Programming with Relaxed Streams

Univ. of Virginia Dept. of Comp. Sci. Tech Report CS-2007-17

Dec. 2007

Jiayuan Meng
jm6dg@cs.virginia.edu

Shahrukh Rohinton Tarapore*
starapor@atl.lmco.com

Shuai Che
sc5nf@cs.virginia.edu

Jiawei Huang
jh3wn@cs.virginia.edu

Jeremy W. Sheaffer
jws9c@cs.virginia.edu

Kevin Skadron
skadron@cs.virginia.edu

University of Virginia
Computer Science

ABSTRACT

Diminishing returns in single thread performance have forced a reevaluation of priorities in microprocessor design. Recent architectures have foregone deeper pipelining in favor of multiple cores per chip and multiple threads per core. The day approaches when processors with hundreds or thousands of cores are commonplace, but programming models for these *manycore* architectures lag far behind the architectures themselves. We are developing *Fractal*, a manycore architecture and associated programming model we call *relaxed streaming*. Relaxed streaming allows flexible and convenient stream access, implicit memory management and dependency enforcement, and the decoupling of sequential and parallel phases of execution. This paper presents relaxed streaming in the context of our Fractal API, discussing the benefits of a relaxed streaming model over more traditional streaming models, especially in terms of convenience and ease of use.

1. INTRODUCTION

As we move into an era of manycore microprocessor architecture, data movement becomes a far more challenging problem. Cores may spend more time waiting for data than actually computing with it. Of course, conventional cache systems alleviate this concern somewhat; however, write-allocation, inclusive hierarchies, complicated coherence protocols with high overhead, and false sharing situations all lead to unnecessary data transfer. Streaming provides a semantic notion that simplifies data movement. Its single assignment nature guarantees coherency, and its predictable access patterns define and localize a thread's global data working set. With streams, data movement is addressed in an explicit way, which potentially leads to performance gains and increased energy efficiency. Our *pattern* concept, introduced in Section 2.1, capitalizes on this behavior and provides a useful abstraction.

Existing streaming models include many useful features, but often package them with undesirable restrictions. Stanford's *Imagine* [8] uses *StreamC* and *KernelC* [4] to program streaming applications. In these models, streams are represented as one-dimensional FIFO (first in, first out) queues. Kernels are parallel, homogeneous operations applied to the stream elements. *StreamIt* [17] also models one-dimensional streams, but is somewhat less restrictive in that it allows an indexed *peek* operation on elements in the queue. Unfortunately, StreamIt does not support stream reuse; reusing a stream requires an explicit, high-overhead copy operation. *Brook* [2] extends streams to accommodate indexable structures—but not indexable streams!—its workspace is based on the shape of its input streams, requiring all input streams to share the same dimensions.

While supporting indexable streams as Brook does, our Fractal API also decouples the kernel workspace from the concept of the stream, allowing a relaxed stream access model. The workspace is defined on a multi-dimensional grid we call the *domain*. Traditional streaming is implemented over the domain, allowing threads to have access to multiple elements in each input and output stream. Moreover, the dimensions of a stream have no bearing on a workspace, which is strictly dictated by the domain. As a result, we can define a kernel that operates on a subset of stream elements (Listing 5 gives an example of this) without first creating a stream with like dimensions. This also enables streams to have multiple producer kernels writing to different elements of the same stream and eases the handling of various boundary conditions.

Graphics processors (GPUs) presented the first, and to some extent still the only, commodity architecture capable of accelerating data-parallel and streaming workloads. GPUs are not streaming architectures—though they do share much in common—but rather, special purpose computing devices designed to accelerate real-time raster-based rendering. GPUs started to introduce programmability in some functional units with NVIDIA's NV30 series hardware in 2003. Early *GPU programmers* were forced to recast data-parallel problems as rendering, shoehorning compute problems into an OpenGL [15] model to use graphics processors as general purpose processing engines (*GPGPU*). NVIDIA soon released

*Now with Lockheed Martin Advanced Technology Laboratories

Cg [12], a language for writing graphics shaders. Before Cg, all shaders had to be written in assembly, but even Cg was designed for shading geometry in raster graphics, not for GPGPU, leaving GPU programmers confined to a very restrictive, data-parallel model inside a graphics interface. Brook has been ported to the GPU, using both OpenGL and Microsoft's *Direct3D* interfaces.

More recently, NVIDIA released *CUDA* (Compute Unified Device Architecture), a language—with hardware support—designed specifically for GPGPU [13]. CUDA removes the graphics interface model from GPU programming and offers a more open memory model, including random access read and write. CUDA forms groups of shader cores into what NVIDIA terms "multiprocessors" or *MPs*. These MPs share on chip storage to accommodate large streams while providing fast access. CUDA 1.1 allows concurrent execution of independent kernels—a nice place to map the Fractal API's dependency graphs (Section 2.1)—but inter-kernel communication must be through global memory.

Sequoia provides a nice separation of functionality and memory management for hardware with explicit memory management [5]. Programmers organize their computation and transfer data among different levels of the memory hierarchy explicitly; however, difficulty arises when programmers are confronted with the exposed memory hierarchy. Explicit memory management is a very foreign concept to most programmers, and poses challenges even for experienced programmers. This proves be especially troublesome in the face of tasks, such as mergesort, that aggregate large amount of data.

Ct also provides streaming functionality in its model, and is very similar to the Fractal API in many ways [1]. One important distinction of Ct from our model is found in Ct's thread creation methodology, which bases the number of threads on hardware parameters. Fractal assumes free thread creation and instantiates one per element in the domain.

Our Fractal API addresses many of the restrictions of these earlier programming models, allowing arbitrary computational domains independent of stream dimensions, as well as multiple input and output streams, a memory model that falls between implicit and explicit memory management, and a host of other features that make parallel programming easier. Section 3 discusses these features and the benefits they impart in more detail. The next section defines the concept of *relaxed streaming* and gives an example of a relaxed stream program in the Fractal API.

2. RELAXED STREAMS AND PROGRAMMING MODEL

To define *Relaxed Streaming*, we first define a set of characteristic concepts, then present a programming model.

2.1 Relaxed Streams

A *domain* is an ordered n-space array of vectors (possibly a strided subset of a larger, abstract domain). These vectors contain their own n-space coordinate or address within the domain. *Streams* are n-space ordered arrays of data, and are independent of domains. *Kernels* execute over domains, taking zero or more streams and uniform parameters as inputs, instantiating one thread per domain vector. The *thread index* is the value of that thread's domain vector, and is an implicit kernel parameter (since the set of threads on a kernel, k, and domain, d, is defined by $\{k \times d\}$). A *pattern* is a characteristic of a kernel, k, defining the set of stream elements that may be accessed by a thread executing k, as a function of its thread index (note that this is always statically calculable, since it is finitely bounded by the stream length, though currently we require the programmer to supply a pattern function). Patterns are related to the concept of the *iterator*, but are a new concept in the context of streaming. An example of a pattern is shown in Figure 1. A *stream dependency graph* (SDG) is a bipartite directed graph with stream- and kernel-nodes on opposite sides, stream-kernel edges indicating inputs and kernel-stream edges indicating outputs. Figure 2 shows SDGs for a number of applications that we have implemented.

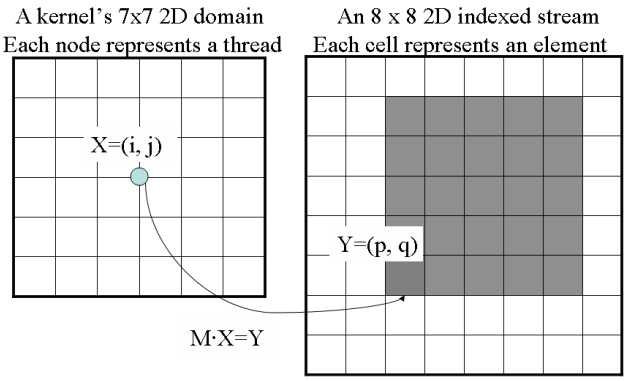

Figure 1: A pattern defines that part of a stream which is assessable by a given thread. Above, a thread with index $\langle i, j \rangle$ is consuming a 5×5 area on a stream, at location $\langle p, q \rangle$. The pattern tells the size and position of the accessed region. The pattern size, $(5, 5)$, is specified explicitly in the program, and the position is stored as a transformation matrix, M, that maps the domain index $\langle i, j \rangle$ to the lower bound, $\langle p, q \rangle$, of its position in the pattern.

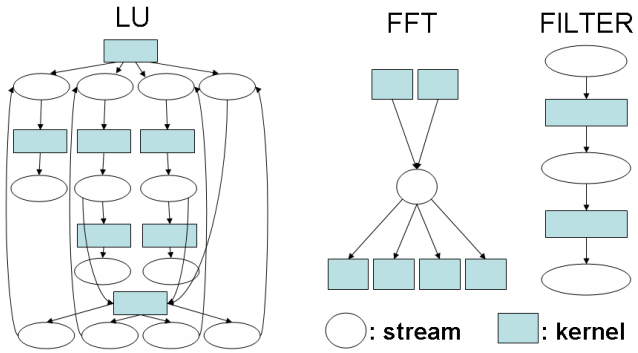

Figure 2: SDGs for three applications: LU decomposition, Fast Fourier Transform, and Edge Detection.

Given the definitions above, *Relaxed Streaming* is a domain-based streaming model, synchronized on SDGs, in which stream access by a thread is restricted to the bounds of that stream's pattern over the thread.

2.2 Programming with Relaxed Streams

Our Fractal API is designed to achieve maximum parallelism on an architecture with a traditional, out-of-order control processor

Listing 1: A definition of a kernel in our relaxed streaming model

```
1  iknl(firFunc, index /* thread index */)
   {
3    kargs /* Argument list */ {
       uparams(float **, coef),
5      ixflow(float, src), /* input stream */
       oxflow(float, dst)  /* output stream */
7    }

9    kbody /* Function body */ {
       long i   = index[0];
11     long j   = index[1];
       long w   = src.dims()[1];
13     float val = 0.f;

15     for(long y = -1; y <= 1; y++) {
         for(long x = -1; x <= 1; x++) {
17         val += xflowR(src, i2D(i + y, j + x, w)) *
                                  coef[y + 1][x + 1];
19       }
       }
21     xflowW(dst, i2D(i, j, w), val);
     }
23 }
```

and an array of simple, slave cores, each with individual L1 and streambuffers and shared L2 (Leverich et al. compare local store based streaming memory to traditional cache-based memory on parallel systems and find that even cache-based systems can benefit from stream programming models [10]). This target processor, which we call *Fractal*, shares features in common with Cell [7], GPUs [11], TRIPS [14], RAW [16], Merrimac [3], Imagine [8], and our API is theoretically compatible with all of these. We aim to build a programming interface that abstracts away the underlying hardware while explicitly exposing parallelism and reducing programmer workload.

Listing 1 demonstrates the implementation of a kernel in our relaxed streaming API. This kernel is used in an image filtering SDG in Listing 2.

Our API is implemented in C++. The kernel definition in Listing 1 starts with the definition of a new kernel, `firFunc`, on line 1. The `kargs` block in lines 3–7 defines three parameters: a uniform, `coef`, and input and output streams, `src` and `dst`, respectively. The kernel body is implemented in the block in lines 9–22. `dims()`, in line 12, is a method defined on streams, which returns an indexable vector containing the dimensions of the stream. `xflowR` and `xflowW` are stream read and write routines. This kernel implements a linear blur filter over a 3-by-3 neighborhood.

The SDG in Listing 2 uses `firFunc` and another kernel, `renormFunc`, to calculate a blur over an image stream, `input` and store it in `output`. Intermediate results are stored in `temp`. Fist, a context is defined for this SDG on line 5. The three streams are defined in lines 8–11. Lines 15–23 instantiate the `renormFunc` kernel. Its domain is specified on 15 and 16. Twenty names the kernel, and 21 and 22 specify the input and output streams with their associated patterns, respectively. The block beginning on line 25 does the same work for the blur filter from Listing 1, but note the passing of a uniform parameter in lines 30 and 31. Thirty-seven sees the launch of the SDG and on line 40 we block until the SDG completes.

Listing 2: An image filtering algorithm in our relaxed streaming model

```
1  void fir(unsigned char *src, float *dst,
           int height,          int width,
3          float coef[3][3])
   {
5    frContext cxt; /* Define a context */

7    /* Then define the streams */
     mem2idxflow<unsigned char> input(src,    2,
9                                     height, width);
     idxflow<float> temp(2, height, width);
11   idxflow2mem<float> output(dst,    2,
                               height, width);
13
     /* Instantiate the kernels */
15   cxt.domain2D(0, height - 1, 1,
                  0, width  - 1, 1) {
17     /* Each renormalization thread consumes *
        * one element and  produces another    *
19      * The pattern is a 2D point.           */
       kernel k_renorm = cxt.genKernel(renormFunc);
21     k_renorm.in(input, point(2));
       k_renorm.out(temp, point(2));
23   }

25   cxt.domain2D(1, height - 2, 1,
                  1, width  - 2, 1) {
27     /* Each filter thread consumes a 3-by-3 *
        * neighborhood of the input stream     *
29      * and produces a single element.       */
       kernel k_fir = cxt.genKernelParam(firFunc,
31                                       coef);
       k_fir.in(ren_img, window2D(3, 3));
33     k_fir.out(edge_img, point(2));
     }
35
     /* Now fire off the SDG */
37   cxt.launch();

39   /* And wait for the result */
     cxt.wait();
41 }
```

3. BENEFITS OF A RELAXED STREAMING MODEL

The Fractal API is implemented in C++. Data and task parallel problems are easily implemented in Fractal, which attempts to address the difficulty of parallel programming with a clean interface and useful abstractions, while allowing programmers to provide high level hints that, combined with the nature of streams, enable the runtime and hardware to implicitly and safely perform useful optimizations.

3.1 Convenience and Expressiveness

Fractal decouples sequential and parallel phases of execution. When implementing Fractal applications, the programmer specifies SDGs and then launches them. Sequential code can continue execution until it voluntarily blocks at a `wait()`. Programmers can think about sequential and parallel code independently and need only consider one synchronization point for an entire SDG, without regard to the complexity of the dependency graph.

Fractal decouples data parallelism from task parallelism. Kernels implement distinct tasks, the kernels themselves independent, the dependencies of their inputs and output streams dictated by the

Listing 3: Sequoia requires explicit memory management with knowledge of hardware parameters

```
instance {
  name     = matmul_mainmem_inst
  task     = matmul :: inner
  runs_at  = main_memory
  calls    = matmul_LS_inst
  tunable U = 128 , X = 64 ,V = 128
}
```

Listing 4: A sparse kernel in Brook requires a copy of the sparse elements to a new stream and uses a dense kernel for computation

```
float src<len>, half_src<len / 2>;
float dst<len / 2>;

streamDomain(half_src, src, 1, 0, len / 2);
kernelFunc(half_src, dst);
```

SDG. At the same time, threads represent fine-grained units of data parallelism. As a result, both levels of parallelism are exposed, but simultaneously distinguished from each other, in a clean and easily understood manner, which leaves space to optimize each specific type of parallelism separately.

Fractal does not require explicit memory management. Maximum stream size is not limited by on-chip storage. Moreover, no explicit double buffering or *direct memory access* (DMA) operations are required. The programmer need only specify high level parameters about stream access patterns for each kernel. *Patterns* describe the stream working set for each thread, which, in our experience, is easy for an average programmer to understand and work with. The patterns give hints to the middleware and hardware to manage data movement. Listing 3 shows a Sequoia *specification mapping*, in which the programmer must specify hardware parameters that Sequoia uses in DMA operations, something which is undesirable, in general, and not necessary in Fractal.

Fractal enables a kernel to access streams sparsely. A decoupled domain relaxes the constraint that one element corresponds to one thread. In addition to *dense kernels* which consume the entire stream element by element, Fractal allows *sparse kernels* that consume or produce a subset of the stream. Sparse kernels are present in many scientific computing applications; LU decomposition is one example. In StreamC [4] and Brook [2], sparse kernels have to be converted to dense kernels by copying a sparse subset of stream elements to a new stream, and using a dense kernel to operate over this derived stream (See Listing 4 for an example in Brook, with a Fractal example in Listing 5).

Fractal arms each thread with visible indices to handle index-dependent control flow. Applications such as image processing and fluid simulation all need special care to handle boundary conditions. In StreamC [4] and Brook [2], streams have to be split explicitly to handle the branch conditions. Fractal provides an alternative for programming convenience. Thread indices are assessable as common variables (Listing 1). This is not only be used to access multiple elements, but also to recognize boundary conditions and influence the thread's control flow.

Listing 5: A sparse kernel is instantiated in Fractal by adjusting domain parameters

```
idxflow<float> src(1, len), dst(1, len / 2);
cxt.domain1D(0, len / 2, 1) {
  kernelFunc.iii(src).ooo(dst);
}
```

Listing 6: Part of an image filtering operator implementation in StreamIt. The neighborhood operation requires explicit copy and gather phases. Compare to the fractal example in Listing 2 which has direct access to the image as a stream

```
float->float splitjoin
  DiffuseImage(float coef[][]) {
  /* Every 9 elements form a stencil */
  split roundrobin(9);
  for(int n = 0; n < ((ROWS - 2) *
                      (COLUMNS - 2)); n++)
    add diffuseFunc(coef);
  join roundrobin;
}

float->float filter diffuseFunc(float coef[][]) {
  work pop 9 push 1 {
    /* Gather the stencil and *
     * compute one output     */
  }
}
```

Fractal allows efficient gathering and scattering. A kernel may gather or scatter from multiple streams or multiple elements within a stream. With decoupled domains and streams in Fractal, a thread can access multiple streams. Each thread in Fractal is able to access any stream elements within the space denoted by associated patterns. StreamIt programs process one stream per kernel, allowing a single stream to be split and merged using primitives such as `split` and `roundrobin`. Brook has similar restrictions. The regrouping and duplication required in both Brook and StreamIt complicates code, reduces programmer productivity, and may introduce complex sources of inefficiency. Listing 6 shows an except from an image filtering application in StreamIt that does split and gather operations to work in a 3-by-3 neighborhood.

Fractal eases constructing complex dependency graphs. No explicit dataflow primitives (e.g. pipeline, split-join, or feedback-loop primitives) are needed. Instead, the SDG is inferred from kernels' producer and consumer relationships. Significant effort is saved when constructing complex SDGs, such as the one present in LU decomposition (Figure 2). Synchronization is an implicit property of the SDG (though additional mutual exclusion primitives are exported by the API for use within kernels). By allowing streams to have elements produced by multiple, distinct kernels, the effort of segmenting and merging streams is reduced, significantly improving programmability and code readability.

3.2 Opportunities for Optimization

Fractal reorganizes dependency graphs, optimizing data management and flow. Fractal middleware can rearrange the SDG in a manner similar to the optimizations described by Gordon et al. [6]. In Gordon's work, granularity coarsening clusters kernels with overlapping working sets into groups that have significantly

reduced communication overhead. Parallel clusters do not communicate, and represent distinct blocks of independent task level parallelism. Kernels within a cluster are assigned to cores with spatial locality. A cluster can then be split into sequences of data parallel kernels, and pipeline parallelism can be further applied to chains of producers and consumers as they are deployed onto different cores. All of these optimizations are achievable within a relaxed streaming model.

Fractal reduces execution overhead for data parallel threads. For architectures that accommodate SIMD execution, performance and energy costs can be reduced by vector execution [9]. MIMD systems can also benefit from the Fractal API with lower overhead in the creation of fine-grained threads. Uniform parameters are dictated once per kernel to each core for the lifetime of that kernel.

Fractal localized global thread working sets to patterns. Patterns define the set of stream elements a thread will potentially access, essentially defining the global working set of each thread. With this information, middleware and hardware can implicitly coordinate DMA transfers and automate bulk load.

Fractal enables efficient on-chip communication. With *patterns* that describe the stream working set of a thread, communication patterns can be deduced. The on-chip network is then able to configure or reserve a virtual circuit to accelerate data movement.

4. CONCLUSIONS AND FUTURE WORK

We have proposed a *relaxed streaming* model that allows programmers to focus on high-level programming problems without being unduly concerned with underlying hardware features. The relaxed streaming model and API presented in this paper are being developed in conjunction with our new Fractal architecture. Fractal is still under heavy development, but we present a brief description of its features here, especially as they relate to relaxed streaming and the Fractal programming model.

4.1 Fractal

Fractal is a MIMD manycore architecture that targets general purpose computation with streaming. It distinguishes a non-inclusive stream storage hierarchy, which includes first level streambuffers and second level spill buffers, from the coherent cache hierarchy. Fractal utilizes the concept of a domain to quickly spawn potentially many thousands of data parallel threads, which greatly aids in realizing the performance potential of parallel resources. Streambuffers can be used as staging areas in temporal multiplexing of kernels, reducing pattern transfer overhead when switching kernels. By aggregating threads that have overlapping access patterns into thread groups and scheduling them on a single core, locality within a single kernel can be increased. Preliminary results show that this yields a speedup of over $2\times$ over non aggregated threads when the number of cores scale up to 32. Fractal allows us to issue bulk loads automatically, but patterns make it possible to make efficient use of indexed streams as end-to-end communication mediums. We are exploring spatial multiplexing of kernels to further exploit temporal locality.

4.2 Future work

Data-dependent streams need to be integrated into the Fractal API. This is useful for applications such as rendering. In triangle rasterization, every three vertices can produce zero to n, where n can be in the millions, *fragments*—an internal representation of data that may later color a pixel in graphics hardware—which need to be added to an output stream. Although we can coarsely define a maximum length, it would be highly desirable to have a variable sized I/O primitive rather than over-provisioning for this type of problem.

The fractal architecture and simulator are still works in progress, and are developed hand-in-hand with with the API and programming model. We are getting preliminary results from this infrastructure, but it is still far from complete. We look forward to performing the types of performance simulations that Fractal will allow.

5. ACKNOWLEDGEMENTS

This research was funded in part by NSF grant no. IIS 0612049. We would also like to thank John Owens for his helpful comments on streaming architectures. The ideas in this paper is built upon initial work done for a course project in CS 754 in Fall 2006 <http://www.cs.virginia.edu/ skadron/wiki/cs754/index.php/Main_Page>

6. REFERENCES

[1] J. F. Anwar Ghuloum, Eric Sprangle. Flexible parallel programming for tera- scale architectures with Ct. White paper, Intel, Apr. 2007.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[3] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[4] A. Das, W. J. Dally, and P. R. Mattson. Compiling for stream processing. In E. R. Altman, K. Skadron, and B. G. Zorn, editors, *PACT*, pages 33–42. ACM, 2006.

[5] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, 2006.

[7] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM Press.

[8] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, Sept. 2002.

[9] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *IEEE Micro*, 24(6):84–90, 2004.

[10] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer*

architecture, pages 358–368, New York, NY, USA, 2007. ACM Press.

[11] D. Luebke and G. Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.

[12] W. R. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, August 2003.

[13] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide, 2007. http://developer.download.nvidia.com/compute/cuda/08/NVIDIA_CUDA_Programming_Guide_0.8.pdf.

[14] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, D. B. J. Huh, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 422–33, June 2003.

[15] M. Segal and K. Akeley, editors. *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*. Silicon Graphics Inc., Oct. 2004.

[16] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.