be moved to the VP program in a proper manner. Our technique moves the code in the following three steps (see Figures 4 and 5).

1. Code movement: We add the code at the end of the VP program, as shown in Figure 5(a). At that point, since VPs have completed the original computation, the concatenated code is allowed to use all temporary registers in VPs.

2. Modification on inputs: The inputs to the code must be changed to obtain them directly in the VP program. Such inputs are values (1) in the FP input registers t# and (2) in constant registers c# (see Figure 4(a)). For the former case, we add some instructions in the VP program in order to save the origin of the VP output registers oT# as the new inputs to the code. However, these additional instructions are not required for most GPGPU implementations, which do not change the values of input registers v#. Thus, in most cases, we are allowed to replace operands t# with v# (lines 3 and 9 in Figure 5(a)). For the latter case, we replace operands c# with other unused c# if necessary.

```
      1: mov   oPos,   v0
      2: mov   oT0.xy, v1
→     3: mov   r0.z,   v1.y
→     4: mov   r1.y,   c4.x          ; 0.0f
→     5: mov   r0.y,   c4.x
→     6: rcp   r0.w,   c3.x          ; -1.0f/W
→     7: mov   r0.x,   -r0.w
→     8: add   r1.x,   r0.w,  r0.w   ; 2.0f/W
→     9: add   r0.xy,  r0,    v1
+    10: mov   oT1.xy, r0.xy
→    11: add   r1.x,   r1.x,  r0.x
+    12: mov   oT2.x,  r1.x
→    13: add   r1.y,   r0.z,  r1.y
+    14: mov   oT2.y,  r1.y
```

(a)

```
1: texld  r2,  t1,  s0
2: texld  r0,  t2,  s0
3: texld  r1,  t0,  s0
4: mul    r2,  r2,  c0.x
5: mad    r1,  c1.x, r1,  r2
6: mad    r0,  c2.x, r0,  r1
7: mov    oC0, r0
```

(b)

Figure 5. Example of (a) VP and (b) FP programs after code motion. See Figure 3 for the original program. Instructions marked with notations '→' and '+' are those moved from the FP program and those added during code motion, respectively. Underlined operands are changed to obtain correct results.

3. Modification on outputs: The data obtained by the new code must be sent properly to FPs. To do this, we add an instruction that sets the data to the unused output registers oT# in VPs (lines 10, 12, and 14 in Figure 5(a)). The data must also be received by FPs. Therefore, we replace the operands that refer the data with the unused input registers t# in FPs (lines 1 and 2 in Figure 5(b)). Note here that a proper pair of input/output registers must be selected to avoid passing data to wrong registers.

3.3. Code Selection Strategy

Although our code motion technique reduces the instruction slots used in FPs, it requires (1) more instruction slots in VPs and (2) more registers for passing data from VPs to FPs. For example, in Figure 5, our technique reduces the number of instruction slots in FPs from 16 to 7 slots, but increases that in VPs from 2 to 14 slots. Therefore, we must give priorities to movable instructions if these computational resources are not sufficient to move all of them. In such a case, we currently select a combination of instructions that minimizes the number of instructions slots required for the FP program.

Our technique reduces the number of instructions per fragment, and thereby reduces execution time spent for the FP program. In contrast to this timing benefit, there are some disadvantages in our technique. One apparent disadvantage is that it takes more time on VPs due to increased instructions in the VP program. The other disadvantage is that, more data is passed from VPs to FPs, so that this increase may decrease the entire performance if the rasterizer would become a performance bottleneck.

Despite the disadvantages mentioned above, we think that our technique is effective in improving the performance of typical GPGPU programs. As we mentioned in Section 2.3, such programs usually use FPs as a computational engine and access textures to fetch data. In this programming model, there are many instructions for computing the texture coordinates, which are usually movable instructions, as presented later in Section 4.

Table 3. Experimental environments.

Component	Specification
CPU	Pentium 4 3.0-GHz
Main memory	1 GB
Graphics bus	AGP 8X
GPU	ATI Radeon X800 Pro Core clock 475 MHz Video memory 256 MB 6 VPs and 12 FPs
Operating system	Windows 2000
Graphics API	DirectX 9.0 [23]

4. Experimental Results

We now present some experimental results showing the effectiveness of our technique. We investigate the technique from the following two viewpoints: the reduction of execution time and the improvability for a variety of GPU programs. Table 3 shows the specification of the experimental machine, which consists of commodity hardware.

4.1. Timing Results

We apply our technique to a Gaussian filter that smoothes an $N \times N$ pixel image. Pixels here are stored in a 32-bit RGBA format. The kernel size of the filter is set

Table 4. Timing results for a Gaussian filter with kernel size of $K \times K$ pixel, where $K = 13$. V and F represents the number of instruction slots required for VP and FP programs, respectively. Ratio represents the reduction percentage of FP slot or time.

Instructions			Image	Filtering		Readback	Total		Performance
V (slot)	F (slot)	Ratio $R_F(\%)$	size N	Time t (ms)	Ratio R_t (%)	Time (ms)	Time T (ms)	Ratio R_T (%)	$12KN^2/t$ (GFLOPS)
2	63	—	256	1.3	—	2.0	3.3	—	7.8
			512	4.9	—	5.3	10.2	—	8.3
			768	10.9	—	11.9	22.8	—	8.4
			1024	19.5	—	21.0	40.5	—	8.4
6	60	5	256	1.2	6	2.0	3.2	2	8.3
			512	4.7	6	5.3	10.0	3	8.8
			768	10.3	6	11.9	22.2	3	8.9
			1024	18.2	7	21.0	39.2	3	9.0
9	57	10	256	1.2	11	2.0	3.2	4	8.8
			512	4.4	11	5.3	9.7	5	9.4
			768	9.7	11	11.9	21.6	6	9.5
			1024	17.2	11	21.0	38.2	5	9.5
12	54	14	256	1.1	17	2.0	3.1	6	9.4
			512	4.1	17	5.3	9.4	8	10.0
			768	9.1	17	11.9	21.0	8	10.1
			1024	16.1	18	21.0	37.1	8	10.2
15	51	19	256	1.0	22	2.0	3.0	9	10.0
			512	3.8	23	5.3	9.1	11	10.7
			768	8.4	23	11.9	20.3	11	10.9
			1024	14.9	23	21.0	35.9	11	11.0
18	48	24	256	0.9	28	2.0	2.9	11	10.8
			512	3.5	29	5.3	8.8	14	11.6
			768	7.8	29	11.9	19.7	14	11.8
			1024	13.8	29	21.0	34.8	14	11.8
21	45	29	256	0.9	33	2.0	2.9	13	11.7
			512	3.2	34	5.3	8.5	16	12.6
			768	7.2	34	11.9	19.1	17	12.8
			1024	12.7	35	21.0	33.7	17	12.9
23	43	32	256	0.8	39	2.0	2.8	15	12.8
			512	3.0	40	5.3	8.3	19	13.8
			768	6.6	40	11.9	18.5	19	14.0
			1024	11.6	40	21.0	32.6	19	14.1

to be $K \times K$ pixel. This 2-D filter is realized using 1-D filtering in the horizontal and vertical directions. The initial implementation is typically developed according to the stream programming model. It requires 2 slots for the VP program and 63 slots for the FP program.

There are 20 movable instructions composing 7 independent groups in terms of data dependencies. Accordingly, if all groups are moved, the VP program requires approximately 20 instruction slots and 7 oT# (t#) registers in addition to the initial usage. Because we have enough registers and instruction slots to move all groups, we then move them in a stepwise manner to investigate the relationship between the reduction of instruction slots and that of execution time. Table 4 shows the results. In the following discussion, let V and F be the number of instruction slots consumed by the VP program and the FP program, respectively.

As we move more groups to the VP program, the FP program uses less instruction slots F, finally achieving a minimum of 43 slots. On the other hand, the VP program requires more slots V, resulting in a maximum of 23 slots. Thus, we generate seven variations in addition to the initial implementation.

We then measure the execution time of these eight implementations with varying the size of image N from 256 to 1024 pixels. In Table 4, Notation T represents the total time of implementation, containing (1) the filtering time t spent on the GPU and (2) the readback time required for data transfer from video memory to main memory. Note

here that time t includes the initialization time for sending data from main memory to video memory. This initial procedure is automatically processed by underlying APIs, and thus we cannot separate the initialization time from the processing time. Ratio R_F, R_t, and R_T represent the reduction percentage as compared to the original implementation.

For all implementations, we obtain ratio $R_t > 0$, so that our technique successfully reduces the filtering time t. When we move all movable instructions ($F = 43$), we obtain the best result with achieving 39–40% reductions against the original implementation. These reductions can be explained by comparing ratio R_t with R_F. These ratios show a linear relationship between the reduction of execution time and that of instruction slots. That is, the number of instruction slots F affects the execution time t on the GPU. Therefore, F can be used as a selection measure for minimizing time t if all instructions cannot be moved.

The disadvantages mentioned in Section 3.3 are not revealed in the timing results, because the FP program is executed much more time as compared to the VP program. In our implementation, given an $N \times N$ pixel image, the VP program is invoked only four times to process the vertices of the image while the FP program is invoked N^2 times to process all fragments. Actually, on the experimental machine with 6 VPs and 12 FPs, the FP program is executed at least 2^{13} times as much as the VP program. Therefore, in typical GPGPU implementations, we will achieve higher performance as we move more instructions, even though the VP program takes more time.

Although our technique achieves 39–40% reductions, these percentages are not the same as those observed in the total time T (15–19%). These less reductions are due to the readback time, which takes relatively long time as compared with the filtering time. Thus, the readback of textures is not so fast on our machine whose GPU is connected by an AGP bus. Because the AGP bus provides a low bandwidth of 266 MB/s from video memory to main memory (2.1 GB/s for the opposite direction), this problem will be resolved by using faster interconnects such as PCI Express buses, which provide a bandwidth of 4 GB/s for both directions.

4.2. Improvability

We next apply our technique to a variety of GPU programs: (1) 4 kernels each implementing 1-D and 2-D convolution, integer sorting, and summation; (2) 13 kernels composing a Navier-Stokes fluid simulator [27]. Because it originally does not have a VP program, we have added a VP program that simply pass data to FPs, as presented in Figure 3(a); and (3) 9 shader programs [9] as examples of graphics (non-GPGPU) applications. These shaders apply a visual effect to rendering objects.

As shown in Tables 5 and 6, our technique successfully reduces the number of instruction slots F in 9 out of 17 kernels. Among these kernels, Scroll and Scroll2 kernels achieve the best reduction of 55% in terms of slots F. These kernels are similar to the example in Figure 3, because they mainly consists of address computation for two texture references. Thus, address computation is one of the typical computations that can offload FPs to VPs. Note here that the 55% reduction results in about 15% less time in these cases. This small effect is due to the initialization time mentioned in Section 4.1, because this overhead is relatively large in these small kernels.

On the other hand, Advect kernel results in no improvement, though it consumes relatively many instruction slots F, as compared to the others. The reason for this is that, it performs many texture references, which are not executable on VPs, and moreover, all instructions have data dependencies from these texture data. Also, Sorting kernel in Table 5 fails to reduce instruction slots F. However, this kernel is improved by exchanging the execution order of instructions, which is automatically generated by a compiler. This can be explained as follows. Suppose that we have two instructions, r = a then r+= b, and operands a and b are un-

Table 5. Number of instructions before and after applying our technique. Ratio represents the reduction percentage of FP slots.

Kernel	Before		After		Ratio
	V	F	V	F	R_F (%)
1-D convolution	2	63	23	44	30
2-D convolution	2	55	40	47	15
Sorting	2	40	14	40	0
Summation	2	16	9	9	44

Table 6. Number of instructions in a fluid simulator [27].

Kernel	Before		After		Ratio (%)	
	V	F	V	F	R_F	R_t
ActuallyRender	2	1	2	1	0	—
Clear	2	2	2	2	0	—
Copy	2	2	2	2	0	—
Add	2	6	2	6	0	—
Splat	2	9	2	9	0	—
Scroll	2	11	8	5	55	15
Scroll2	2	11	8	5	55	11
Vortex	2	16	2	16	0	—
Divergence	2	17	10	9	47	37
Subgradient	2	19	10	11	42	39
Jacobi	2	21	10	13	38	37
Advect	2	34	2	34	0	—
Display	2	41	10	33	20	20

Table 7. Number of instructions in shader programs [9].

Shader program	Before		After		Ratio
	V	F	V	F	R_F (%)
Multitexture shader	8	3	8	3	0
Dot-3 bump mapping	19	3	19	3	0
Toon shader	28	9	28	9	0
Ambient shading	6	—	6	—	—
Diffuse shader	9	—	9	—	—
Lambertial diffuse shader	12	—	12	—	—
Shilhouette shader	20	—	20	—	—
Fresnel shader	20	—	20	—	—
Phon-blinn	25	—	25	—	—

movable and movable, respectively. In this case, we cannot move both instructions, but if we exchange them, r=b then r+=a, we can move the first instruction to VPs.

In contrast to the GPGPU programs mentioned above, all shader programs fail to reduce instruction slots F (Table 7). These shaders do not utilize the programmability of FPs, and thus do not have FP programs.

In summary, address computation for texture references is typically movable instructions. Reordering instructions could yield further improvement. On the other hand, there will be no improvement if the FP program has (1) few instruction and (2) many data dependencies from texture data, especially if such dependencies exist in the beginning of the FP program.

5. Related Work

To the best of our knowledge, there are a few papers [7,12,16,17] that try to improve the performance of GPGPU applications. Hall et al. [12] present cache-aware algorithms for matrix multiplication on the GPU. Their algorithms are evaluated by Fatahalian et al. [7] using real GPUs. These cache-aware algorithms are specific to matrix multiplication. On the other hand, our technique does not depend on particular applications. Also our technique is a post-compile optimization approach, so that it does not need any modification to underlying algorithms.

Jiang et al. [17] also try to improve the performance of matrix multiplication. They automate the tuning process by selecting the best implementation from multiple versions according to empirical evaluation. Their tuning strategy is similar to that in ATLAS [31] in terms of employing parameterized code generators that can generate multiple codes according to input tuning parameter values. A similar study [16] is also presented for LU decomposition. Because these projects optimize only the FP program, our optimization approach is unique in that it offloads FPs to VPs.

Because our technique optimizes assembly programs, it can be integrated with high-level compilers. For example, it can work as a post-compile optimization technique by using assembly code generated by high-level compilers such as Brook [6] and Cg [22]. Note here that Brook is designed for GPGPU applications but it employs only FPs as a computational engine. Cg generates efficient code but the optimization is independently applied to each of the VP and FP programs.

With respect to graphics applications, many techniques have been proposed to improve the rendering performance [8]. These techniques first find a performance bottleneck in the pipeline execution, and then try to reduce the amount of data streams that go through the bottleneck. For example, they simplify or eliminate polygonal objects to reduce the number of vertices if VPs limit the entire performance. These reduction-based techniques are effective to graphics applications, which are allowed to reduce the level of details in the output scene. However, they cannot be applied to GPGPU applications, which must not omit computations that change the computational results.

As we presented in Figure 2, moving instructions from FPs to VPs corresponds to moving them outside the inner fragment loop. Therefore, in this interpretation, our code motion technique can be regarded as a loop-invariant code motion technique [2, 4], which aims at reducing the computational amount of the entire loop. This computational reduction is also effective for future GPUs that will integrate VPs and FPs into unified sharders. Such GPUs will be capable of dynamically allocating unified sharders to vertex or fragment operations, so computational reduction plays an important role in achieving the best resource allocation.

6. Conclusions

We have presented a code motion technique that aims at accelerating GPU applications by moving assembly instructions from FPs to VPs. Our technique differs from prior techniques in terms of the following points: our approach offloads FPs to VPs, so that differs from other approaches that independently optimizes each of the VP and FP programs; our post-compile optimization can be integrated with high-level compilers; and our technique keeps the same I/O specification between the CPU and the GPU, allowing us to use CPU programs without any modification.

The experimental results show that (1) our technique reduces execution time of a Gaussian filter by approximately 40%; (2) it successfully reduces the FP workload in 10 out of 18 GPGPU programs; and (3) there is a linear relationship between the reduction of FP instruction slots and that of the filtering time. Thus, The effectiveness of the technique depends on the number of instructions moved to VPs. In current GPUs, this is usually limited by the number of

registers that pass data from VPs to FPs.

One future work is to verify the computational results in terms of error. As we mentioned before, FPs and VPs have different designs. Therefore, by moving instructions from FPs to VPs, the results might be changed due to this difference. Although we have confirmed that the experimental programs return the same results, we think more detailed verifications are needed, because the GPU seems not be rigorous with errors [14]. We are also planning on developing a tool to automate the code motion procedure.

References

[1] General-Purpose Computation Using Graphics Hardware, 2005. http://www.gpgpu.org/.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, editors. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] T. Akenine-Möller and E. Haines, editors. *Real-Time Rendering*. Morgan Kaufmann, San Mateo, CA, second edition, July 2002.

[4] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graphics*, 22(3):917–924, July 2003.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graphics*, 23(3):777–786, Aug. 2004.

[7] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware (GH'04)*, pages 133–137, Aug. 2004.

[8] R. Fernando, M. Harris, M. Wloka, and C. Zeller. Programming graphics hardware. In *EUROGRAPHICS 2004 Tutorial Note*, Aug. 2004.

[9] R. Fosner. *Real-Time Shader Programming*. Morgan Kaufmann, San Mateo, CA, 2003.

[10] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC2005)*, Nov. 2005.

[11] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, Reading, MA, second edition, Jan. 2003.

[12] 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, Mar. 2003.

[13] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware (GH'02)*, pages 109–118, Sept. 2002.

[14] K. E. Hillesland and A. Lastra. GPU floating point paranoia. In *Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04)*, pages C–8, Aug. 2004.

[15] K. E. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using computer graphics hardware. In *Proc. SIGGRAPH'99*, pages 277–286, Aug. 1999.

[16] F. Ino, M. Matsui, and K. Hagihara. Performance study of LU decomposition on the programmable GPU. In *Proc. 12th Int'l Conf. High Performance Computing (HiPC'05)*, pages 83–94, Dec. 2005.

[17] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'05)*, pages 185–196, Sept. 2005.

[18] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, Mar. 2001.

[19] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proc. Int'l Conf. High Performance Computing and Communications (SC2001)*, Nov. 2001.

[20] J. Lengyel, M. Reichert, B. R. Donald, and D. P.Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *Proc. SIGGRAPH'90*, pages 327–335, Aug. 1990.

[21] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7/8):444–456, Dec. 2003.

[22] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graphics*, 22(3):896–897, July 2003.

[23] Microsoft Corporation. DirectX, Asm Shader Reference, 2005. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/reference.asp.

[24] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, Mar. 2005.

[25] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965.

[26] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *EUROGRAPHICS 2005, State of the Art Report*, pages 21–51, Aug. 2005.

[27] V. Palmer. Navier-Stokes fluid simulator, Dec. 2004. http://www.strangebunny.com/techdemo_stokes.php.

[28] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, Reading, MA, fourth edition, Dec. 2003.

[29] D. Stevenson. A proposed standard for binary floating-point arithmetic. *IEEE Computer*, 14(3):51–62, Mar. 1981.

[30] H. Takizawa and H. Kobayashi. Multi-grain parallel processing of data-clustering on programmable graphics hardware. In *Proc. 2nd Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'04)*, pages 16–27, Dec. 2004.

[31] 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, Jan. 2001.