On the use of GPUs in Mathematical Computation

Matt Skerritt

School of Mathematical and Physical Sciences
University of Newcastle, Australia

Workshop on Mathematics and Computation
University of Newcastle, Australia
19–21 June 2015

1. Introduction
2. Theory
3. Practice
4. Further Work

Section 1

Introduction

The GPU is a "Graphics Processing Unit" sometimes referred to as a video card.

- Originally responsible for putting images on the screen.
- Later gained image processing ability (e.g., vertex and pixel shading).
- More recently have become fully programable.
- Competition in Video Game industry has driven consistent increases in power.

These days every desktop computer has some sort of GPU in it. Sometimes this will be part of the CPU, but often will be an entirely separate add-on card.


00
3
Estus Flask
3598

GPUs use a *SIMD*[1] architecture for highly (massively?) parallel operations. This architecture works best with *data parallel* problems.

- The same code is run many times simultaneously on different data.
- Requires a different mode of thought to single-threaded programming.
- Want to avoid differences in execution paths such as decision branches or loops.

Example (squaring a list of numbers)

Input : list I
Output: list O
begin
    $i \leftarrow$ process number
    $n \leftarrow i^{\text{th}}$ element of I
    i^{th} element of $O \leftarrow n^2$
end

[1] Single Instruction, Multiple Data

GPUs use a *SIMD*[1] architecture for highly (massively?) parallel operations. This architecture works best with *data parallel* problems.

- The same code is run many times simultaneously on different data.
- Requires a different mode of thought to single-threaded programming.
- Want to avoid differences in execution paths such as decision branches or loops.

Example (squaring a list of numbers)

Input : list I
Output: list O
begin
  $i \leftarrow$ process number
  $n \leftarrow i^{\text{th}}$ element of I
  i^{th} element of $O \leftarrow n^2$
end

[1] Single Instruction, Multiple Data

Section 2

Theory

Definition (Parallel Random Access Machine (PRAM))

The PRAM model consists of an unbounded collection of numbered processors

$$P_0, P_1, \ldots$$

and an unbounded collection of shared memory cells

$$C_1, C_2, \ldots$$

Each processor P_i has its own local memory, knows its index i, and can read from and write to the shared memory.

Processors may be activated by some mechanism. Instructions are executed in unit time, synchronised over all active processors.

Some technical issues arise whose solution leads to variants of the model.

Definition

Let M be a PRAM. We say that M computes in *parallel time* $t(n)$ *with* $p(n)$ *processors* if for every input x (encoded with n bits), the machine halts with at most $t(n)$ time steps and activates at most $p(n)$ processors, producing some output.

Definition

Let M be a PRAM. We say that M computes in *sequential time* $t(n)$ if it computes in parallel time $t(n)$ using only a single processor.

Note that each PRAM is assumed to be specific to a problem.

If we encode problems as languages (i.e., as a subset of $\{0,1\}^*$) then we may define the following:

Definition

Let L be a language over $\{0,1\}^*$. The *characteristic function* of L is the function

$$f_L : \{0,1\}^* \to \{0,1\}$$

where $f_L(x) = 1$ if $x \in L$, and $f_L(x) = 0$ if $x \notin L$

Definition

Let L be a language over $\{0,1\}^*$. We say L is *decidable in parallel time* $t(n)$ *with* $p(n)$ *processors* if and only if f_L is computable in parallel time $t(n)$ with $p(n)$ processors.

Sequential time decidability is defined similarly.

We have the following classes of problems:

Definition (Class P)

The class P is the set of all languages L that are decidable in sequential time $n^{O(1)}$

Definition (Class NC)

The class NC is the set of all languages L that are decidable in parallel time $(\log n)^{O(1)}$ with $n^{O(1)}$ processors.

Note that $NC \subseteq P$. It is unknown if $NC = P$.

If we are interested in computing functions directly, instead of language decision problems we have the following analogous classes.

Definition (Class *FP*)

The class *FP* is the set of all functions $f : \{0,1\}^* \to \{0,1\}^*$ that are computable in sequential time $n^{O(1)}$

Definition (Class *FNC*)

The class *FNC* is the set of all functions $f : \{0,1\}^* \to \{0,1\}^*$ that are computable in parallel time $(\log n)^{O(1)}$ with $n^{O(1)}$ processors.

Definition (NC Turing reducibility)

Let B and B' be problems in Class P. We say that B is *NC Turing reducible* to B' if and only if there is a PRAM which solves B with the aid of a unit-cost oracle for problem B', and this PRAM computes B using parallel time $(\log n)^{O(1)}$ with $n^{O(1)}$ processors.

Definition (P-hardness)

A language, L, is *P-hard under NC reducibility* if every $L' \in P$ is *NC* Turing reducible to L.

Definition (P-completeness)

A language, L, is *P-complete under NC reducibility* if $L \in P$ and it is P-hard.

P-completeness is to Class *NC* as *NP*-completeness is to Class P.

Section 3

Practice

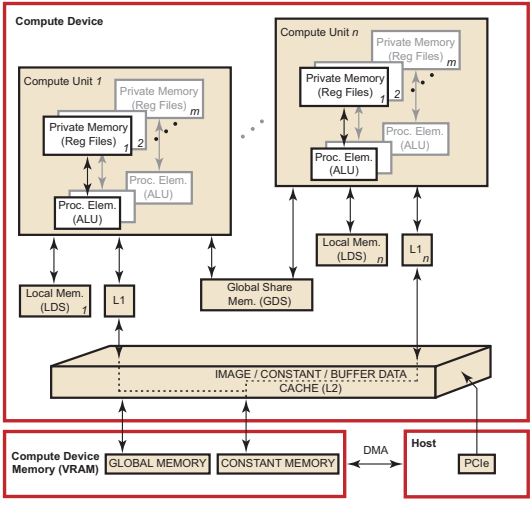
There are two competing API's for general purpose GPU (GPGPU) programming:

- CUDA, which is Nvidia specific.
- OpenCL which is platform agnostic, and can also be used with CPU's and other compute devices.

Both seem otherwise quite similar.

I use OpenCL, and the remainder of this talk will assume OpenCL (inasmuch as it is platform specific at all.)

In practice, a GPU will have an architecture very similar to the following:



Source: AMD Accelerated Parallel Processing OpenCL Programming Guide

A program executing on a GPU is called a *kernel*. It runs as follows:

- Many *work items* will be assigned, each running the kernel in parallel.
- The work items are sub-divided into *work groups*, each of which may end up assigned to different compute units.
- Typically a work group will have many more work items than the compute unit has individual processing capacity.
- Work groups will be further sub-divided into fixed sized blocks (probably 4, 8, 16, or 32). These blocks run in lock step as a unit on the processors in a schedule handled by the hardware.
- The hardware scheduler can hide memory access latency by scheduling other blocks to execute. This is easier with larger work groups.

Example

An *ATI Radeon HD 6770M* reports 512Mb global memory, 6 compute units, a maximum workgroup size of 256, and a local memory size of 32kb.

Example

A Nvidia *GeForce GTX 680MX* reports 2048Mb global memory, 8 compute units, a maximum workgroup size of 1024, and a local memory size of 48kb.

Note that each compute unit likely has something in the vicinity of 8 actual processors on it. The precise details are model specific, and a little hard to verify.

Example

An Intel *HD Graphics 5000* reports 1536Mb global memory, 40 compute units, a maximum workgroup size of 512, and a local memory size of 64kb.

Example

An *Intel(R) Many Integrated Core Acceleration Card* (actually a Xeon Phi) reports 11634Mb global memory, 240 compute units, a maximum workgroup size of 8192, and a local memory size of 32kb.

Some considerations for writing in practice:

- Data must be coped between the computer's memory and the GPUs memory.
- Maximum memory allocatable to a single buffer is less than the global memory (usually a quarter, have seen a third).
- Local memory is much faster than global memory.
- Local memory is shared only between the work items in the same work group.
- Local memory is divided into banks. Simultaneous accesses to the same memory bank by different work items need to be serialised, and cause slowdowns. (This is called a *bank conflict*).
- Synchronisation of work items during execution is only possible within work group.
- Synchronisation of work items outside of work groups requires multiple kernel executions. (Kernel termination is the synchronisation point).

Algorithms based on *NC* problems will need to be adapted to cope with these (and other) considerations.

A single work group functions the closest to a PRAM, although (as we have seen) the processors are not, strictly speaking, running in parallel. The following lemma is applicable.

Lemma (Brent, 1974)

Suppose that a computation can be performed on a PRAM with t time steps using q operations. Then the same computation can be performed with p processors (where p is less than the number of processors needed for maximum concurrency) in

$$t_p \leq t + \frac{q - t}{p}$$

time steps.

A parallel program which needs more work items that can be scheduled in a single work group needs to find a way to break the computation into work group sized chunks, and to combine the chunks in a separate kernel execution.

Example (Prefix Sum)

Note: this is often referred to as a *scan* in the GPU programming literature.

Let $a_1, \ldots, a_n$ be a finite sequence of numbers. We want to compute a new sequence of $p_1, \ldots, p_n$ where

$$p_i = \begin{cases} 0 & \text{for } i = 1 \\ \sum_{k=1}^{i-1} a_k & \text{for } 2 \leq i \leq n \end{cases}$$

This has an easy sequential algorithm.

A naïve parallel implementaion of prefix sum might look like:

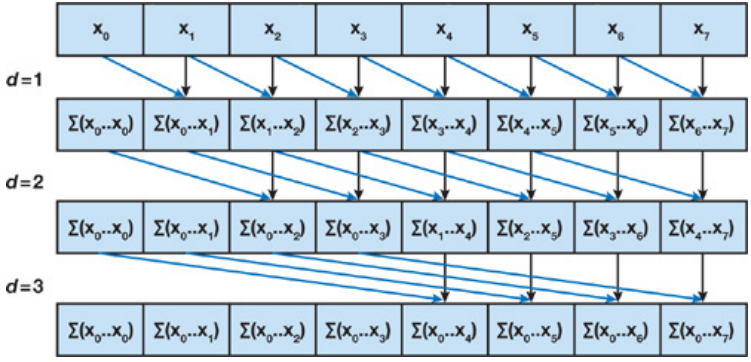
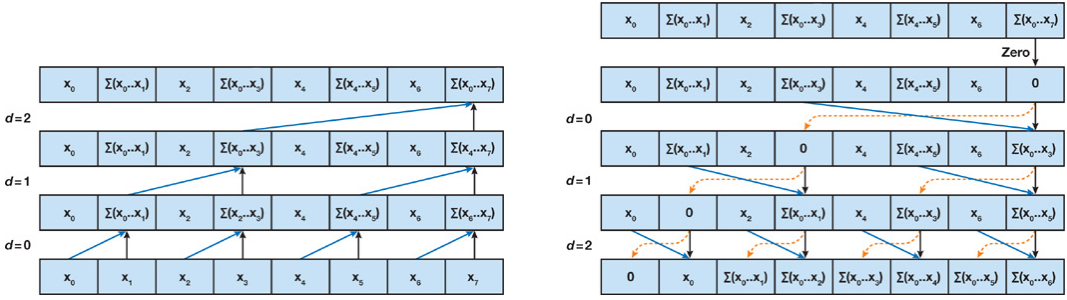


Image from `https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html`

This has good theoreatical parallel time, but in practice performs more total instructions than a sequential scan.

A better implementation uses two phases: an up-sweep (or reduce), and down-sweep:



Images from https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html

Note that this algorithm only works for sequences of size 2^k for somme k, and only on a single work-group. It can be extended to longer sequences at the expense of computing nested prefix sums.[2]

[2] see https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html

Further Work:

- Implement large integer arithmetic, and high precision floating point operations on a GPU
  - In the case where precision is small enough to fit in a single work group (this could be several tens of thousands of decimal digits worth).
  - In the case where precision is too large to fit the numbers in a single workgroup.
- Numerical integration on GPUs.
- Anything else I can get my hands on that looks like it might be implementable on a GPU.

THANKYOU