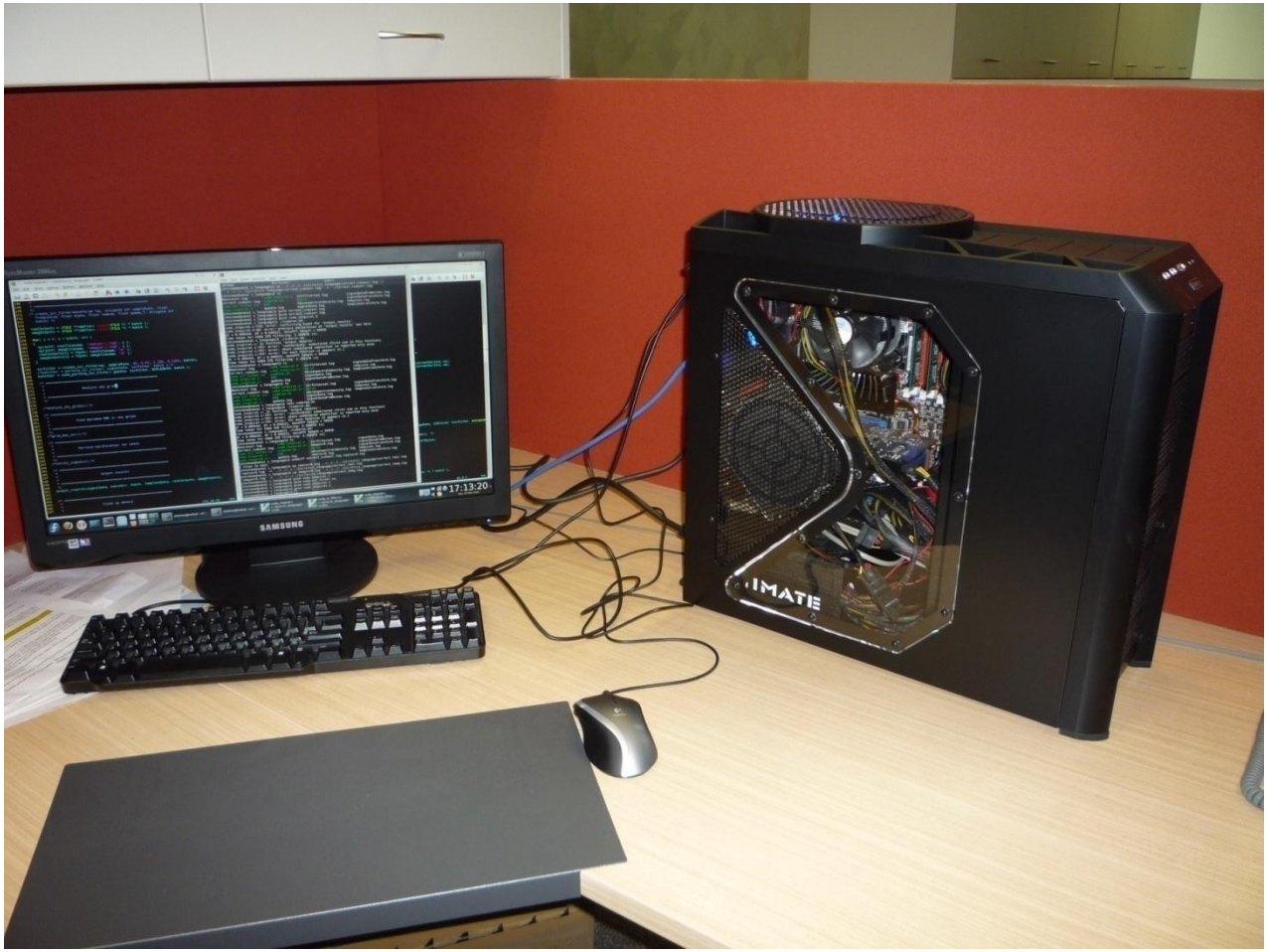

# Brief Introduction to GPU Programming

By Shin Kee Chung

CAASTRO (UWA)

CAASTRO
ARC CENTRE OF EXCELLENCE FOR ALL-SKY ASTROPHYSICS

# Desktop with GPU




# What is GPU?

- GPU stands for Graphics Processing Unit
- Originally developed for graphics rendering
- Its development is mainly driven by the gaming market
- CUDA (Compute Unified Device Architecture) is the GPU programming language developed by NVIDIA
- AMD GPU was not tested due to the lack of library support (such as FFT library)


# GPU Architecture

- Figure 1-2, chapter 1.1, CUDA C Programming Guide.

# GPU and CUDA

- Popular
  - Over 100 million CUDA enabled GPU sold
- Easy to program using CUDA
  - C and C++ Integration
  - Sizeable computing libraries
  - CUDA Matlab Plugins
- Cost effective
  - $400-$500 can provide teraflops performance, or $1000+ for really high performance GPU

# Why CUDA?

- There are two major GPU languages
  - OpenCL
  - CUDA
- OpenCL is more portable, can be run in different brands of GPUs, but has much complicated structure
- CUDA is less portable, but easier to learn, good for picking up concepts
- Note that CUDA will not necessarily run faster in NVIDIA graphics cards.


# CUDA Architecture

- Figure 2-1, chapter 2.2, CUDA C Programming Guide.


# Memory Hierarchy

- Figure 2-2, chapter 2.3, CUDA C Programming Guide.


# Parallel Execution

- Parallelism is the key of GPU programming
  - Extreme similarity with MPI or any other multh-threaded programming
- Only different names

# Example: Simple Inverter

| 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|

| 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|

# Example: Simple Inverter

**C Program**

```
int *run(int *data, int length)
{
    int i;

    for( i = 0; i < length; i++ )
    {
        data[i] = 1 - data[i];
    }
    return data;
}
```

Calculation done in a loop

# Example: Simple Inverter

## CUDA Program, multi-threaded execution

Kernel Function

```
__global__ void invert(int *d_data, int length)
{
    // Getting the thread id, block id and number of threads per block
    int tx = threadIdx.x;
    int bx = blockIdx.x;
    int numThreads = blockDim.x;

    // Inverting the element accessed by each thread
    d_data[bx * numThreads + tx] = 1 - d_data[bx * numThreads + tx];
}
```

No loop is used

Host Function

```
int *run(int *data, int length)
{
        ...
        // Perform inversion, assuming that total threads = array length
        invert<<< blocks, threads >>>( d_data, length );
        // Then copy d_data to data
        ...
        return data;
}
```