Parallel Execution of Kahn Process Networks in the GPU

Keith J. Winstein

keithw@mit.edu

Abstract

Modern video cards perform data-parallel operations extremely quickly, but there has been less work toward using the separate "stream processors" of a graphics-processing unit to run separate threads of execution, because there is no support for multiple processes on a GPU. I created an execution environment that runs the processes of a Kahnian network in parallel, on a GPU, and allows inter-process communication through static FIFO channels between processes. Each Kahn process is expressed as a string of bytecode and interpreted by a GPU stream processor. In the future, this kind of system could allow the 128 separate gigahertz processors in a modern consumer video card to run 128 separate virtual machines in parallel, each at speeds approaching a gigahertz. For that to be possible, however, we would need to dramatically speed up the "virtual machine" bytecode interpreter.

1 Introduction

Recently, much interest has focused on programming parallel processors designed for computer graphics. For example, the nVidia GeForce 8800 GTX has 128 scalar "stream processors" clocked at 1.35 GHz and claims a theoretical peak performance of 518 gigaFLOPS. If achievable on general-purpose codes, this kind of performance would put any contemporary microprocessor to shame.

But can we make effective use of these processors, targeted at algorithms with massive data parallelism, for general-purpose problems? Not easily. Ideally, we might execute parallel algorithms on a GPU the same way we do on an SMP array of CPUs — by scheduling threads of execution across its 128 stream processors, with operating-system support for inter-process communication.

There is no easy way to do this, however, because modern GPU manufacturers do not provide an interface to program stream processors directly.[1] Instead, the general-purpose GPU community has had to work through OpenGL or DirectX, two graphics interfaces that provide languages for programmable "shaders" that can be executed on the video card.

A shader[2] is typically an array of identical cellular automata. For each pixel in an output frame, the shader must calculate its color. Typically in general-purpose GPU (or "GPGPU") work, the input to a shader will be a two-dimensional array of pixels encapsulating the current state of the system at time n. The output of the system is another 2D array of pixels — the result of invoking the shader many times, independently and in parallel, once for each output pixel. This represents the state at time $n + 1$.

[1]nVidia has announced, but not released, a "CUDA" C compiler for the GPU, which may improve the situation.

[2]Here I describe "fragment shaders," which operate on pixels. OpenGL and DirectX also provides programmable "vertex shaders," which operate on geometric points. These are less popular in GPGPU work.

While this schema has wide applicability to many scientific and computational problems, it does not suit traditional multithreaded algorithms at all. In a normal multi-tasking system, each thread of execution is able to modify its own region of memory or stack variables, make calls to operating-system service functions (such as syscalls), wait for IO and get woken up by the kernel, etc. By contrast, each parallel invocation of a shader is independent and may only write to one pixel. There is no notion of long-running processes or thread-local storage (even a stack or persistent variables).

I set out to create an execution environment for parallel algorithms on an OpenGL GPU, by implementing these traditional OS functions inside a single shader. My system allows long-running processes, running concurrently, to exist because it interprets in parallel a simple bytecode, saving a process counter and registers for each process so it can return where it left off. Although each process only has access to three registers, the system also provides inter-process communication between processes through FIFO channels. Thus, in practice, a process can have access to as much local state as it wants.

This schema for parallel algorithms (independent processes interacting through a static dataflow graph of FIFOs) was proposed by Kahn[3]. Thus, I describe my system as a parallel execution environment for Kahn process networks. In this paper, I will outline my design and implementation in C++ and OpenGL Shader Language, using Kahn's own "simple parallel program" as an example.

2 Bytecode Design

Ideally, the processes in a Kahnian network would each be expressed in the OpenGL Shader Language, compiled to GPU machine instructions[4], and executed efficiently on the GPU. Unfortunately, that's not possible, because a GPU lacks support for a scheduler or for persistent shader-local storage.

Instead, I created my own simple bytecode to express the processes. Then I implemented a virtual machine for this bytecode in the OpenGL Shader Language, in a single shader. Each process's bytecode (which is immutable) is stored in a row of an array. The first column of each row contains the current "register file" for the process – the process counter, and three 8-bit registers named A, B, and C.[5]

At each step of execution, the shader (which is run in parallel on each "register file") looks up the current instruction in the current row, executes it, and then writes the new state back into the register file. The bytecode instructions are:

JUMP(addr)	Jumps to this address.
STORE(reg, val)	Stores val in reg.
ADD(dst, reg1, reg2)	Adds regs and stores in dst.
JUMP_EQ(a,r1,r2)	Jumps to a if regs are equal.
NEGATE(reg)	Negates register.
WAIT(fifo)	Wait on a FIFO. Puts val in C.
SEND(fifo, reg)	Send the contents of reg on a fifo.

As an example, in Kahn's Algol-like language, he expresses his process g, which waits on a FIFO and sends the integers it receives out to two alternating FIFOs, like this:

[3]The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, Information processing, pages 471-475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[4]These are undocumented; it is not yet possible to program an nVidia or ATI GPU "in the raw."

[5]It would have be possible to provide more registers by using OpenGL's "multitexturing" feature.

```
Process g(integer in U ; integer out V, W) ;
  Begin integer I ; logical B ;
  B := true ;
  Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := ~B ;
    End ;
  End ;
```

In my attached code, the same process is expressed like this:

```
/* Process g */
NEWPROG;
/* 1 */ STORE( A, 1 ); /* A := true */

/* beginning of loop */
/* 2 */ WAIT( 6 ); /* C := WAIT( U ) */
/* 3 */ STORE( B, 1 ); /* to compare B against */
/* 4 */ JUMP_EQ( 7, A, B ); /* jump to 7 if B == 1 */
/* 5 */ SEND( 8, C ); /* send C on W (#8 )*/
/* 6 */ JUMP( 8 );
/* 7 */ SEND( 7, C ); /* send C on V (#7) */
/* 8 */ NEGATE( A );  /* A = ~A */
/* 9 */ JUMP( 2 );
```

In Kahn's system and mine, FIFOs are static connections between pairs of processes. They must be set up and connected in advance. In my system, the declaration looks like this:

```
  FIFO( 4, 2, 0 ); /* FIFO (#4), from proc. 2 to 0. */
```

The special FIFO terminating at node "255" is used to print output to the terminal.

3 Bytecode Implementation

The implementation of the parallelized bytecode interpreter in OpenGL Shader Language posed several difficulties that I had not anticipated. Some of them were niggling details – for instance, although integer types are available in the language, all pixel coordinates, and all values, must be floating point numbers. Recursion is forbidden.

I implement each process as a row of 32-bit cells representing the bytecode. The only cell that is "live" (i.e., that has a shader running on it) is the 0th one in the row — this stores the local state of the process, i.e., its process counter and three 8-bit registers, in a 32-bit cell.

The FIFOs are also implemented as rows of cells, all active. The rightmost cell receives data from processes when they go into the "SEND" state. The leftmost cell stores data that processes will receive when they call WAIT(). Cells pass their data to the left if there is a vacancy.

The biggest difficulty was thinking in the cellular-automaton style of an OpenGL shader, where each parallel invocation writes to only one memory location. How could I implement something like a FIFO, when it is impossible for one thread of execution to read a value into a processes local memory AND update the read pointer of the FIFO? For that matter, how could I even have cells "pass their data to the left," when no one shader thread is able to add data to the left cell AND erase it from the right cell?

The solution was a rule-based scheme that guarantees that those two actions will happen simultaneously, even though they occur in separate shader invocations. For example, when a process hits the WAIT instruction, it adds 128 to its process counter and stores the FIFO number it's waiting on in register C. When is it safe to return a value from WAIT()? Only when (1) the shader invocation running on the register file sees that the FIFO has data available in its leftmost cell, and (2) the shader invocation running on the leftmost cell of the FIFO sees that the process is WAITing on it.

When both of these conditions are true, only then can the shader interpreting the process copy the data from the FIFO and subtract 128 from the process counter. And only then can the leftmost cell of the FIFO erase its data, freeing up that cell to have data passed in from the right. Since both shader invocations fire only on the same set of preconditions, we can guarantee that the actions will occur at the same step. But it is not a convenient way to program. Of course, I should have expected this, since the whole reason for my project is to try to translate between the inconvenient data-parallel world of OpenGL and the Kahnian schema of parallel threads of execution.

4 Kahn's program and performance

In Kahn' 1974 paper, he gives an example of a four-process, five-channel network expressed in an Algol-like language:

- Node f waits for a value on channel Y, prints the result to the console, and sends it on channel X. It then does the same for a value received after waiting on channel Z, and repeats this alternation forever.
- Node g waits for a values on channel X. It alternately sends them on channels $T1$ and $T2$.
- Node h_0 sends an initial 0 on channel Y, then parrots (also to channel Y) everything it gets from waiting on channel $T1$.
- Node h_1 sends an initial 1 on channel Z, then parrots (also to channel Z) everything it gets from waiting on channel $T2$.

Kahn observes that this network should produce an alternative sequence of 0's and 1's. What happens if we run it in my GPU execution environment?

Node f is implemented like this:

```
/* Process f */
NEWPROG;
/* 1 */ STORE( A, 1 ); /* A := true */

/* beginning of loop */
/* 2 */ STORE( B, 1 ); /* to compare A against */
/* 3 */ JUMP_EQ( 6, A, B ); /* jump to 6 if A == 1 */
/* 4 */ WAIT( 5 ); /* C := WAIT( V ) */
/* 5 */ JUMP( 7 ); /* alternate of conditional is over */
/* 6 */ WAIT( 4 ); /* C := WAIT( U ) */
/* 7 */ SEND( 9, C ); /* print to console */
/* 8 */ SEND( 6, C ); /* send on X */
/* 9 */ NEGATE( A );
/* 10 */ JUMP( 2 );
```

The implementation for process g was shown above. Process $h0$ looks like this:

```
/* Process h0 */
NEWPROG;
/* 1 */ STORE( A, 0 ); /* INIT := 0 */
/* 2 */ SEND( 4, A );

/* beginning of loop */
/* 3 */ WAIT( 7 ); /* -> C */
/* 4 */ SEND( 4, C );
/* 5 */ JUMP( 3 );
```

Process $h1$ is the same, except it initializes A to 1 instead of 0, and shuffles values between channels 8 and 5, instead of 7 and 4.

The FIFOs have to be explicitly set up as well, like this:

```
FIFO( 4, 2, 0 ); /* Y */
FIFO( 5, 3, 0 ); /* Z */
FIFO( 6, 0, 1 ); /* X */
FIFO( 7, 1, 2 ); /* T1 */
FIFO( 8, 1, 3 ); /* T2 */
FIFO( 9, 0, 255 ); /* PRINT */
```

When all is said and done, we can run the four processes (and six FIFOs) in parallel on the GPU. What do we get?

```
$ g++ -Wall -g -O2 -o gputest4 gputest4.cpp -lX11 -lGL -lGLU
$ ./gputest4  | head -n 30
0
1
0
1
0
1
0
1
0
1
```

```
0
1
0

⋮
```

The system works! Unfortunately, it works rather slowly — about 600 lines printed per second. There are probably several explanations for the poor performance. First, the GPU I have tested on (an nVidia 6200 TurboCache) only has four hardware threads of execution at 300 MHz, not the 128 at 1.3 GHz on nVidia's current top-of-the-line card. Second, for debugging purposes, the shader sends the system state to the CPU on every clock tick — this is unnecessary and could be removed.

But more fundamentally, interpreting bytecode is slow, in part because the things it requires (in particular, data-directed array indexing) are not what OpenGL drivers and cards generally expect graphics-driven shaders to do. If nVidia's new "CUDA" system provides closer-to-the-metal access to the GPU, perhaps this situation will improve.

5 Conclusions and future work

The major goal of this project was to implement an execution environment for long-running parallel processes, with inter-process communication, in an OpenGL GPU, and this was largely accomplished. However, I also intended to produce a *chip-agnostic* environment for Kahn process networks. That is, what if we could say, at synthesis time, which processes we wanted to execute on the GPU and which on the CPU?

Such a synthesis tool would be provocative because it would allow us to implement a Kahnian network once, and then allow a computer to experiment with running various processes on the CPUs and others on the GPU to see which arrangement produced the best results. Unfortunately, after finishing and debugging the OpenGL interpreter, I did not have time to produce a parallel interpreter in C. This would be a much more straightforward task, however, and the results might still be interesting.

In short, it is possible to graft traditional process-based semantics onto the data-parallel shader model of parallel programming in OpenGL, and the result is that we can synthesize and run independent long-running processes in parallel on a GPU. However, my approach (building an interpreted language that we can interrupt and resume without requiring hardware support) will require significant optimization work if one is looking for acceptable performance.