Parallel Computing with GPUs: Optimisation

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

Last Lecture

- All about memory, pointers and storage
- We have seen that C is a low level language
- Now we would like to consider what makes a program fast.

This Lecture

- ❑Optimisation Overview
- ❑Compute Bound
- ❑Memory Bound

When to Optimise

- Is your program complete?
  - If not then don’t start optimising
  - If you haven't started coding then don’t try to perform advanced optimisations until its complete
    - This might be counter intuitive
- Is it worth it?
  - Is your code already fast enough?
  - Are you going to optimise the right bit?
  - What are the likely benefits? Is it cost effective?
    - (number of runs × number of users × time savings × user's salary) - (time spent optimizing × programmer's salary)

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization*** *is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”* Donald Knuth, Computer Programming as an Art (1974)

First step: Profiling

- Which part of the program is the bottleneck
  - This may be obvious if you have a large loop
  - May be less obvious in a complicated program or procedure
- Manually profiling using `time()` function
  - We can time critical aspects of the program using the time command
  - This gives us insight into how long it takes to execute.
- Profiling using a profiler
  - Unix: gprof
  - VS2017: Built in profiler

Profiling with clock() - Windows only

- #include time.h
- The clock() function returns a clock_t value the number of clock ticks elapsed since the program was launched
- To calculate the time in seconds divide by CLOCK_PER_SEC

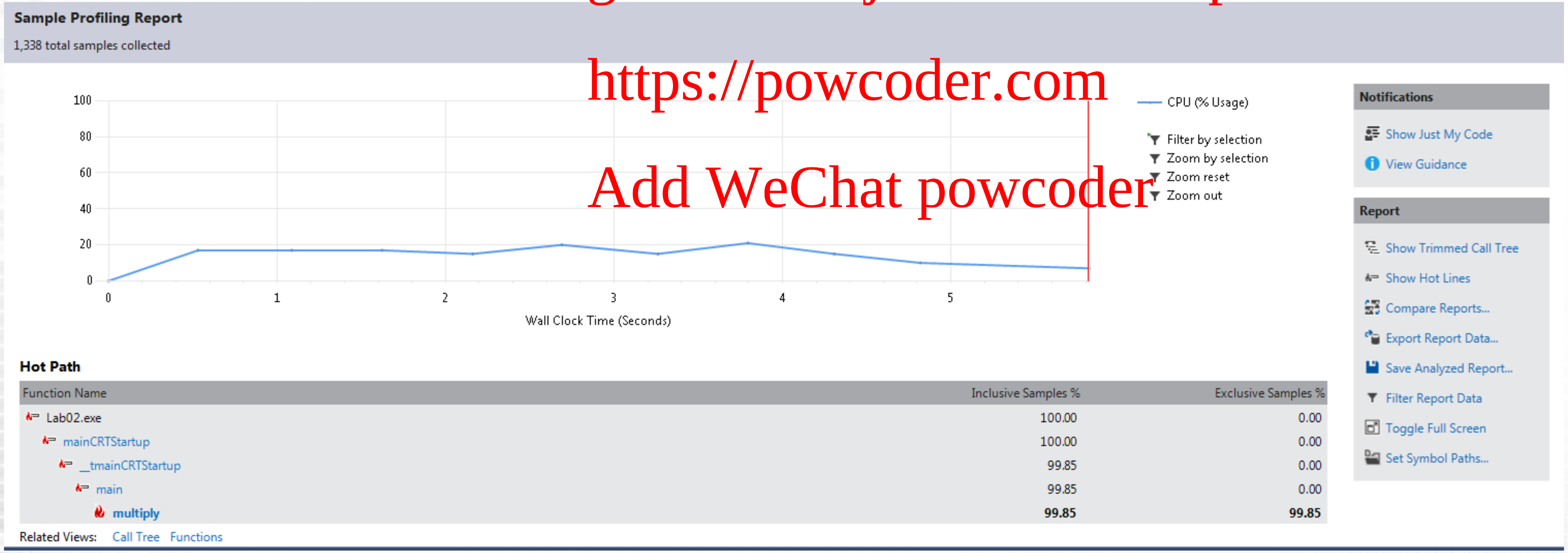

```
clock_t begin, end;
float seconds;

begin = clock();
func();
end = clock();

seconds = (end - begin) / (float)CLOCKS_PER_SEC;
```

VS2017 Profiling Example

- Debug->Performance and Diagnostics
  - Start
  - Select CPU Sampling, Finish (or next and select project)
  - No Data? Your program might not run for long enough to sample

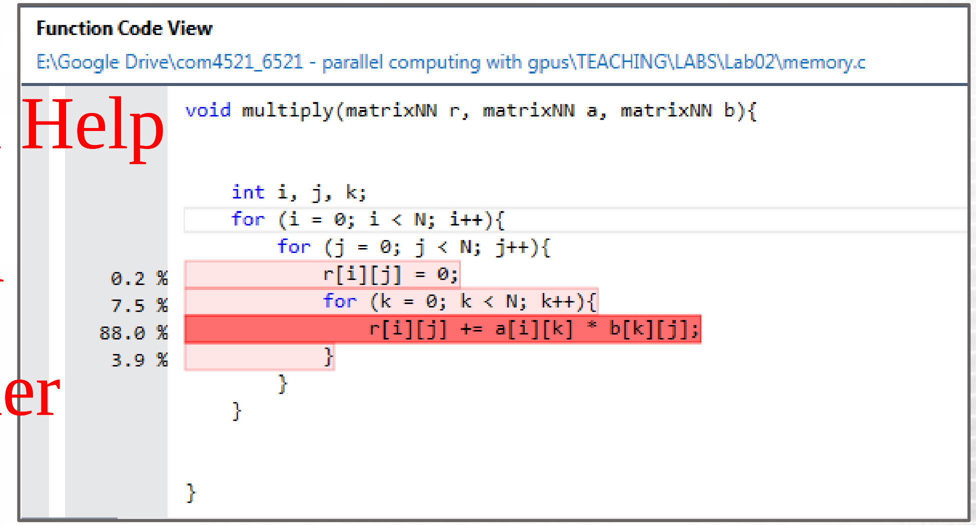


The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

VS2017 Profiling Example

- Samples
  - The profiler interrupts at given time intervals to collect information on the stack
  - Default sampling is 10,000,000 clock cycles
- Inclusive Samples
  - Time samples including any sub call
- Exclusive Samples
  - Time samples excluding any sub calls
- Hot Path
  - Slowest path of execution through the program
    - **Best candidate for optimisation**
  - Select the function for a line-by-line breakdown of sampling percentage



Compute vs Memory Bound

- Compute bound
  - Performance is limited by the speed of the CPU
  - CPU usage is high: typically 100% for extended periods of time
- Memory Bound
  - Performance is limited by the memory access speed

  - CPU usage might be lower
  - Typically the cache usage will be poor
    - poor hit rate if fragmented or random accesses

- ❑ Optimisation Overview
- ❑ Compute Bound
- ❑ Memory Bound

Compute Bound: Optimisation

- Approach 1: Compile with full optimisation
  - msvc compiler is very good at optimising code for efficiency
  - Many of the techniques we will examine can be applied automatically by a compiler.
  - Optimisation: Compiler /O Optimisation property
  - Help the compiler
    - Refactor code to make it clear (clear to users is clear to a compiler)
    - Avoid complicated control flow

Optimisation Level	Description
/O1	Optimises code for minimum size
/O2	Optimises code for maximum speed
/Od	Disables optimisation for debugging
/Oi	Generates intrinsic functions for appropriate calls
/Og	Enables global optimisations

Compute Bound: Optimisation

- Approach 2: Redesign the program
  - Compilers cant do this and it is most likely to have the biggest impact
  - If you have a loop that is executed 1000's of times then find a way to do it without the loop.
  - Be familiar with algorithms
    - Understand big O(n) notation
    - E.g. Sequential search has many faster replacements

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

http://bigocheatsheet.com/

The University Of Sheffield. NVIDIA GPU RESEARCH CENTER

Compute Bound: Optimisations

- Approach 3: Understand operation performance
  - Cost of going to disk is massive
  - Loop Invariant Computations: move operations out of loops where possible
  - Strength reduction: replace expression with cheaper ones

Core i7 Instruction	Cycle Latency
Integer ADD SUB (x32 and x64)	1
Integer MUL (x32 and x64)	3
Integer DIV (x32)	17-28
Integer DIV (x64)	28-90
Floating Point ADD SUB (x32)	3
Floating Point MUL (x32)	5
Floating Point DIV (x32)	7-27

http://www.agner.org/optimize/instruction_tables.pdf

Compute Bound: Optimisations

- Approach 4: function in-lining
  - In-lining increases code size but reduces function calls.
    - Make your simple function a macro
    - Use the _inline operator
  - Be sensible: Not everything should be in-lined


```
float vec2f_len(vec2f a, vec2f b)
{
    vec2f r;
    r.x = a.x - b.x;
    r.y = a.y b.y;
    return (float)sqrt(r.x*r.x + r.y*r.y);  //requires #include <math.h>
}
```

```
#define vec2f_len(a, b) ((float)sqrt((a.x-b.x)*(a.x-b.x) - (a.y-
b.y)*(a.y-b.y)))
```

```
_inline float vec2f_len(vec2f a, vec2f b)
{
    return (float)sqrt((a.x-b.x)*(a.x-b.x) - (a.y-b.y)*(a.y-b.y));
}
```

Compute Bound: Optimisations

- Approach 5: Loop unrolling
  - msvc can do this automatically
  - Reduces the number of branch executions

```
for (int i=0; i<100; i++){
    some_function(i);
}
```



```
for (int i=0; i<100;){
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
}
```

Compute Bound: Optimisations

- Approach 6: Loop jamming
  - Combine adjacent loops to minimise branching (for ranges over the same variable)
  - E.g. Reduction of iterating and testing value `i`


```
for (i=0; i<dim, i++){
    for (j=0;j<dim; j++){
        matrix[i][j] = rand();
    }
}
for (i=0; i<dim, i++){
    matrix[i][i] = 0;
}
```

```
for (i=0; i<dim, i++){
    for (j=0;j<dim; j++){
        matrix[i][j] = rand();
    }
    matrix[i][i] = 0;
}
```

Compute Bound: Optimisations

- Approach 6: Global or heap variables
  - Avoid referencing global or heap variables from within loops
    - Global variables can not be cached in registers
    - Better to write to a local variable
  - Make a local copy of the variable which can be cached
    - Be careful that nothing else requires the variable before you modify it

```
int count;

void test1(void)
{
    int i;
    for(i=0;i<N;i++){
        count += f();
    }
}
void test2(void)
{
    int i, local_count;
    local_count = count;
    for(i=0;i<N;i++){
        local_count += f();
    }
    count = local_count;
}
```

Compute Bound: Optimisations

- Approach 7: Function calls
  - Functions are a good way of modularising code
  - Function calls do however have an overhead
    - Stack and program counter must be manipulated
  - It can be beneficial to avoid function calls within loops


```
void f()
{
    //lots of work
}

void test_f()
{
    int i;
    for(i=0;i<N;i++){
        f();
    }
}
```

```
void g()
{
    int i;
    for(i=0;i<N;i++){
        //lots of work
    }
}

void test_g()
{
    g();
}
```

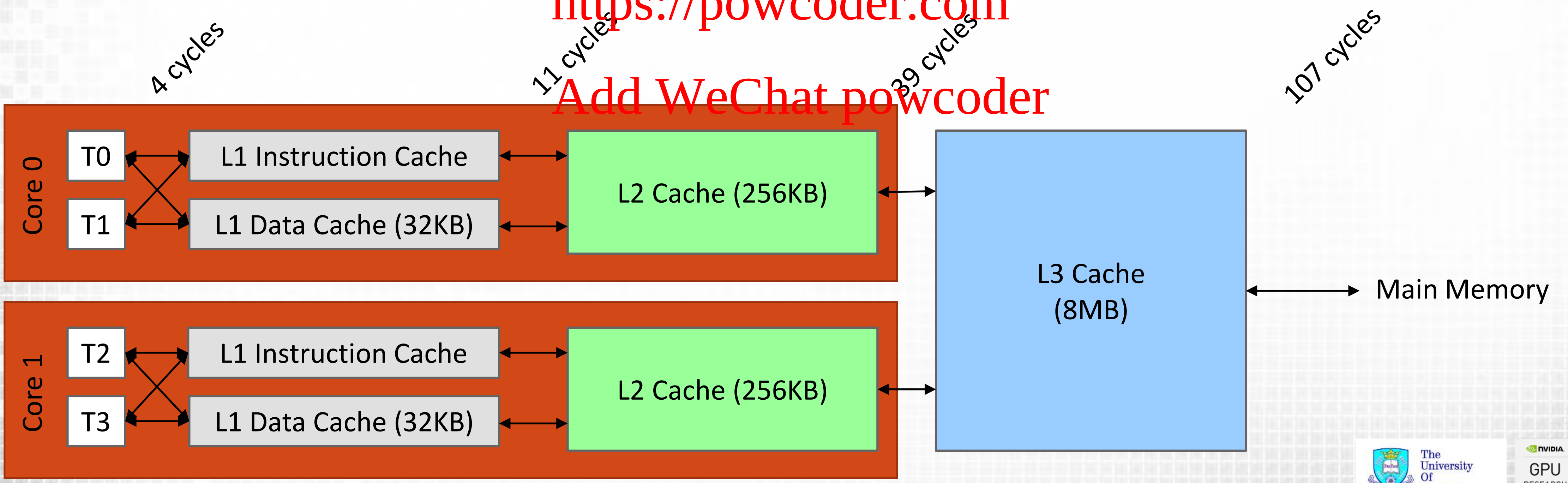
Compute Bound: Optimisations

- Approach 8: Don't over use the stack
  - Loops rather than recursion
    - C compilers are very good at optimising loops
    - Only certain recursive functions can be optimised
    - Function calls increase stack usage
  - Avoid compile time allocation large structures or arrays on the stack
    - E.g. `int x[10000000];`
    - Use the **heap** or global arrays
  - Avoid passing large structures as argument
    - They are copied by value
    - Pass a pointer instead

- ❑ Optimisation Overview
- ❑ Compute Bound
- ❑ Memory Bound

Memory Bound: Optimisation

- ❑Approach 1: Locality of data access
  - ❑This is by far the most important consideration
  - ❑CPU cache is small amount of very fast hierarchical memory
    - ❑Holds contents of recently accessed memory locations
    - ❑MUCH faster than main memory (orders of magnitude)



Memory Bound: Optimisation (Locality)

- Memory is read in cache lines of 64 bytes
  - Accessing a single bytes requires movement of the entire cache line
  - Reading patterns with common locality within cache lines reduced memory movement
    - Fewer wait (or idle) cycles
- Memory lines are pre-fetched
  - Predicable access patterns are good
  - Linear access patterns are **very** cache friendly (predictable and good locality)

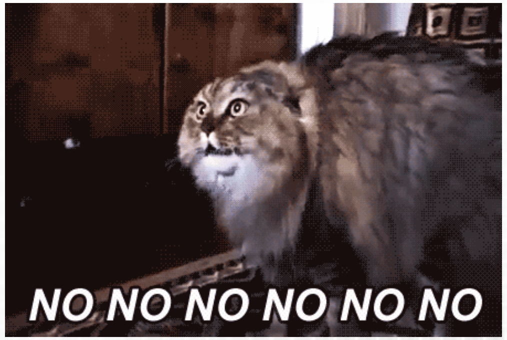
Memory Bound: Optimisation

- ❑Approach 2: Column major access
  - ❑A special case of approach 1
    - ❑Important for FORTRAN users.
  - ❑Column major access has poor utilisation of cache lines
    - ❑Despite predictability only a single value from each cache line is accessed
  - ❑The alternative: row major access
    - ❑Iterate the righter most index first
    - ❑Good utilisation of the cache line


```
float array[N][M];
int i, j;

for (j = 0; j < M; j++){
    for (i = 0; i < N; i++){
        array[i][j] = 0.0f;
    }
}
```

No!



The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Memory Bound: Optimisation

- Approach 3: Nice structures
  - Make your structures cache friendly
    - Multiples of cache size
    - Structures are padded: /Zp (Struct Member Alignment): default
      - **Any member whose size is less than 8 bytes will be at an offset that is a multiple of its own size based on the largest struct variable member size**
      - **any member whose size is 8 bytes or more will be at an offset that is a multiple of 8 bytes**
  - Reduce `struct` size as a result of padding
    - Arrange similar sized structure elements to avoid padding
  - Increase `struct` size to help padding
    - Add chars at the end of your structure to help it align with cache line size


```
struct sa{
    int a;
    char b;
    int c;
    char d;
};
```

```
struct sb{
    int a;
    int c;
    char b;
    char d;
};
```

What is the size of each `struct`?

Memory Bound: Optimisation


```
struct sa{        /* 16 bytes
total */
int  a;        /* 4 bytes */
char b;        /* 1 byte  */
char pad[3]; /* 3 bytes */
int  c;        /* 4 bytes */
char d;        /* 1 byte  */
char pad[3]; /* 3 bytes */
};
```

```
struct sb{         /* 12 bytes
total */
int  a;          /* 4 bytes */
int  c;          /* 4 bytes */
char b;          /* 1 byte  */
char d;          /* 1 byte  */
char pad[2]; /* 2 bytes */
};
```

```
struct sa{
    int a;
    char b;
    int c;
    char d;
};
```

sizeof(): 16

```
struct sb{
    int a;
    int c;
    char b;
    char d;
};
```

sizeof(): 12

Further Reading:
http://www.catb.org/esr/structure-packing/

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Summary

- Profiling can be used tell us where are programs spend time
- Time critical sections are candidates for optimisation
- Optimisations can be used to improve both compute and memory bound applications
- Most obvious optimisation technique is to try another algorithm
- The msvc compiler performs many optimisations but careful coding can help it
- Always try and have good locality of memory accesses to improve cache usage
- Optimisation requires lots of trial and error!