Programmable and Scalable Architecture for Graphics Processing Units

Carlos S. de La Lama[1], Pekka Jääskeläinen[2], and Jarmo Takala[2]

[1] Universidad Rey Juan Carlos,
Department of Computer Architecture,
Computer Science and Artificial Intelligence,
C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain
carlos.delalama@urjc.es
[2] Tampere University of Technology,
Department of Computer Systems,
Korkeakoulunkatu 10, 33720 Tampere, Finland
pekka.jaaskelainen@tut.fi, jarmo.takala@tut.fi

Abstract. Graphics processing is an application area with high level of parallelism at the data level and at the task level. Therefore, graphics processing units (GPU) are often implemented as multiprocessing systems with high performance floating point processing and application specific hardware stages for maximizing the graphics throughput.
In this paper we evaluate the suitability of Transport Triggered Architectures (TTA) as a basis for implementing GPUs. TTA improves scalability over the traditional VLIW-style architectures making it interesting for computationally intensive applications. We show that TTA provides high floating point processing performance while allowing more programming freedom than vector processors.
Finally, one of the main features of the presented TTA-based GPU design is its fully programmable architecture making it suitable target for general purpose computing on GPU APIs which have become popular in recent years.

Key words: GPU, GPGPU, TTA, VLIW, LLVM, GLSL, OpenGL

1 Introduction

3D graphics processing can be seen as a compound of sequential stages applied to a set of input data. Commonly, graphics processing systems are abstracted as so called *graphics pipelines*, with only minor differences between the various existing APIs and implementations. Therefore, *stream processing* [1], where a number of kernels (user defined or fixed) are applied to a stream of data of the same type, is often thought as the computing paradigm of graphics processing units.

Early 3D accelerating GPUs were essentially designed to perform a fixed set of operations in an effective manner, with no capabilities to customize this process [2]. Later, some vendors started to add programmability to their GPU

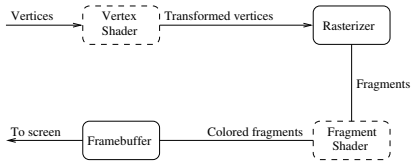

Fig. 1. Simplified view of the customizable OpenGL pipeline.

products, leading to standardization of "shading languages". Both of the major graphics APIs (OpenGL and DirectX) proposed their own implementation of such languages. DirectX introduced the *High Level Shading Language* [3], while OpenGL defined the *OpenGL Shading Language* (GLSL) [4], first supported as an optional extension to OpenGL 1.4 and later becoming part of the standard in OpenGL 2.0.

GLSL is similar to the standard C language, but includes some additional data types for vectors and matrices, and library functions to perform the common operations with the data types. Programs written in GLSL (called *shaders*) can customize the behavior of two specific stages of the OpenGL graphics pipeline (dashed boxes in Figure 1) [5]. *Vertex shaders* are applied to the input points defining the vertices of the graphics primitives (such as points, lines or polygons) in a 3D coordinate system called *model space*.

Depending on the type of primitive being drawn, the rasterizer then generates a number of visible points between the transformed vertices. These new points are called *fragments*. Each drawn primitive usually produces the equal number of fragments as there are covered pixels on the screen. The rasterizer interpolates several attributes, such as color or texture coordinates, between vertices to find the corresponding value (called *varying*) for each fragment, and the programmable *fragment shader* can postprocess and modify those values.

The movement to allow programming of parts of the graphics pipeline led to GPU vendors providing custom APIs for using their GPUs for more general purpose computing (GPGPU) [6], extending the application domain of GPUs to a wide range of programs with highly parallelizable computation. Finally, in the end of 2008, a vendor-neutral API for programming heterogeneous platforms (which can include also GPU-like resources) was standardized. The OpenCL standard [7] was welcomed by the GPGPU community as a generic alternative to platform-specific GPGPU APIs such as NVIDIA's CUDA. [8]

This paper presents a design work in progress of a programmable and scalable GPU architecture based on the Transport Triggered Architecture (TTA), a class of VLIW architectures. The proposed architecture we call TTAGPU is fully programmable and implements all of the graphics pipeline in software. TTAGPU can be scaled at the instruction and task level to produce GPUs with varying size/performance ratio, enabling its use in both embedded and desktop systems. Furthermore, the full programmability allows it to be adapted for GPGPU style of computation, and, for example, to support the OpenCL API. While common practice in GPU design goes through the intensive use of data-parallel models,

our approach tries to exploit parallelism at instruction level, thus avoiding the programmability penalty caused by SIMD operations.

The rest of the paper is organized as follows. Section 2 discusses briefly the related work, Section 3 describes the main points in the TTAGPU design, Section 4 provides some preliminary results on the floating point scalability of the architecture, and Section 5 concludes the paper and discusses the future directions.

2 Related Work

The first generation of programmable GPUs included specialized hardware for vertex processing and fragment processing as separate components, together with texture mapping units and rasterizers, set up on a multi-way stream configuration to exploit the inherent parallelism present on 3D graphic algorithms.

As modern applications needed to customize the graphic processing to a higher degree, it became obvious that such heterogeneous architectures were not the ideal choice. Therefore, with the appearance of the *unified shader model* in 2007 [9], the differences between vertex and fragment shaders begun to disappear. Newer devices have a number of *unified shaders* that can do the same arithmetic operations and access the same buffers (although some differences in the instruction sets are still present). This provides better programmability to the graphic pipeline, while the fixed hardware on critical parts (like the rasterizer) ensures high performance. However, the stream-like connectivity between computing resources still limits the customization of the processing algorithm. The major GPU vendors (NVIDIA & ATI) follow this approach in their latest products [10, 11].

The performance of the unified shader is evaluated in [12] by means of implementing a generic GPU microarchitecture and simulating it. The main conclusion of the paper is that although graphical performance improves only marginally with respect to non-unified shader architectures, it has real benefits in terms of efficiency per area. The shader performance analysis in the paper uses shaders implemented with the OpenGL ARB assembly-like low level language. Although already approved by the Architecture Review Board, this is still an extension to the OpenGL standard, while GLSL is already part of it, which is why we have used it as our shader program input language. Furthermore, new trends on parallel non-graphical computations on GPUs are geared towards using high level languages.

A different approach to achieve GPU flexibility is being proposed by Intel with its Larrabee processor [13]. Instead of starting from a traditional GPU architecture, they propose a x86-compatible device with additional floating point units for enhanced arithmetic performance. Larrabee includes very little specific hardware, the most notable exception being the texture mapping unit. Instead, the graphics pipeline is implemented in software, making it easier to modify and customize. Larrabee is to be deployed as a “many-core” solution, with number

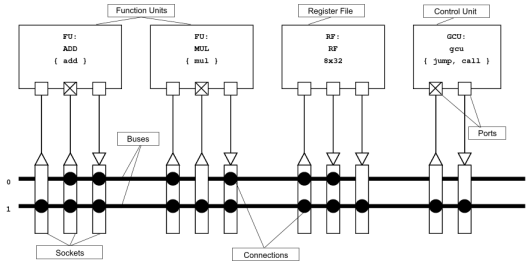

Fig. 2. Example of a TTA processor.

of cores in 64 and more. Each core comprises a 512-bit vector FPU capable of 16 simultaneous single-precision floating point operations.

3 TTAGPU Architecture

The goal of the TTAGPU design is to implement an OpenGL-compliant graphics API which is accelerated with a customized TTA processor, supports the programming of the graphic pipeline as described in the OpenGL 2.1 specification [14] (GLSL coded shaders) and allows high-level language programmability especially with support for OpenCL API in mind. Therefore, the design follows a software-based approach, similar to Larrabee, with additional flexibility provided through programmability. However, as not being tied to the x86 architecture, the datapath resource set can be customized more freely to accelerate the GPU application domain.

3.1 Transport Triggered Architectures

VLIWs are considered interesting processor alternatives for applications with high requirements for data processing performance[15] and with limited control flow, such as graphics processing.

Transport Triggered Architectures (TTA) is a modular processor architecture template with high resemblance to VLIW architectures. The main difference between TTAs and VLIWs can be seen in how they are programmed: instead of defining which operations are started in which function units (FU) at which instruction cycles, TTA programs are defined as data transports between register files (RF) and FUs of the datapath. The operations are started as side-effects of writing operand data to the "triggering port" of the FU. Figure 2 presents a simple example TTA processor.[16]

The programming model of VLIW imposes limitations for scaling the number of FUs in the datapath. Upscaling the number of FUs has been problematic in VLIWs due to the need to include as many write and read ports in the RFs as there are FU operations potentially completed and started at the same time. Additional ports increase the RF complexity, resulting in larger area and critical

path delay. Also, adding an FU to the VLIW datapath requires potentially new bypassing paths to be added from the FU's output ports to the input ports of the other FUs in the datapath, which increases the interconnection network complexity. Thanks to its programmer-visible interconnection network, TTA datapath can support more FUs with simpler RFs [17]. Because the scheduling of data transports between datapath units are programmer-defined, there is no obligation to scale the number of RF ports according to the number of FUs [18]. In addition, the datapath connectivity can be tailored according to the application at hand, adding only the bypassing paths that benefit the application the most.

In order to support fast automated design of TTA processors, a toolset project called TTA-based Codesign Environment (TCE) was started in 2003 in Tampere University of Technology [19]. TCE provides a full design flow from software written in C code down to parallel TTA program image and VHDL implementation of the processor. However, as TTAGPU was evaluated only at architectural level for this paper, the most important tools used in the design were its cycle-accurate instruction set simulator and the compiler, both of which automatically adapt to the set of machine resources in the designed processors.

Because TTA is a statically scheduled architecture with high level of detail exposed to the programmer, the runtime efficiency of the end results produced with the design toolset depends heavily on the quality of the compiler. TCE uses the LLVM Compiler Infrastructure [20] as the backbone for its compiler tool chain (later referred to as 'tcecc'), thus benefits from its global optimizations such as aggressive dead code elimination and link time inlining. In addition, the TCE code generator includes an efficient instruction scheduler with TTA-specific optimizations, and a register allocator optimized to produce better instruction-level parallelism for the post-pass scheduler.

3.2 Scaling on the Instruction Level

The TTAGPU OpenGL implementation is structured into two clearly separated parts. First part is the API layer, which is meant to run on the main CPU on the real scenario. It communicates with the GPU by a command FIFO, each command having a maximum of 4 floating-point arguments. Second part is the software implementation of the OpenGL graphics pipeline running in the TTA. We have tried to minimize the number of buffers to make the pipeline stages as long as possible, as this gives the compiler more optimization opportunities.

The OpenGL graphics pipeline code includes both the software implementation of the pipeline routines itself, in addition to the user defined shader programs defined with GLSL. For the graphics pipeline code, we have so far implemented a limited version capable of doing simple rendering, allowing us to link against real OpenGL demos with no application code modification. Because tcecc already supports compilation of C and C++, it is possible to compile the user-defined GLSL code with little additional effort by using C++ operator overloading and a simple preprocessor, and merge the shader code with the C implementation of the graphics pipeline.

Compiling GLSL code together with the C-based implementation of the graphics pipeline allows user-provided shaders to override the programmable parts, while providing an additional advantage of global optimizations and code specialization that is done after the final program linking. For example, if a custom shader program does not use a result produced by some of the fixed functionality of the graphics pipeline code, the pipeline code will be removed by the dead code elimination optimization. That is, certain types of fragment shader programs compiled with the pipeline code can lead, to higher rasterizer performance.

Preliminary profiling of the current software graphics pipeline implementation showed that the bottleneck so far is on the rasterizer, and, depending on its complexity, on the user-defined fragment shader. This makes sense as the data density on the pipeline explodes after rasterizing as usually a high number of fragments are generated by each primitive. For example, a line can be defined using two vertices from which the rasterizer produces fragments enough to represent all the visible pixels between the two vertices. Thus, in TTAGPU we concentrated on optimizing the rasterizer stage by creating a specialized rasterizer loop which processes 16 fragments at a time.

The combined rasterizer/custom fragment shader loop (pseudocode shown in Fig. 3) is fully unrolled by the compiler, implementing effectively a combined 16-way rasterizer and fragment processor on software. The aggressive procedure inlining converts the fully unrolled loop to a single big basic block with the actual rasterizer code producing a fragment and the user defined fragment shader processing it without the need for large buffers between the stages. In addition, the unrolled loop bodies can be often made completely independent from each other, improving potential for high level of ILP exposed to the instruction scheduler. In order to avoid extra control flow in the loop which makes it harder to extract instruction level parallelism (ILP) statically, we always process 16 fragments at a time "speculatively" and discard the possible extra fragments at the end of computation.

```
for i = 1...16 do
   f = produce_fragment() // the rasterizer code
   f = glsl_fragment_processor(f)
   write_to_framebuffer_fifo(f)
```

Fig. 3. Pseudocode of the combined rasterizer/fragment shader loop body.

3.3 Scaling on the Task Level

In order to achieve scalability on the task level, we placed hardware-based FIFO buffers at certain points in the software graphics pipeline. The idea is to add "frontiers" at suitable positions of the pipeline allowing multiple processors to

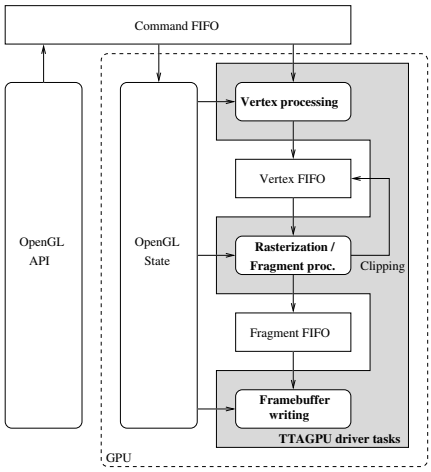

Fig. 4. High-level software structure.

produce and process the FIFO items arbitrarily. It should be noted, however, that it is completely possible in this configuration that the same processor produces and processes the items in the FIFOs. In this type of single core setting, the hardware FIFO merely reduces memory accesses required to pass data between the graphics pipeline stages.

The guidelines followed when placing these buffers were: 1) separate stages with different data densities, 2) place the FIFOs in such positions that the potential for ILP at each stage is as high as possible, and 3) compile the user-defined shader code and related graphics pipeline code together to maximize code specialization and ILP.

These three points are met by placing two hardware FIFOs in the pipeline. One after the vertex processing, as the number of processed vertices needed for primitive rasterizing changes with the different rendering modes (points, lines or polygons), resulting in varying data density. This FIFO allows vertex processing to proceed until enough vertices for primitive processing are available. It also serves as an entry point for new vertices generated during clipping.

The second FIFO is placed after fragment processing, and before the framebuffer writing stage. Framebuffer writing has some additional processing to perform (ownership test, blending, etc.) that cannot be performed completely on per-fragment basis as they depend on the results of previous framebuffer writes. This FIFO allows us to create the highly parallelizable basic block block performing rasterization and fragment processing with no memory writes as the frame buffer writing is done with a custom operation accessing the FIFO.

The hardware-supported FIFOs have a set of status registers that can be used to poll for FIFO emptiness and fullness. This enables us to use light weight co-operative multithreading to hide the FIFO waiting time with processing of

resource	1 FPU	2 FPU	4 FPU	8 FPU	16 FPU
floating point units	1	2	4	8	16
32 bit x 32 register files	1	2	4	8	16
1 bit boolean registers	2	4	8	16	32
transport buses	3	6	12	24	48
integer ALUs	1	1	1	1	1
32 bit load-store units	1	1	1	1	1
32 bit shifters	1	1	1	1	1

Table 1. Resources in the TTAGPU variations.

elements from the other FIFOs. Software implementation structure is shown in Figure 4.

The clean isolation between stages allows the system to connect sets of processors that access the FIFO elements as producers and/or consumers making the system flexible and scalable at the task level. Scaling at the task level can be done simply by adding either identical TTAs or even processors with completely different architectures to the system. The only requirement placed for the added processors is the access to the hardware FIFOs.

4 Results

In order to evaluate the ILP scalability of the TTAGPU in the combined rasterizer/fragment processor loop, we implemented a simple example OpenGL application that renders number of lines randomly to the screen and colors them with a simple fragment shader.

The goal of this experiment was to see how well the single TTAGPU cores scale at the instruction level only by adding multitudes of resource sets to the architecture and recompiling the software using tcecc. The resource set we used for scaling included a single FPU, three transport buses, and a register file with 32 general purpose 32 bit registers. The resources in the benchmarked TTAGPU variations are listed in Table 1.

In order to produce realistic cycle counts for floating point code, we used the pipeline model of the MIPS R4000 floating point units of which description was available in literature [21]. The unit includes eight floating-point operations that share eleven different pipeline resources. However, our benchmark used only addition, division, multiplication and comparison of floating point values.

The benchmark was executed using the TCE cycle-accurate processor architecture simulator for TTAGPUs with the different number of resource sets. Figure 5 shows the speedup improvements in the unrolled rasterizer loop from just adding multiples of the "scaling resource sets" to the machine and recompiling the code. This figure indicates that the ILP scalability of the heavily utilized rasterizer loop is almost linear thanks to the aggressive global optimizations and a register allocator that avoids the reuse of registers as much as possible, reducing the number of false dependencies limiting the parallelization between the

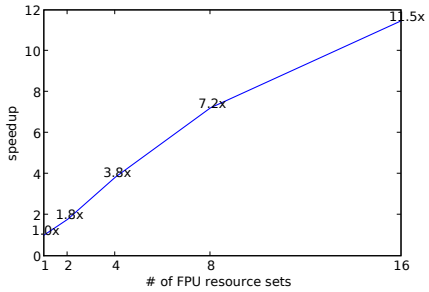

Fig. 5. Scalability of the rasterizer loop with different number of floating point resources.

loop iterations. The scaling gets worse when getting closer to the 16 FPUs version because a hard limit of about 500 general purpose registers in our compiler, and because the loop was implemented with only 16 iterations. With a larger iteration count there would be more operations with which to hide the latencies of the previous iterations.

5 Conclusions

In this paper we have proposed a mainly software-based implementation of a graphics processing unit based on the scalable TTA architecture. We have shown TTA is an interesting alternative to be used for applications where high data processing performance is required, as is the case with GPUs. TTA provides improved scalability at the instruction level in comparison to VLIWs, due to its programmer-visible interconnection network.

The scalability of the proposed TTAGPU on both the task and the instruction level makes the system an interesting platform also to be considered for other data parallel applications designed to be executed on GPU-type platforms. Evaluating the proposed TTAGPU platform for supporting applications written using the OpenCL 1.0 standard [7] is being worked on. Additional future work includes completing the OpenGL API implementation, evaluating the multi-core performance of TTAGPU and implementing an actual hardware prototype.

Acknowledgments. This research was partially funded by the Academy of Finland, the Nokia Foundation and Finnish Center for International Mobility (CIMO).

References

1. Stephens, R.: A survey of stream processing. Acta Informatica **34**(7) (jul 1997) 491 – 541

2. Crow, T.S.: Evolution of the Graphical Processing Unit. Master's thesis, University of Nevada, Reno, NV (dec 2004)
3. St-Laurent, S.: The Complete Effect and HLSL Guide. Paradoxal Press (2005)
4. Kessenich, J.: The OpenGL Shading Language. 3DLabs, Inc. (7 sep 2006)
5. Luebke, D., Humphreys, G.: How GPUs work. Computer **40**(2) (feb 2007) 96–100
6. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum **26**(1) (mar 2007) 80–113
7. Khronos Group: OpenCL 1.0 Specification. (Februrary 2009) `http://www.khronos.org/registry/cl/`.
8. Halfhill, T.R.: Parallel Processing with CUDA. Microprocessor Report (jan 2008)
9. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro **28**(2) (mar 2008) 39–55
10. Wasson, S.: NVIDIA's GeForce 8800 graphics processor. Tech Report (nov 2007)
11. Wasson, S.: AMD Radeon HD 2900 XT graphics processor: R600 revealed. Tech Report (may 2007)
12. Moya, V., Gonzlez, C., Roca, J., Fernndez, A., Espasa, R.: Shader Performance Analisys on a Modern GPU Architecture. In: 38th IEEE/ACM Int. Symp. Microarchitecture, Barcelona, Spain, IEEE Computer Society (12–16 nov 2005)
13. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. ACM Transactions on Graphics **27**(18) (aug 2008)
14. Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification. Silicon Graphics, Inc. (30 jul 2006)
15. Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.B., Rodman, P.K.: A VLIW architecture for a trace scheduling compiler. In: ASPLOS-II: Proc. second int. conf. on Architectual support for programming languages and operating systems, Los Alamitos, CA, USA, IEEE Computer Society Press (1987) 180–192
16. Corporaal, H.: Microprocessor Architectures: from VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
17. Corporaal, H.: TTAs: missing the ILP complexity wall. Journal of Systems Architecture **45**(12-13) (1999) 949–973
18. Hoogerbrugge, J., Corporaal, H.: Register file port requirements of Transport Triggered Architectures. In: MICRO 27: Proc. 27th Int. Symp. Microarchitecture, New York, NY, USA, ACM Press (1994) 191–195
19. Jääskeläinen, P., Guzma, V., Cilio, A., Takala, J.: Codesign toolset for application-specific instruction-set processors. In: Proc. Multimedia on Mobile Devices 2007. (2007) 65070X–1 – 65070X–11 `http://tce.cs.tut.fi/`.
20. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. Int. Symp. Code Generation and Optimization, Palo Alto, CA (March 20–24 2004) 75
21. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)