General Purpose Computing Using Graphics Processing Units (GPGPU Computing)

Stuart Monro
monros@dcs.gla.ac.uk

Terminology

What is a Graphics Processing Unit (GPU)?
A circuit to produce computer graphics.

Parallel Processing or Concurrency?
Parallel processing

Data parallel – where the same process is carried out on all (or a lot of) the data simultaneously.

Task parallel – where different processes are carried out simultaneously (not necessarily using the same data).

Data & task parallel processing will be explained in more detail later

Device vs Host?
Host – CPU
Device – GPU

Motivation

- A lot of research into the use of GPUs to implement parallel programming techniques
- GPUs are now consumer level devices
- Interest in GPU use within HPC rapidly increasing (available and cost effective)
- Parallel programming on GPUs is not straightforward
- Development of an abstract model or framework

History of GPU Computing

1992 – Silicon Graphics release OpenGL

Mid 1990s - release of first person games such as Doom, Duke Nukem 3D & Quake

2001 - Nvidia release GeForce 3 implementing Microsoft DirectX 8.0

2003 – continually improving performance of CPUs begins to slow

2005 – researchers begin to investigate GPUs as alternative platform to support HPC

2006 – Nvidia release the CUDA architecture to support general purpose GPU computing

2008 – OpenCL specification released

CUDA and OpenCL

What are they?
CUDA – Nvidia's parallel computing architecture.
OpenCL – the open equivalent of CUDA

What are they used for?
CUDA – SETI
Protein folding simulation
Password recovery
OpenCL – no real world applications as yet identified but available on Nvidia & AMD devices. Included in Apple's Snow Leopard OS.

Differences between them
OpenCL standard indicates that it will support task as well as data parallelism.
OpenCL not tied to a single architecture
OpenCL is not proprietary, managed by the Khronos group

Current state of play with both
Both are still under development however adoption of CUDA & research into its uses has been more widespread to date.

Why Use a GPU?

Does your program have the following requirements?

- Large computations (lots of number crunching)
- Substantial parallelism (need to get a lot done simultaneously)
- Throughput more important than latency (successful computation over time delay)

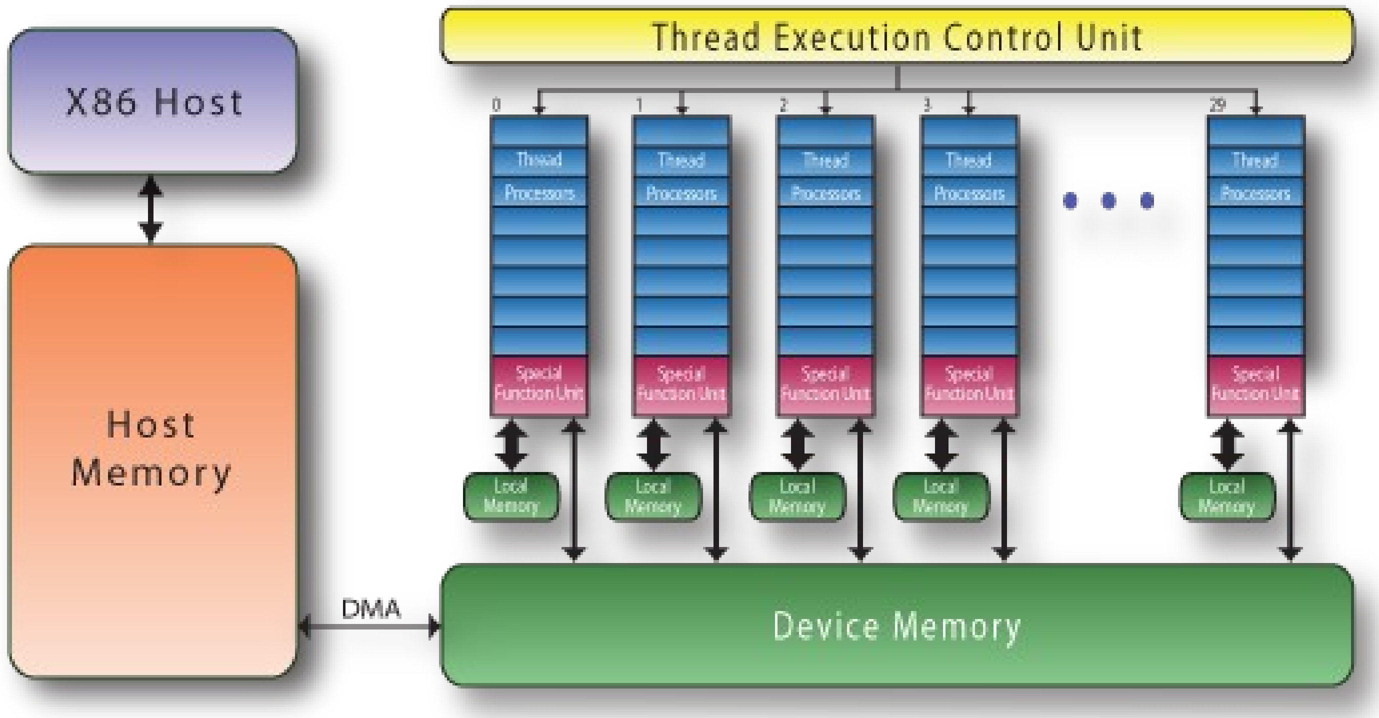
Why Use a GPU?

Three of the top five supercomputers in the world use Nvidia GPUs

Rank	Name	Location	GPUs	Speed
1	Tianhe-1	China	7,168	2.507 PF
3	Nebulae	China	4,640	1.27 PF
4	Tsubame 2.0	Japan	4,200	1.192 PF

Reduced power consumption
Accelerators for specific functions

The Architecture of a GPU

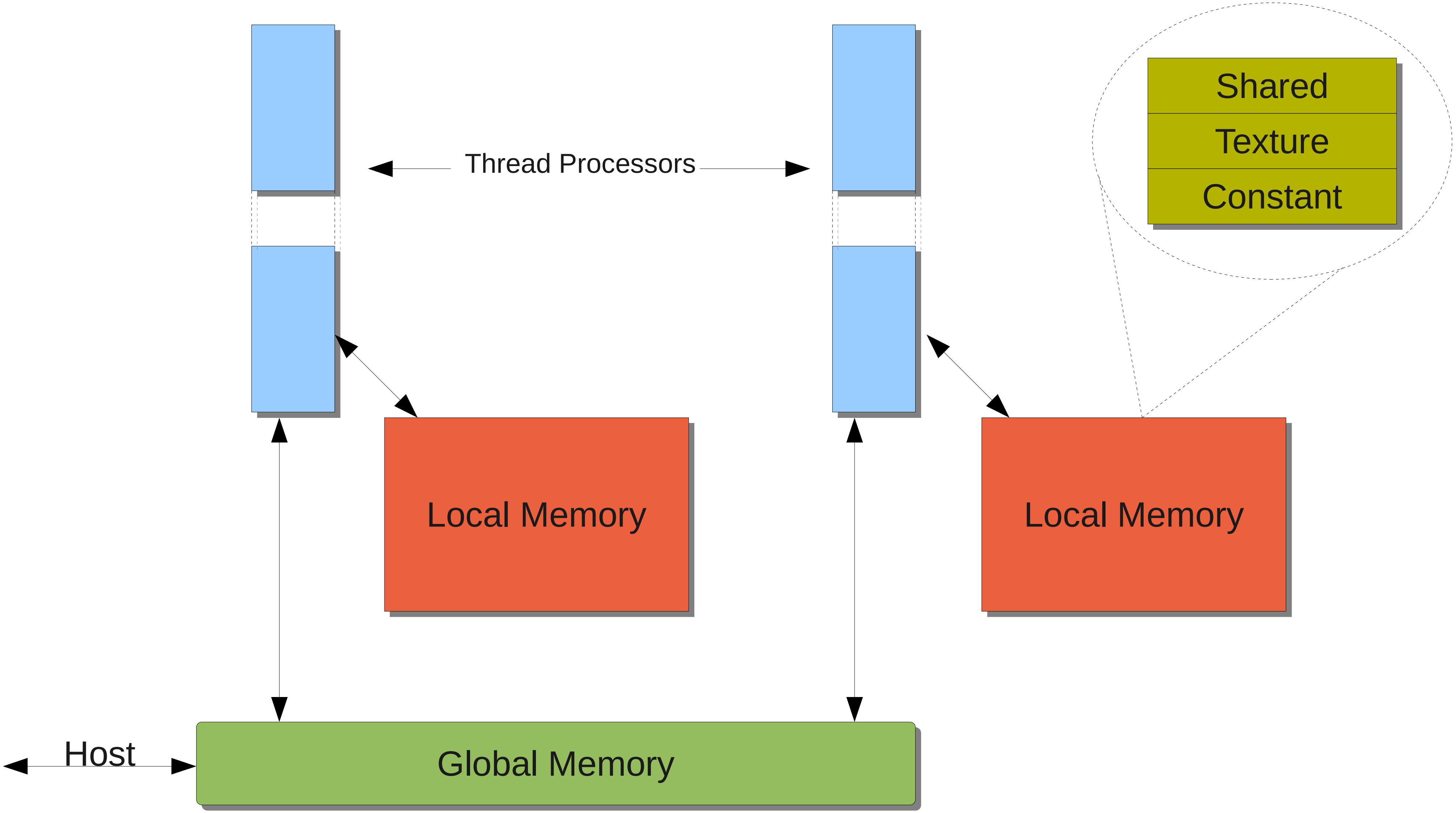


Source: http://www.pgroup.com/lit/articles/insider/v1n1a1.htm

Key features:

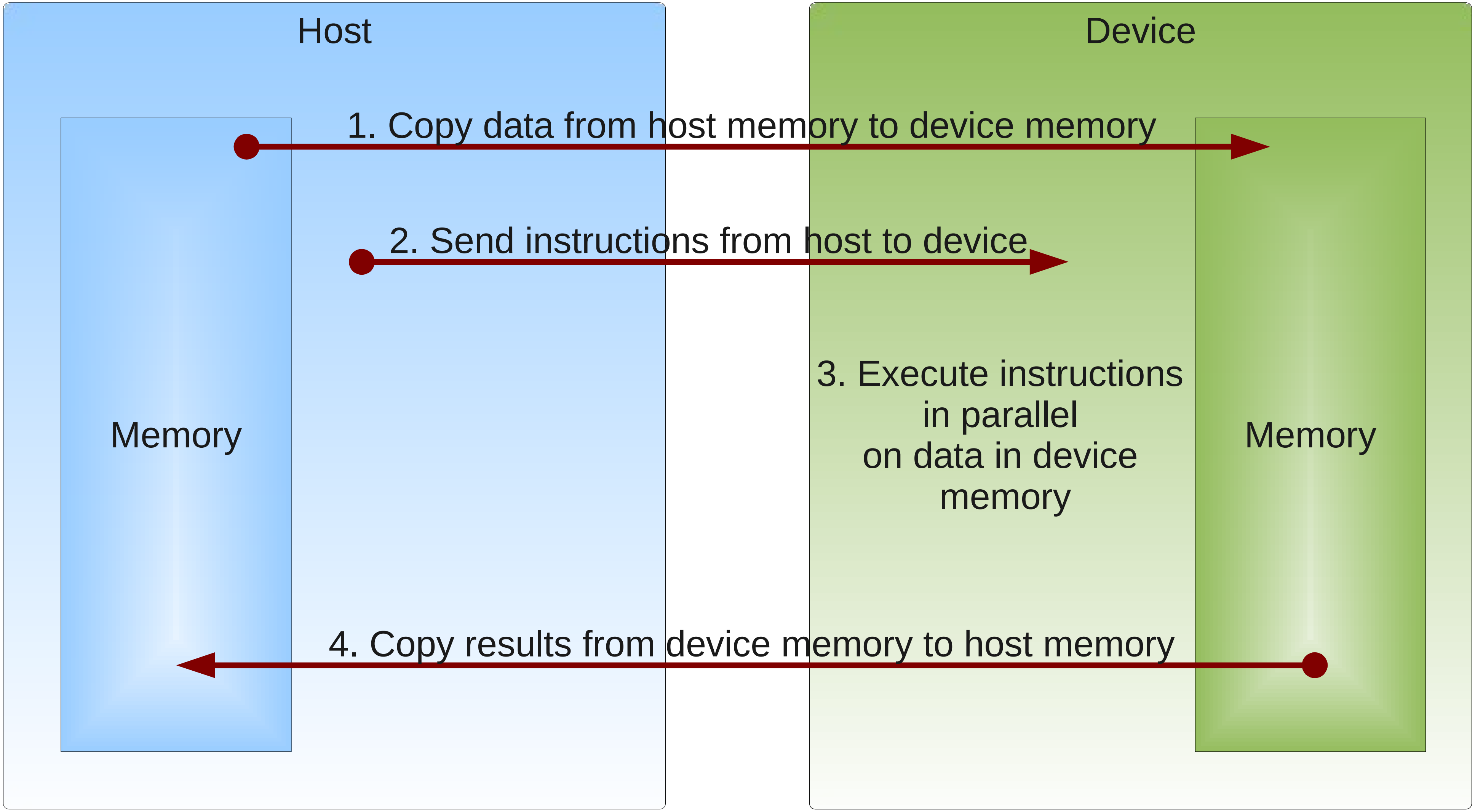
- Processors
- Memory
- Interconnect

GPU Memory



Thread Processors
Shared
Texture
Constant
Local Memory
Local Memory
Host
Global Memory

The Programming Model



CUDA Challenges

- Installing the SDK
- Understanding the model
  - The movement of data & results between host and device
  - Where code should be executed
- Suitability of code for parallelisation
  - Exploitable parallelism
  - Data dependency
- Ensuring memory requirements of the code are achievable
- Understanding & correctly implementing the different memory types on the device

CUDA Challenges (cont)

- Architectural differences between devices – memory, compute capability

- Thread Management
  - Execution flow
  - Same operation in parallel
  - Limited interaction

Current Work

Parallelisation of Standard Algorithms (Vector Addition)

1	2	3	4	+	5	6	7	8	=	6	8	10	12

Sequential approach:

```
    for (i = 0; i < N; i++)
     c[i] = a[i] + b[i];
```

Parallel approach

```
__global__ void vectorAdd(int *a, int *b, int *c)
{
  int bId = blockIdx.x;
  if (bId < NUMOFCALCS)
    {c[bId] = a[bId] + b[bId];}
}
int main()
{
    ...
    vectorAdd<<<NUMOFCALCS,1>>>(dev_a, dev_b, dev_c);
    ...
}
```

Current Work

Cost Model

Developing a formula which can help to predict if program performance will improve or deteriorate through the use of a GPU

$$T(P) = \sum_{i=1}^{r} T(K_i) \text{ sec.}$$

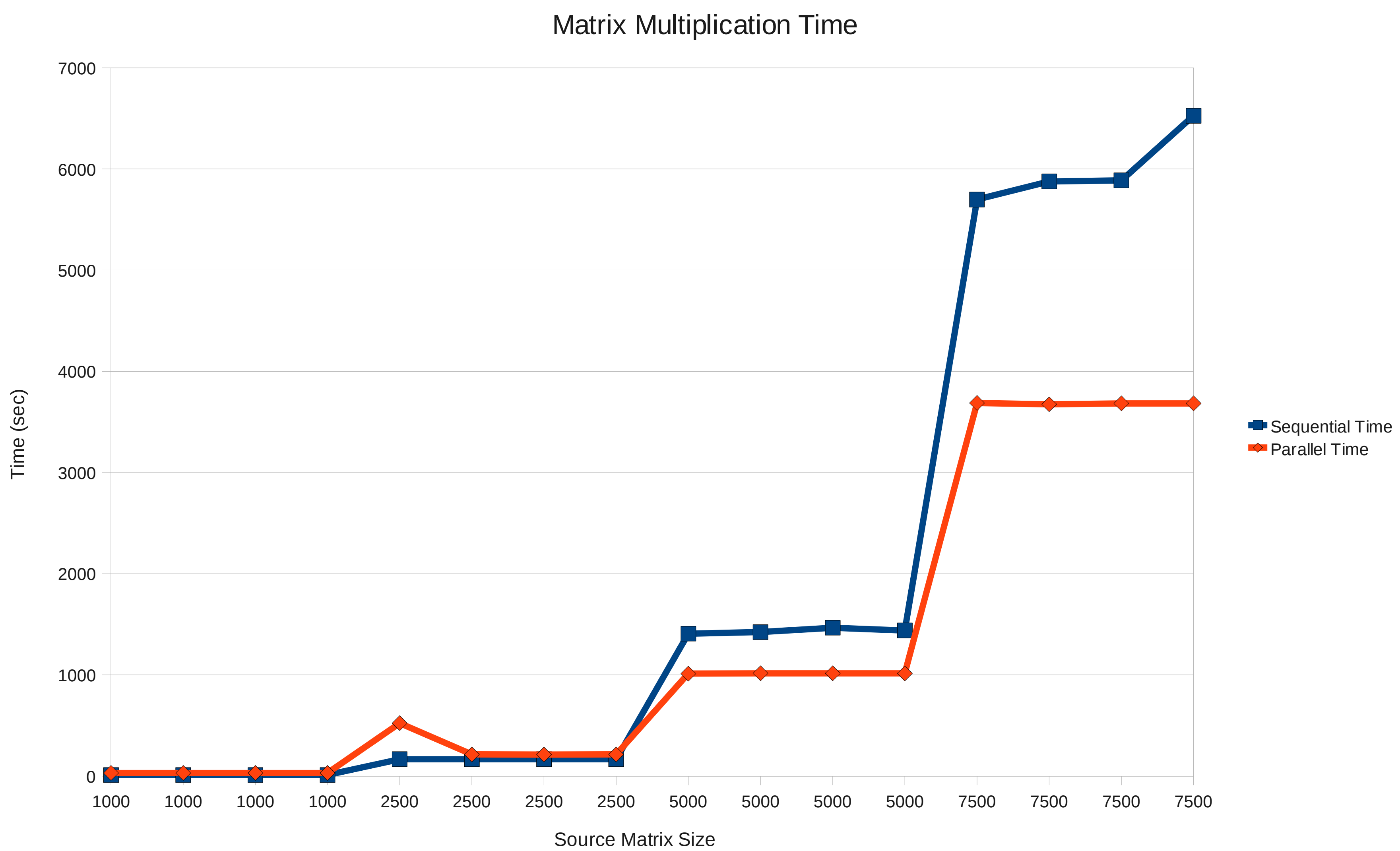
- Cost of computation
- Cost of memory access (global and shared)

- Cost of computations on the CPU
- Cost of communication with CPU
- Texture & constant memory
- Atomic operations

$$T_{\text{pdgemm\_comm}}(n, pr, pc, p) = \log_2 p \cdot \frac{n^2}{p} \cdot \frac{1}{\tau} + \left\lceil \frac{n}{nb} \right\rceil \cdot \log_2 p \cdot \lambda$$

Current Work

Cost Model



Current Work

Abstract Model (A consistent set of concepts for GPU programming)

Develop an abstract model of the code written for the GPU in order to:

Identify (where possible) commonalities
- Allocation of memory, data structures & transfers

Highlight programming challenges & consider possible solutions
- Identification of code suitable for parallelisation
- Identification of code not suitable for the GPU (pointers to pointers)
- Memory restrictions

Determine what options need to be presented to a programmer
- Compute capability
- Memory utilisation (off chip vs on chip)
- Host – device communication optimisation

Thesis

- A cost model can be found which can be used to predict the performance of different data parallel algorithms on different chip architectures
- A data parallel GPU cost model can be combined with an existing CPU cost model
- A programming framework can be developed which will abstract away from the architectural details of the GPU
- That framework can be developed in such a way that the portability of programs between different chip architectures will be possible
- The syntax used within that framework by programmers to express their algorithms will be executable