Small-Scale Reconfigurability for Improved Performance and Double-Precision in Graphics Hardware

Kevin Dale[†], Jeremy W. Sheaffer[†], Vinu Vijay Kumar[†], David P. Luebke[‡],
Greg Humphreys[†], and Kevin Skadron[†]
(submitted for review on November 30, 2006)

We explore the application of *Small-Scale Reconfigurability* (SSR) to graphics hardware. SSR is an architectural technique wherein functionality common to multiple subunits is reused rather than replicated, yielding high-performance reconfigurable hardware with reduced area requirements (Vijay Kumar and Lach 2003). We show that SSR can be used effectively in programmable graphics architectures to allow double-precision computation without affecting the performance of single-precision calculations and to increase fragment shader performance with a minimal impact on chip area.

1 Introduction

Every hardware system makes a tradeoff between performance and flexibility. At one end of the spectrum, general purpose processors provide maximum flexibility at the expense of performance, area, power consumption, and price. Custom ASICs are the other extreme, providing maximum performance at a minimum cost, albeit for only a very narrow set of applications.

Modern graphics hardware requires both high performance and flexibility, placing it somewhere between these two extremes. Traditional intermediate hardware solutions like FPGAs are inappropriate for graphics processors because of their large size and low performance relative to their fixed-logic counterparts. Small-scale reconfigurability (SSR) provides an attractive compromise; systems that use SSR components can approach the high speed and small size of ASICs while providing some specialized configurability (Vijay Kumar and Lach 2003). In this paper, we explore the applicability of SSR to programmable graphics hardware.

The simplest example of a reconfigurable component is two fully functional components connected with a multiplexer (see Fig. 1). Although these two

---

[†]University of Virginia, {kdale, jws9c, vv6v, humper, skadron}@virginia.edu
[‡]NVIDIA Research, david@luebke.us

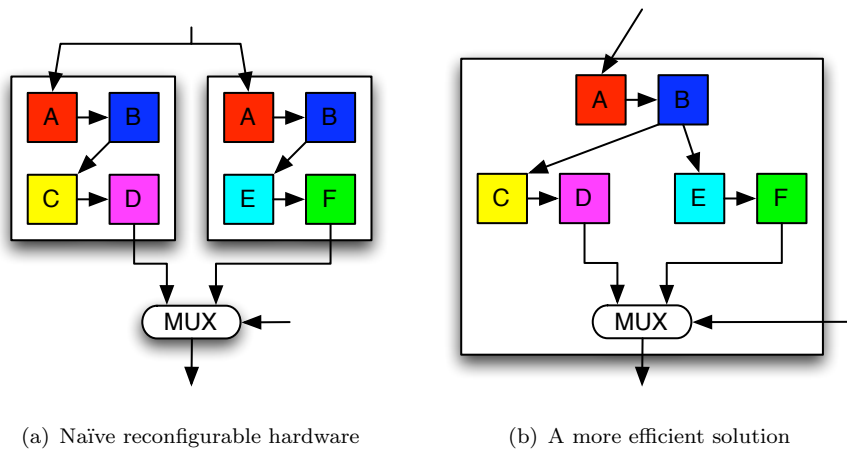(a) Naïve reconfigurable hardware   (b) A more efficient solution

Figure 1. A naïve implementation of reconfigurable hardware can be built by simply multiplexing between two distinct, unmodified units (a), but a more efficient design would reuse common substructure to avoid replication (b).

components are disjoint, in typical usage they will contain substantially similar redundant substructures, which is precisely the situation in which SSR performs best. Rather than replicate all of the redundant structure, one can instead reuse common substructure, and do so at a fine granularity within a single component (Vijay Kumar and Lach 2003, Chiricescu et al. 2002).

A common SSR unit is the morphable multiplier. These multiplier-adders can be reconfigured into a multiplier or an adder in a single cycle. When used to create fixed-point units, morphable multipliers yield a nearly 17% reduction in total area when compared to the sum of the sizes of their constituent parts (Chiricescu et al. 2002).

Graphics processors, like specialized multimedia processors and DSPs, are a particularly suitable target for SSR due to their vector-processor like operations. When the same operation is performed repeatedly in SIMD fashion, reconfiguration and its associated overhead is infrequently needed, and any cost can be amortized over many instructions. Furthermore, SSR-based components typically have lower static power requirements because less hardware goes unused.

2   Related Work

Dynamically reconfigurable hardware has been a popular topic in recent computer architecture literature, especially in the FPGA and reconfigurable computing communities. The configurability of these systems serves myriad design

goals, among them improved performance, power, area, and fault tolerance characteristics.

Even et al. (1997) describe a dual mode IEEE multiplier—a pipelined unit capable of producing one double-precision or two single-precision multiplications every clock cycle with a three cycle latency. The authors argue that the reuse of substructure yields a cheap device that performs well for both precisions. They further claim that the single precision mode is particularly useful for SIMD applications, like graphics, because it is conducive to systems on which the same operation is regularly repeated on large numbers of data points.

Guerra et al. (1998) explore *built-in-self-repair* (BISR) and its application to fault tolerance, manufacturability, and application-specific programmable processor design. Previous work in the area of dynamic repair had made use of specialized redundant units to replace damaged units; their paper describes the synthesis of more general units that can replace any of several units on a chip when damage is detected. The authors coin the term HBISR (*heterogeneous BISR*) for the technique.

A *morphable multiplier* is a device capable of performing either a fixed point multiply or add using the same hardware structure (Chiricescu et al. 2002). Morphable multipliers require less area than the sum of the area needed for a separate multiplier and adder (in fact, they require only slightly more than a multiplier alone), while imposing negligible performance penalties.

Metrics like area, performance, and power are easily quantified, but it is less obvious how to measure the increasingly important metric of hardware flexibility. Compton and Hauck have defined a testing method and quantification metric for flexibility of reconfigurable hardware (Compton and Hauck 2004). Other examples of relevant research in reconfigurable hardware include Kim et al. (1997), Chiou et al. (2005).

The work in this paper makes use of Brook (Buck et al. 2004), a stream-based programming language which allows the programmer to write general-purpose applications for a GPU without worrying about the sometimes byzantine details of GPU programming. Our experiments all use Chromium (Humphreys et al. 2002) to intercept and analyze streams of graphics commands made by real applications. The primary advantage of using Chromium is that we ensure that our workloads are not contrived. Although we use Brook and Chromium without modification, we have enhanced the Qsilver graphics architectural simulator (Sheaffer et al. 2004, 2005) to model the necessary aspects of the fragment pipeline. A detailed description of our modifications to Qsilver and our experimental setup are presented in Sect. 3 and Sect. 4.

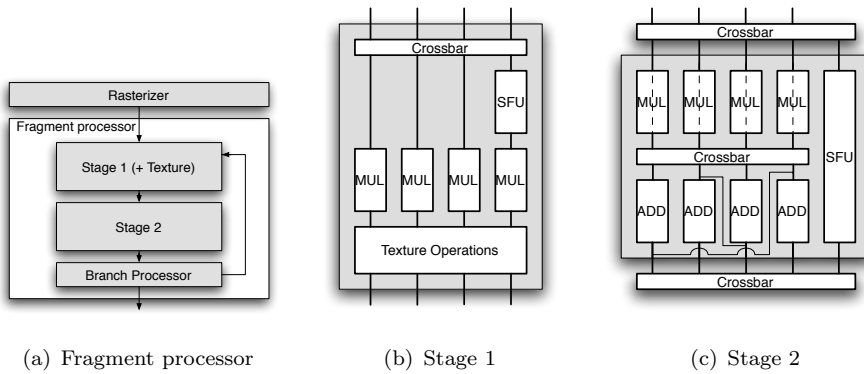(a) Fragment processor   (b) Stage 1   (c) Stage 2

Figure 2. Baseline fragment units used for comparison. Stage 2 can take up to three 4-channel operands, one of which directly feeds the ADD units and whose data path is represented here by dashed lines. Note the additional data paths that cascade the ADD units; these allow for a single-pass dot product (Seifert 2004).

3   Simulation Setup

3.1   *The Qsilver Simulator*

Qsilver is a simulation framework for graphics architectures that can simulate low-level GPU activity for any existing OpenGL application (Sheaffer et al. 2004). Qsilver uses Chromium (Humphreys et al. 2002) to intercept and transform an OpenGL application's API calls and create an annotated trace that encapsulates geometry, timing, and state information. This trace serves as input to the Qsilver simulator core, which performs an accurate timing simulation of the graphics hardware and produces detailed statistics.

Qsilver is configured at runtime with a description of its pipeline. In these experiments we simulate an NV4x-like architecture, with a pipeline configuration similar to that of NVIDIA's 6800 GT, so we configure Qsilver to model a system with 6 vertex pipelines and 16 fragment pipelines. The fragments are tiled in blocks of 2×2, so we effectively have 4 tile pipelines, each of which can process 4 fragments simultaneously. NV4x GPUs use a similar tiled configuration in the fragment engine (Kilgariff and Fernando 2005).

To account for modifications to the fragment pipeline, we enhanced Qsilver to track fragment shader activity. Our modified Qsilver simulator stores a per-triangle identifier which uniquely specifies which, if any, fragment shader was bound when that triangle was being rendered. We also store the text of the fragment shaders so that they can be analyzed by the Qsilver simulator core.

3.2 *Baseline Architecture*

Both of the following experiments hold fixed the graphics pipeline described above and focus on the programmable path of the fragment engine. While the NV4x vertex engine follows a MIMD architecture, its fragment engine is truly SIMD in nature. Additionally, in many modern games the majority of fragments are shaded by fragment programs (see Fig. 3), so we focus our efforts on the programmable path in the fragment engine.

Our baseline fragment pipeline, depicted in Fig. 2, is similar to that found in NV4x GPUs[1]. A single fragment unit contains two stages; four-channel fragments (RGBA) reach stage 1 from either the rasterizer or fragment pipeline loopback. Stage 2 can execute instructions in parallel with stage 1 in *dual-issue* mode as well sequentially, taking its operands from the output of stage 1. Crossbars route operands to the appropriate functional units, and *Special Function Units* (SFUs) are used to perform special scalar operations like reciprocal square root. The fragment units can also operate in *co-issue* mode, whereby a single 4-channel data path functions as two distinct data paths, with independent instructions executing in parallel, on the same unit, across these two data paths—e.g., a 3-vector and a scalar, or two 2-vectors (Kilgariff and Fernando 2005).

`NVShaderPerf`[2]—a utility that displays shader scheduling information for NVIDIA hardware—is used to schedule programs for our baseline architecture. While NV4x GPUs have dedicated hardware for performing common half-precision operations in parallel with full-precision operations, none of the fragment programs tested included any half-precision operations. However, to be sure of a legitimate comparison of performance along the full-precision path, `NVShaderPerf` is configured to schedule programs for our NV4x-like architecture using the full-precision path only.

3.3 *Benchmarks*

For benchmarking, we use the recent game Doom III (see Fig. 3), as well as four demo programs included with the Brook distribution. We use Chromium to intercept the fragment programs used in each benchmark and to generate traces for simulation under Qsilver. Each benchmark, along with its fragment programs, is summarized below.

(i) **doom3**, a representative 50-frame demo from the game. Includes a shader for general per–pixel lighting and a special effects shader.

---

[1]Based on those details that have been made available to the public or indirectly obtained via patents and extensive benchmark tests (see Kilgariff and Fernando 2005, Seifert 2004, for additional details).
[2]Unified compiler version 77.80.

<div align="center">(a)   (b)</div>

Figure 3.   Screen captures from the `doom3` benchmark. On the left, the color of each pixel is modulated to indicate which fragment program generated it. The right image is the unmodified rendering from the game. Notice that the majority of pixels are generated by programmable fragment shaders.

(ii) **bitonic_sort**, a parallel sorting network. Includes a main sorting kernel and simple pass-thru kernel.

(iii) **image_proc(25,25)**, an image convolution shader. Includes a main convolution kernel and pass–thru kernel.

(iv) **particle_cloth(5,10,15)**, a cloth simulation. Includes six kernels that implement the simulation.

(v) **volume_division(100)**, a volume isosurface extractor. Includes ten kernels for various stages of the extraction.

4   Experiments and Results

In this section, we describe two experiments we performed to validate our hypothesis that using SSR components in a modern GPU architecture can benefit certain applications. We show improved performance across a set of test applications with only a minimal impact on GPU die area and also demonstrate that double-precision floating point capabilities can be added to the fragment pipeline without affecting the performance of single-precision applications.

4.1   *Increased Throughput*

We first compared the simulated performance of the NV4x-like fragment pipeline to that of an SSR fragment pipeline architecture, whose fragment units are depicted in Fig. 4.
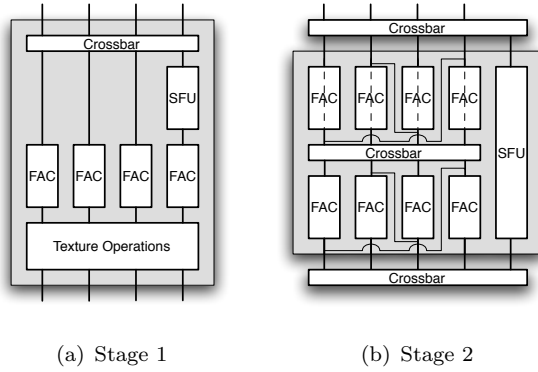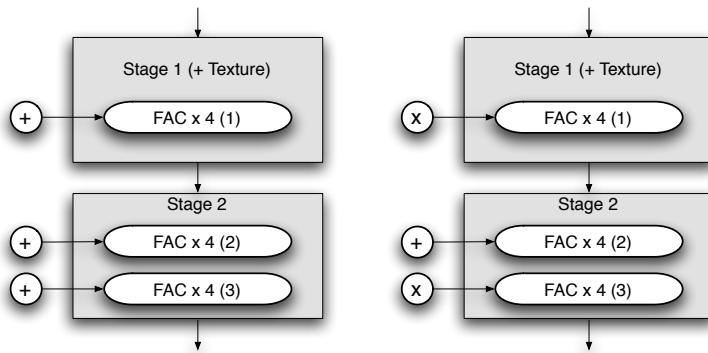
(a) Stage 1   (b) Stage 2

Figure 4.   Proposed SSR fragment units for the first experiment. FAC modules are Flexible Arithmetic Units, and they replace each of the ADD and MUL units in our baseline architecture.

4.1.1   *Target SSR architecture.* The fragment units in our target SSR architecture are similar to those in the baseline architecture; however, we replace both the multipliers and adders in stages 1 and 2 with single-precision *Flexible Arithmetic Units* (FACs). An FAC can be very quickly reconfigured to perform either a multiplication or an addition and uses only slightly more gates than a multiplier. With current technology, these FACs can produce a result every cycle and can be reconfigured between cycles, assuming a 400 MHz clock and a two-stage pipeline (Vijay Kumar and Lach 2003). Finally, in the first set of FACs in our SSR architecture, we duplicate the accumulate data paths from the baseline architecture's ADD units. These data paths require a trivial amount of additional area overhead.

In addition to supporting all the existing functionality of our baseline units, the modified SSR units provide new scheduling opportunities beyond those of the baseline. First, the baseline fragment pipe is only capable of performing a single full-precision 4-vector addition per pass in stage 2 (Seifert 2004), while the SSR pipeline is capable of performing three in one pass—one in stage 1 and two chained additions in stage 2 (see Fig. 5a). Moreover, there is more freedom to schedule dot product and multiply-accumulate operations, both of which are extremely common in fragment programs. For example, the SSR pipeline can execute a 32-bit 3-channel dot product (DP3) and dependent scalar-vector multiplication—e.g., the expression $(\vec{a} \cdot \vec{b})\vec{c}$—in a single pass by computing the per-channel multiply of $\vec{a}$ and $\vec{b}$ in stage 1, accumulating the channel products to obtain $\vec{a} \cdot \vec{b}$ in the first set of FACs in stage 2, and performing a scalar-vector multiply in its second set of FACs (Fig. 5b). Extending this scheduling approach to co-issue configurations is straightforward.

Stage 1 (+ Texture)		Stage 1 (+ Texture)
+ → FAC x 4 (1)		x → FAC x 4 (1)

Stage 2		Stage 2
+ → FAC x 4 (2)		+ → FAC x 4 (2)
+ → FAC x 4 (3)		x → FAC x 4 (3)

(a) Single-pass configuration for multiple 4-vector additions.

(b) Single-pass configuration to compute $(\vec{a} \cdot \vec{b})\vec{c}$.

Figure 5.   Two example configurations that provide additional scheduling opportunities for the SSR fragment pipeline.

4.1.2   *Shader analysis.* Considering the additional scheduling opportunities provided by the SSR architecture, as well as the known scheduling constraints of NV4x GPUs, we hand-scheduled each fragment program for our SSR fragment engine. As was done for the fragment program schedules on the baseline architecture, we limited program schedules for the SSR architecture to the full-precision path as well.

With 16 fragment pipelines and a 400Mz clock, both architectures have a maximum throughput of 6 GP/s (gigapixels per second). This assumes a 1-cycle texture lookup. Results for the benchmarks and their constituent shaders are given in Table 1. For most of the shaders, the SSR architecture provides an improvement over the baseline. The amount of improvement of course varies from shader to shader, depending upon the mix of instructions and corresponding scheduling advantages for the SSR architecture. Frequent dot product and multiply-accumulate instructions across the benchmarks led to the performance improvements for the SSR architecture seen in Table 1.

4.1.3   *Full pipeline simulation.* There are a number of possible bottlenecks in the full graphics pipeline, however, that can prevent performance improvements in the fragment engine core from being realized across the full pipeline. First, available memory bandwidth and cache performance in the texture units can prevent the pipeline from achieving maximum pixel throughput. GPUs typically incorporate a high degree of multithreading to hide the latency introduced by texture memory accesses. For example, when a shader unit arrives at a texture memory instruction, the memory access is initialized and the frag-

Table 1. Per-pixel execution time and pixel throughput, in megapixels per-second (MP/s), for the baseline and target fragment units, assuming a 1-cycle texture lookup.

Benchmark	Shader	Execution time (cycles)		Throughput (MP/s)	
		Baseline	SSR	Baseline	SSR
bitonic_sort	0	15	14	426.67	457.14
	1	1	1	6400.00	6400.00
doom3	0	12	11	533.33	581.82
	1	7	7	914.29	914.29
image_proc	0	28	26	228.57	246.15
	1	1	1	6400.00	6400.00
particle_cloth	0	6	6	1066.67	1066.67
	1	16	15	400.00	426.67
	2	18	17	355.56	376.47
	3	10	8	640.00	800.00
	4	11	9	581.82	711.11
	5	2	2	3200.00	3200.00
volume_division	0	2	2	3200.00	3200.00
	1	1	1	6400.00	6400.00
	2	22	20	290.91	320.00
	3	30	26	213.33	246.15
	4	19	16	336.84	400.00
	5	57	52	112.28	123.08
	6	15	13	426.67	492.31
	7	13	13	492.31	492.31
	8	56	52	114.29	123.08
	9	1	1	6400.00	6400.00

ment is temporarily banked until the memory operation is complete. A context switch occurs immediately, at which point the shader unit can continue doing useful work on another fragment's thread. In a SIMD environment, the thread pool is sufficiently large to effectively hide memory latency in this manner, provided that there is also sufficient memory bandwidth and a large ratio of arithmetic instructions to memory instructions. While texture access patterns vary across the benchmarks included here, all of their shaders contain a large percentage of arithmetic instructions, enabling effective latency-hiding. Qsilver employs a simple probabilistic cache model that can reasonably capture this phenomenon (Sheaffer et al. 2004) in the full pipeline simulation.

The vertex processor can also be a system bottleneck, particularly for scenes with a large number of extremely small (in screen-space) polygons. However, for many GPGPU applications, and for all of those included in the suite of benchmarks here, the vertex processor is relatively inactive. A typical GPGPU application only renders screen-filling quadrilaterals; this makes for an extremely low polygon-to-fragment ratio and a nearly-idle vertex processor.

Qsilver's queue-based simulation architecture models the graphics pipeline at a resolution sufficient to capture the relative amounts of activity between the fragment and vertex engines. This is accomplished by aggregating architectural performance counter data across the course of the simulation. Counters for pre-transform-and-lighting vertex queues, as well as those for fragment creation,
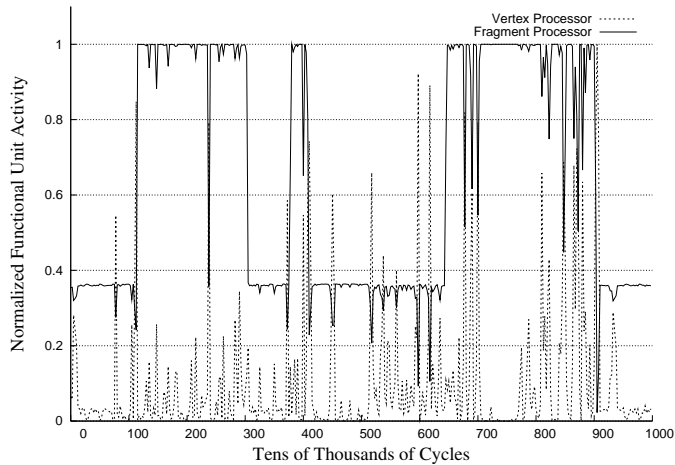
Figure 6.   Traces of pre-transform-and-lighting vertex queue writes and fragments created during a single frame of `doom3`, generated from Qsilver counter data. Fragment activity is at its maximum for a substantial portion of the frame, while vertex activity reaches its peak only once. This frame is representative of all frames across the `doom3` benchmark, indicating that the benchmark is predominantly fragment-bound.

are among Qsilver's counters. Full queues block earlier stages in the pipeline and so imply vertex- or fragment-bound behavior, respectively (Sheaffer et al. 2004). Fig. 6 was generated from these Qsilver counters and indicates that, like the Brook benchmarks, the Doom III benchmark is also predominantly fragment-bound.

For `doom3`, there is also the possibility that the fixed-function path in the fragment engine is used almost exclusively, given that the benchmark only uses two shaders across the entire 50-frame trace. However, the game performs per-pixel lighting with a fragment shader for most objects in the game, using OpenGL fixed-function lighting rarely. The 1-cycle improvement in this heavily-used shader should likely show a significant performance increase over the entire pipeline as well.

Fig. 7 shows performance results from full-pipeline simulations under Qsilver, using the fragment program schedules discussed in the previous section. For all benchmarks, speedup over the entire graphics pipeline for the SSR architecture does indeed correspond with the fragment processor performance results in Table 1. This indicates that all benchmarks are sufficiently fragment processor-bound for the performance increase in the fragment units to translate to a corresponding improvement over the full pipeline.

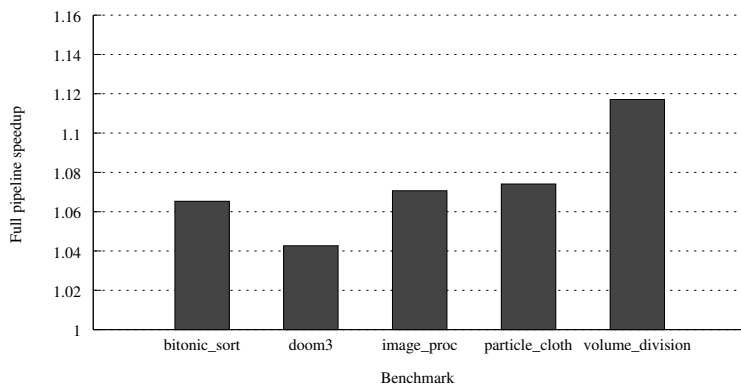Equally as important, based on conservative inverter-equivalent gate count

Figure 7. Speedup for the target SSR architecture over the baseline for a full pipeline simulation under Qsilver.

estimates[1], each FAC requires 12,338 gates, only 710 more than a single-precision multiplier (11,628 gates). Replacing the adders (7,782 gates) requires 4,556 additional gates. This additionally requires the small overhead of a multiplexer to configure the FACs. Given these gate estimates, with 16 fragment pipelines, the cost of our proposed use of SSR is 382,464 gates, which is less than 0.2% of the total area of NVIDIA's 6800 GT (an estimated 222 million transistors(Medvedev and Budankov 2004)).

4.2 *Dual-Mode IEEE Adders and Multipliers*

The GPGPU and scientific computing communities would like to have the ability to perform double-precision calculations on the GPU. Unfortunately for them, the gaming industry drives the graphics hardware industry, and games do not currently require double-precision. We present a method here that can satisfy the demands of the scientific community without compromising the performance of the single-precision path so crucial to video game performance.

A *dual-mode* floating point unit is a small-scale reconfigurable unit capable of performing two simultaneous single-precision operations or one double-precision operation. Dual-mode units can be fully pipelined to produce results every cycle. Like other SSR units, dual-mode multipliers and adders require internal multiplexers for path selection. Additionally, they require a rounding unit capable of flexible rounding modes. The total additional structure for this modification is insignificant (Even et al. 1997). These units are also capable of operating at the modest 400 MHz clock speed of our baseline architecture.

---

[1]All area estimates are given in terms of inverter-equivalent gate area unless otherwise specified.

4.2.1   Target SSR architecture. We simulate a pipeline in Qsilver that uses dual-mode multipliers and adders in the fragment engine, where we replace pairs of single-precision FPUs in the baseline architecture with a single corresponding dual-mode FPU. This effectively gives us an 8-wide double-precision fragment engine with approximately half the throughput of the single-precision configuration. Double-precision configuration also requires that we retask pairs of 32-bit registers as single 64-bit registers. With half as many fragment pipelines, each double-precision pipe has the same number of available 64-bit registers as each single-precision pipe has 32-bit registers (four 32-bit registers per fragment in the case of NV4x GPUs (Kilgariff and Fernando 2005)). By a similar argument, the bandwidth requirements for the memory and register bus systems in 8-wide double-precision mode should not exceed those of the original 16-wide single-precision configuration.

We have conservative area estimates for a double-precision adder and multiplier of 13,456 gates and 37,056 gates, respectively. The real overhead here comes from replacing each pair of single-precision FPUs with one dual-mode FPU, at an approximate cost of 815,744 gates over the entire fragment engine, or 0.4% of the 6800 GT's total area. Note that we have modified only the multiplication and addition units, so additional precision is not available for specialized operations such as logarithms or square roots. Although many scientific applications would benefit greatly from high precision addition and multiplication alone, a full double-precision arithmetic engine would be ideal. Dual-mode reciprocal, square-root, logarithm, and other specialized units are a topic for future exploration.

Table 2. Single- and double-precision GPGPU computations using SSR. Each application comes with the Brook distribution. The *32-bit cycles* row shows the GPU cycle count for our NV4x-like architecture. Note that these timings are identical whether we are using a dual-mode unit configured in single-precision mode or a dedicated single-precision unit. The *64-bit cycles* row shows the cycles required for double-precision after reconfiguration. As expected, none of the programs takes more than twice as long with double-precision than with single-precision.

Benchmark	bitonic_sort	image_proc	particle_cloth	volume_division
32-bit cycles	620	1,252	19,504	254,923,418
64-bit cycles	1177	2,445	38,959	509,846,783
32→64-bit speedup	.527	.512	.501	.500

4.2.2   Full pipeline simulation. To validate our SSR-based graphics architecture capable of both single- and double-precision, we traced the four Brook benchmarks through Qsilver. Results are summarized in Table 2. This table lists the cycle counts for each application in both single- and double-precision modes. Note that the double-precision calculations never require more than twice as long as the corresponding single-precision calculation. Because the

timing results are identical for dual-mode units configured in single-precision mode and dedicated single-precision units, we have shown that by using SSR we can add double-precision addition and multiplication to the graphics pipeline with only a modest increase in gate count and without affecting the performance of the commonly-used single-precision path.

5 Conclusions

We have extended Qsilver to record information on fragment program state in its annotated trace. Our modified Qsilver core then uses this new information, along with fragment program listings and timing information, to model the programmable fragment engine of an NV4x-like architecture. With this framework in place, we have demonstrated the applicability of Small-Scale Reconfigurability to graphics architectures. We have shown that it is possible to increase the throughput of the fragment engine with only a small increase in die area. In addition, we have demonstrated that dual-mode multipliers and adders can provide double-precision in the fragment engine to support scientific computing in the GPGPU community with no detriment to the gamers who drive the market. The vector-like operations performed on GPUs make them a particularly good target for such techniques, since need for reconfiguration is rare in SIMD environments, and since the cost of reconfiguration is amortized over many operations.

6 Future Work

The fragment engine is one of many elements of the graphics pipeline. Applications of SSR will likely yield similar performance improvements in other units as well. Another area of exploration that is likely to be fruitful for SSR is power consumption. Whenever portions of a chip are unused, they use no dynamic power, but they leak static power. By their very nature, SSR components are rarely idle, and should therefore leak a minimum of static power. Power leakage is currently a major issue with GPUs, and reducing leakage becomes crucial as continuing improvements in chip manufacturing technology exacerbate this problem (Sheaffer et al. 2004).

7 Acknowledgments

REFERENCES

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Han-rahan, P. (2004), 'Brook for GPUs: Stream computing on graphics hardware', *ACM Transactions on Graphics* .

Chiou, L.-Y., Bhunia, S. and Roy, K. (2005), 'Synthesis of application-specific highly efficient multi-mode cores for embedded systems', *ACM Transactions on Embedded Computing Systems* .

Chiricescu, S., Schuette, M., Glinton, R. and Schmit, H. (2002), Morphable multi-pliers, *in* 'Proceedings of the International Conference on Field Programmable Logic and Applications'.

Compton, K. and Hauck, S. (2004), Flexibility measurement of domain-specific recon-figurable hardware, *in* 'Proceedings of the ACM/SIGDA Symposium on Field-programmable Gate Arrays'.

Even, G., Mueller, S. M. and Seidel, P.-M. (1997), A dual mode IEEE multiplier, *in* 'Proceedings of the International Conference on Innovative Systems in Silicon'.

Guerra, L. M., Potkonjak, M. and Rabaey, J. M. (1998), 'Behavioral-level synthesis of heterogeneous BISR reconfigurable ASIC's', *IEEE Transactions on VLSI* .

Humphreys, G., Houston, M., Ng, R., Ahern, S., Frank, R., Kirchner, P. and Klosowski, J. T. (2002), 'Chromium: A stream processing framework for in-teractive graphics on clusters of workstations', *ACM Transactions on Graphics* **21**(3), 693–702.

Kilgariff, E. and Fernando, R. (2005), *The GeForce 6 Series GPU Architecture*, Addison-Wesley Pub Co, pp. 471–491.

Kim, K., Karri, R. and Potkonjak, M. (1997), Synthesis of application specific pro-grammable processors, *in* 'Proceedings of Design Automation'.

Medvedev, A. and Budankov, K. (2004), 'NVIDIA GeForce 6800 Ultra (NV40)'. `http://www.digit-life.com/articles2/gffx/nv40-part1-a.html`.

Seifert, A. (2004), 'NV40 technology explained'. `http://3dcenter.org/artikel/nv40_pipeline/index3_e.php`.

Sheaffer, J. W., Luebke, D. P. and Skadron, K. (2004), A flexible simulation frame-work for graphics architectures, *in* 'Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware'.

Sheaffer, J. W., Skadron, K. and Luebke, D. P. (2005), Studying thermal management for graphics-processor architectures, *in* 'Proceedings of 2005 IEEE International Symposium on Performance Analysis of Systems and Software'.

Vijay Kumar, V. and Lach, J. (2003), Designing, scheduling, and allocating flexible arithmetic components, *in* 'Proceedings of the International Conference on Field Programmable Logic and Applications'.