EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 1: August 24

Lecturer: Vijay Garg *Scribe: Michael Bartling*

1.1 Introduction

In this class we introduce the concept of concurrent programming using multiple libraries including Thread programming in Java, pthreads in C, openMP, and CUDA (for GPU compute). We also introduce the concept of formal verification of parallel programming. Finally, we end with an example of a set of parallel forms of a simple algorithm.

1.2 Special Note

Dr. Gargś office hours are a gun free zone.

1.3 Parallel Frameworks

Note: all of the following code examples can be found at the course GitHub `https://github.com/mbartling/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter1-threads`

1.3.1 Java

Parallelism in Java can be achieved in one of a few ways. In the first method we just extend the Java Thread class:

```
public class HelloWorldThread extends Thread {
    public void run() {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
        HelloWorldThread t = new HelloWorldThread();
        t.start();
    }
}
```

Another option is to implement `Run()` such as:

```
class Foo {
    String name;
    public Foo(String s) {
        name = s;
```

```
    }
    public void setName(String s) {
        name = s;
    }
    public String getName() {
        return name;
    }
}
class FooBar extends Foo implements Runnable {
    public FooBar(String s) {
        super(s);
    }
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println(getName() + ": Hello World");
    }
    public static void main(String[] args) {
        FooBar f1 = new FooBar("Romeo");
        Thread t1 = new Thread(f1);
        t1.start();
        FooBar f2 = new FooBar("Juliet");
        Thread t2 = new Thread(f2);
        t2.start();
    }
}
```

1.3.2 OpenMP

OpenMP makes trivial parallel paradigms in C, such as parallel for loops and simple threads, well *trivial*. The following code snippet shows a simple task creation, however parallel for loops are about as simple as `parfor` in Matlab. In GCC on Linux you need to use the `-fopenmp` flag to compile with openMP support.

```
#include <stdio.h>
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
```

```
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  } /* All threads join master thread and terminate */

}
```

1.3.3 pthreads

POSIX threads or *pthreads* for short, are a type of thread compatible with the POSIX standard. pthreads are most commonly programmed in C and C++, however they are available in other languages. Furthermore, pthreads are approximately compatible with most common operating systems. Note, in GCC you need to link against the pthreads library to use them, this can be done with the `-lpthread` linker flag.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *Hello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! from thread %d\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     rc = pthread_create(&threads[t], NULL, Hello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   pthread_exit(NULL);
}
```

1.3.4 GPGPU's and CUDA Programming

Traditionally, GPU programming has been *locked in* to graphics oriented tasks. Over time this programming model has shifted towards a more software defined model. Nowadays, GPGPU's, or general purpose graphics processing units, handle many highly parallelizable tasks such as computing the back-propagation

algorithm in Deep Neural Networks, Hogwild gradient descent methods, recursive and iterative ray-tracing, and other performance critical parallel computations. The basic programming paradigm is the *kernel*. The kernel describes both a task and how to distribute this task on the GPU including the resources to utilize. Optimizing the kernel is challenging both for programmers and for compiler writers.

NOTE: CUDA programs can be compiled using `nvcc`.

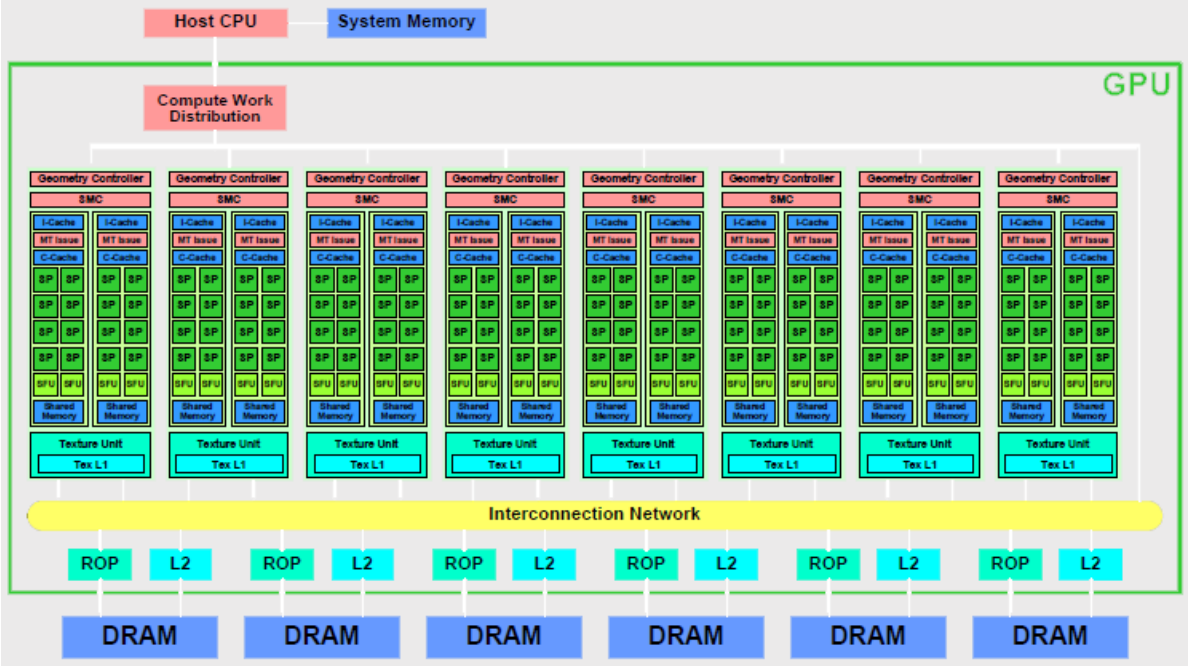

Figure 1.1: GPU Architecture: GPU's are effectively a multicore processor where each *core* (approximately 15-20) has its own set of compute units. These compute units are often confused with cores in the whitepapers (e.g. the Nvidia GTX 980 has 2200 CUDA cores!). The true cores are often referred to as Streaming Multiprocessors.

```
#include <stdio.h>

#define NUM_BLOCKS 16
#define BLOCK_WIDTH 1

__global__ void hello()
{
    printf("Hello world! I'm a thread in block %d\n", blockIdx.x);
}


int main(int argc,char **argv)
{
    // launch the kernel
    hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();
```

```
    printf("That's all!\n");

    return 0;
}
```

1.4 Formal Verification

Oftentimes, determining whether our concurrent model is correct is a non-trivial task. Luckily tools like Promela and Spin exist to formally verify our concurrent model. Spin is used to run promela files and is invoked as following

```
$> spin promelaFile.pml
```

The following is an example of a Promela model:

```
active [2] proctype user()
{
    printf("Hello from thread %d\n", _pid);
}

init {
    printf("Main: Hello from thread %d\n", _pid);
}
```

1.5 Ideal Parallel RAM model: PRAM

We can think of PRAM as an "Ideal Parallel Machine," where each processing element, PE, communicates with a shared memory structure. See 1.5. This model is accessed using multiple methods including:

- CREW
- EREW
- Common CRCW

Where E stands for exclusive, R for read, W for write, and C for concurrent.

1.6 Max Element Algorithm

Suppose we want to find the max element in a sequence or array.

1.6.1 Naive Algorithm

The naive approach is to just access each element 1 time sequentially computing the max. This has the following time and work complexities, assuming all the numbers are unique:

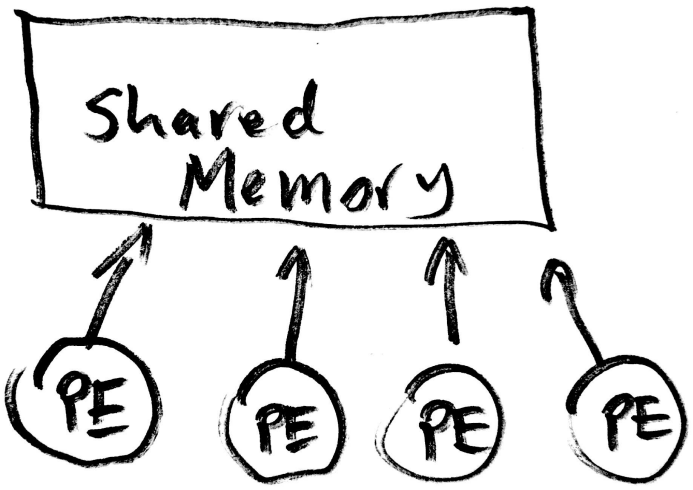

Figure 1.2: PRAM Model

$$T(N) = O(N)$$
$$W(N) = O(N)$$

1.6.2 Binary Tree Algorithm

But can we do better? Yes. If we treat the decision problem as a binary tree then we can get *logn* time complexity. Note, this processes is similar to MapReduce. The basic idea is to divide and conquer. Here, the original list is assigned to a set of workers who make decisions which look like a binary tree. The root of this tree is the max value.

$$T(N) = O(logN)$$
$$W(N) = O(N)$$

Note, we cannot do better than $O(N)$ work complexity since we must visit all nodes. This is just common sense.

1.6.3 Ludicrous Speed

What about an algorithm using as many workers as possible to solve the task? Then we can embed the max value problem into a sort of dependency matrix.

$$T(N) = O(1)$$
$$W(N) = O(N^2)$$

Algorithm 1 Ludicrous Speed Algorithm

```
∀i, isBiggest[i] := 1
for all i, j do
    if A[j] > A[i] then
        isBiggest[i] ← 0
    end if
end for
if isBiggest[i] then
    MAX := A[i]
end if
```