Cost Modelling for Parallel Programming

Stuart Monro
monros@dcs.gla.ac.uk

Motivation

High Level Techniques for Parallel Programming

with a focus on

Programming models and cost models for data parallelism

EU Funded Collaboration

- University of Bayreuth
- Chemnitz University of Technology

Visits:

- Gudula Rünger & Thomas Rauber - Glasgow 2010
- John O'Donnell – Bayreuth 2010
- Stuart Monro – Chemnitz 2011
- Jörg Dümmler – Glasgow 2012 (March)

Motivation

- Sequential algorithms can be sped up by running them on faster machines
  - The whole algorithm will speed up
- Parallel algorithms on parallel architectures benefit from more tuning parameters than just overall speed of the machine
  - Memory usage
  - Thread management
  - Data management
- There are also limitations to the amount of parallelism that can be introduced to an algorithm
  - Amdahl's Law
  - Inherent parallelism
- Restructuring a parallel algorithm in order to tune it is not straightforward. The fundamental aim of this research is to help programmers perform this restructuring exercise.

Motivation

- Extensive research has produced a large body of information regarding parallel programming, algorithms and architectures
- Less clear is how to integrate these – which algorithms suit which architectures
- The investment of programming effort for any combination of algorithm and architecture is very high. So it is important to make the correct decision at the outset
- That decision needs to be based on more than just data/task shorthand and intuition
- Aim is to provide a more concrete foundation to support the decision making process

Exploring the Design Space – An Analogy

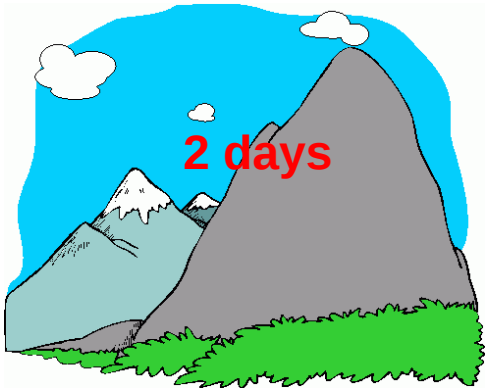
Programmer

Finished Program

Exploring the Design Space – An Analogy

Programmer

Cost Model



Finished Program

Exploring the Design Space

- Restructuring a parallel algorithm in order to tune it is not straightforward.

There is a large design space
- Multicores – task & data parallelism
- FPGAs – task & data parallelism
- CPU clusters – task & data
- GPUs – data (& task?)

- Too expensive in terms of time & effort to code these up to compare them, need a cost model to provide a comparison on which to base a decision

The cost model can also help in terms of considering development time e.g.:
- CPU cluster will result in 15% speed up & will require 100 hours of development time
- GPU will result in 25% speed up & will require 500 hours of development time

If a 15% performance gain is sufficient for the programmer's needs then CPU cluster may be best solution

The cost model informs the decision

Cost Models

- A cost model expresses the execution time, memory consumption or power consumption of an algorithm as a function of relevant parameters
- It can be used to predict the performance of an algorithm based on those parameters
- For example consider the following:

```
int sourceArray[i];
int resArray[i];
int j = 42;

for (k = 0; k < i; k++)
    if sourceArray[k] < j
        resArray[k] = 1;
```

- The performance of a parallel algorithm to carry out this task could potentially depend on the following parameters:
  - Number of comparisons
  - Number of threads
  - Number and type of memory accesses
  - Cache behaviour

Cost Models

- Assuming off-chip memory is used, the cost model could look like the following:

$$f(x) = \frac{logx((Ax+B)+\frac{x}{y}+(Cx))}{D}$$

Where:

x = number of operations
y = number of threads
A & B = constants relating to the number of operations
C = constant relating to the number of off-chip memory accesses
D = constant relating to caching behaviour

f(x) = gives the predicted performance time in cycles

- These constants are known and will vary depending on the architecture used and the structure of the algorithm.

Cost Models

- Cost models can assist both in:
  - Choosing the architecture
  - Structuring the algorithm.
- They can be used by the programmer to make key decisions based on empirical data.
- There is a trade off between the usability and accuracy of the cost model:
  - The more accurate the model, the more complex it will be to use

A cost model does not need to be 100% accurate to be useful

Cost Models

- The cost model performance prediction may vary from the actual performance
- A percentage measure of accuracy is calculated as:

$$100(\frac{A-P}{A})$$

Where:
A = actual performance
P = predicted performance.

For example:

Actual execution time (in cycles) = 105,000

Predicted execution time = 100,000
Accuracy = 4.76%
Predicted execution time = 110,000
Accuracy = -4.76%

- Work at Bayreuth and Chemnitz has concluded that a model which produces an accuracy of the actual time plus or minus 10% is useful

Using a Cost Model in Parallel Programming

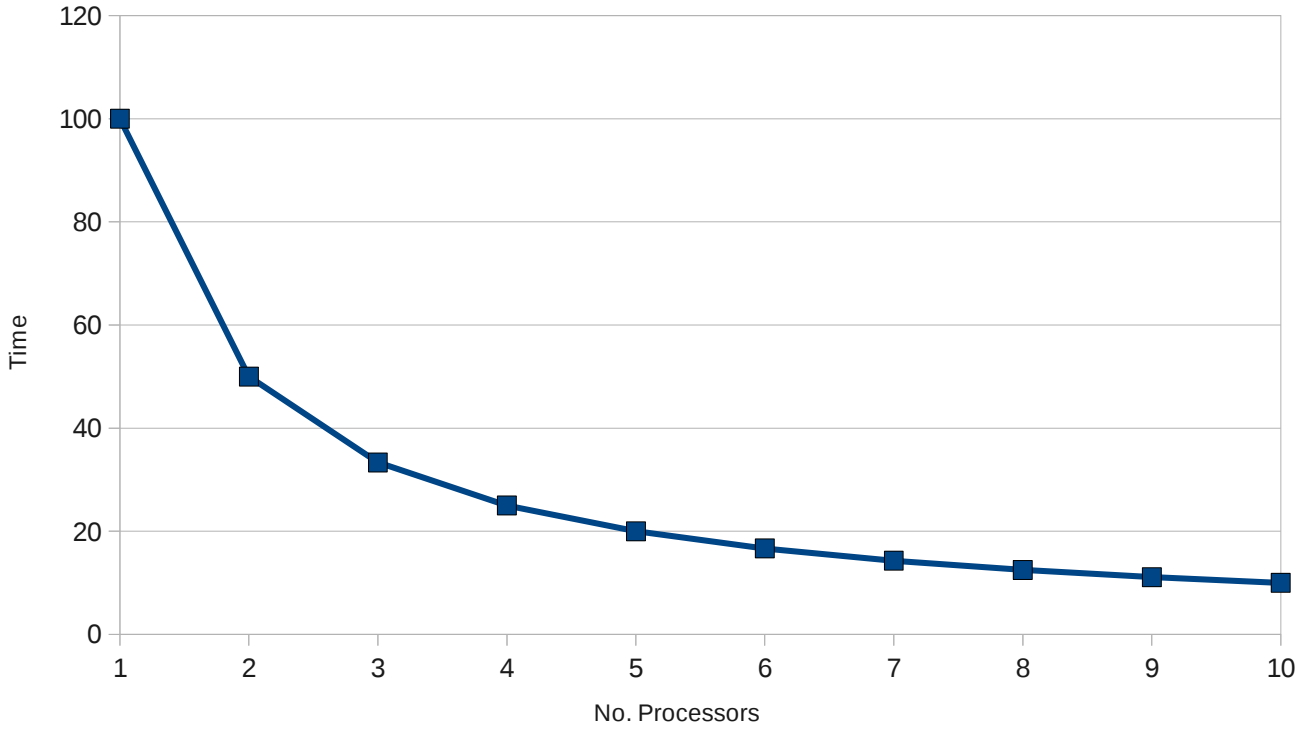
- Parallel programming encompasses a wide range of architectures, from single nodes with a few cores to supercomputers
- The construction & implementation of a supercomputer can cost $100millions
- This makes it important to manage the runtime of applications on such a platform
- A parallel application running in an environment such as a multi-node CPU cluster or a supercomputer will consist of many **tasks**.
- In this context a task is a piece of work which can run on one or more processors
  - A task is **not a thread** but may use a collection of threads
  - Task examples:
    - Execute a specific algorithm (sum reduction of an array)
    - Update a display
    - Gather information from remote sensors

Using a Cost Model in Parallel Programming

- A function will exist relating to the task which can be used to calculate how long the task will take to complete dependant on how many processors are allocated to it e.g.

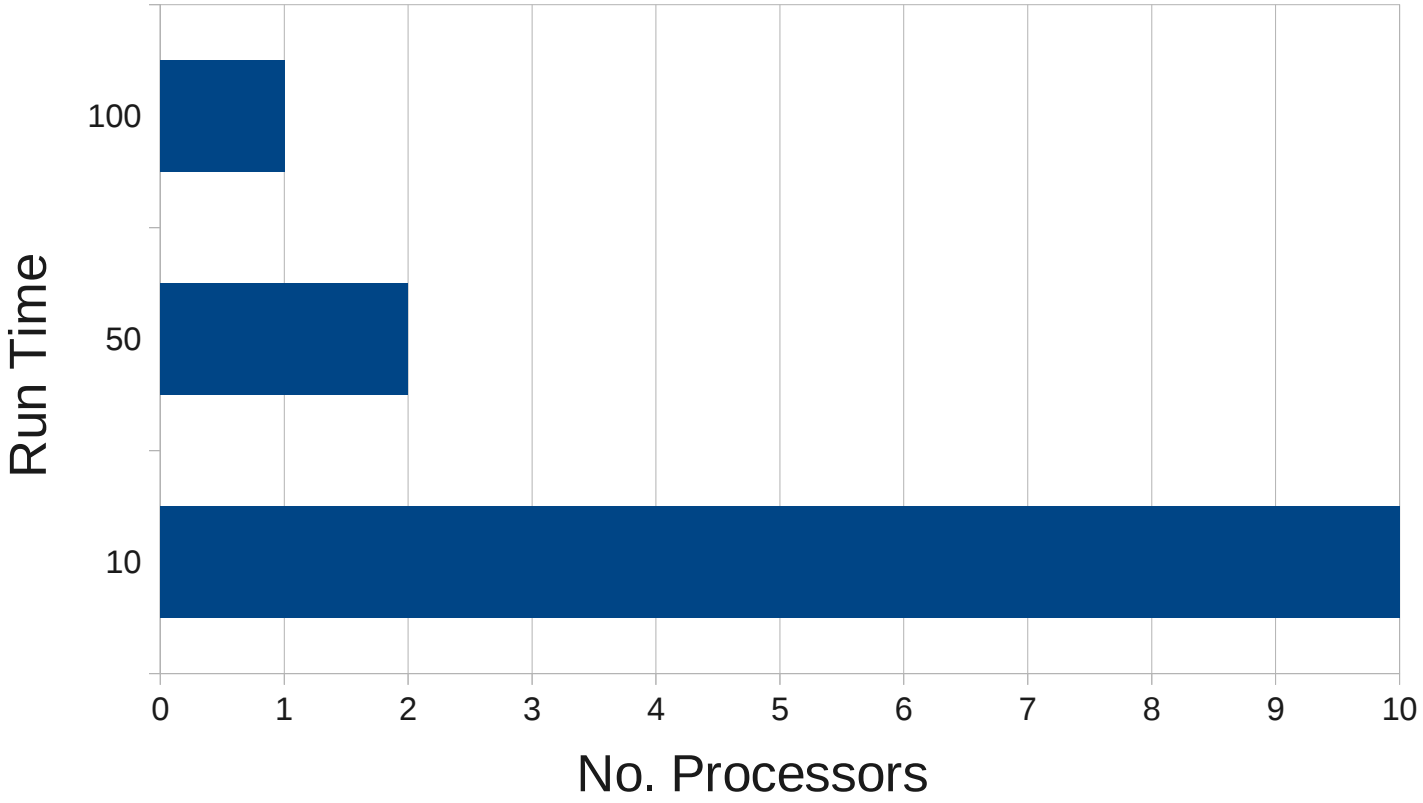
$$f(x) = \frac{A}{x}$$

- In this example
  - A=100
  - x = the number of processors allocated



Using a Cost Model in Parallel Programming

- Using this function we can effectively alter the running time of a task by amending the number of processors allocated to it:

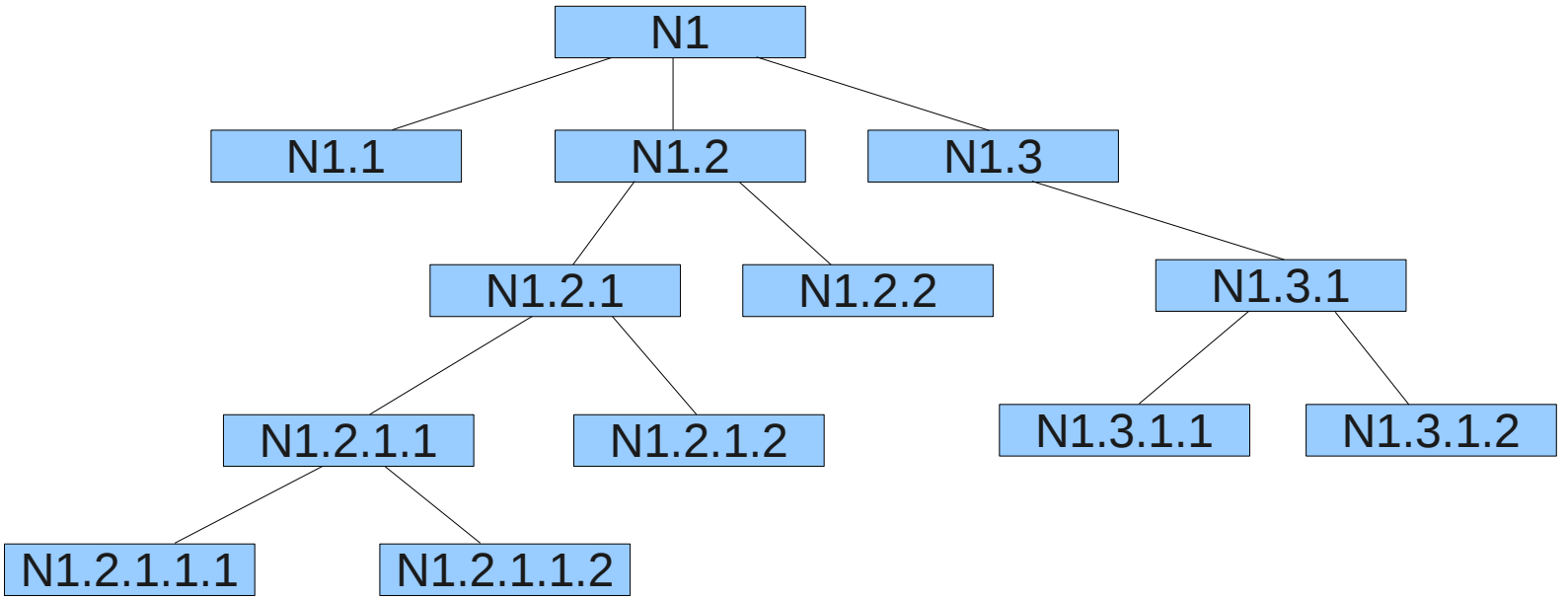


Using a Cost Model in Parallel Programming

- The parallel application can be represented as a tree

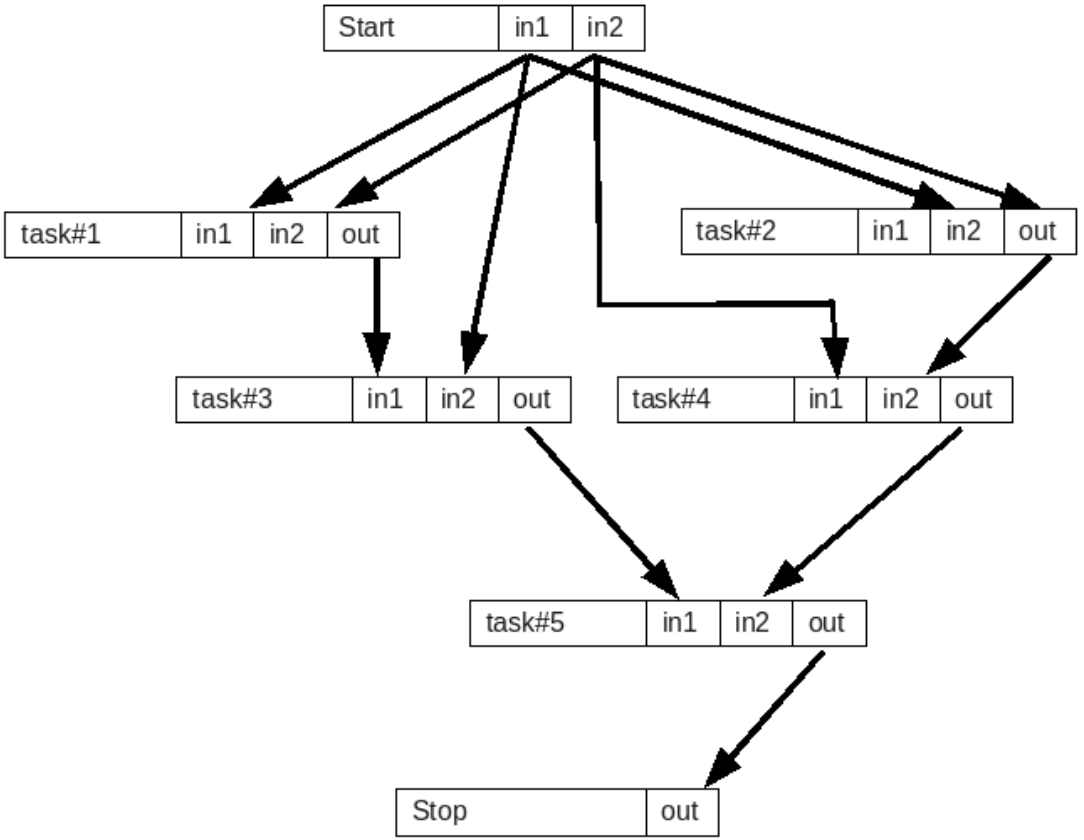
- Each node in the tree will have processors available to it which will allow the execution of specific tasks

- A key aspect of structuring a parallel application will include considering the partition of such tasks, with attention being paid to:
  - Data dependencies
  - Communication costs
  - Processing requirements



Using a Cost Model in Parallel Programming

One node may require the execution of a number of tasks which have their own dependencies



Using a Cost Model in Parallel Programming

If a cost model is available for each task then it should be possible to predict how long each will take to run

Task Number	Time Units Required
1	40
2	60
3	75
4	100
5	100

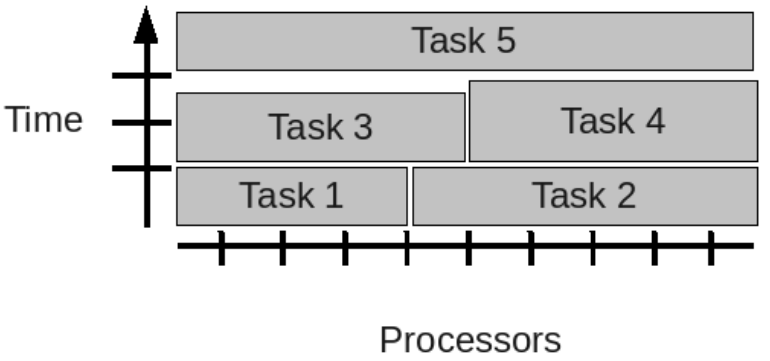
These predictions are based on the assumption that one processor is used for 1 task.

Using a Cost Model in Parallel Programming

If these cost models include parameters for the number of processors each task uses, then the run times can be optimised to the minimum possible length overall for the application by varying the number of processors available to each task.

The example below works on the basis that 10 processors are available in total

Task Number	Time Units Required
1	40
2	60
3	75
4	100
5	100
Total Time Units Required	375



Total Time Taken = 40 Units

This is the approach used by Dümmler et al in their development of a scheduler for multiprocessor task programming

Parallel Programming – Real World Examples

These examples were discussed at the 3rd UK GPU Computing Conference in December 2011. In both cases, efforts made to accelerate existing programs using GPUS were presented

Porting Fortran Oceanographic code to GPUs

- Science & Technology Facilities Council (part of Daresbury Science & Innovation)
- Attempt to accelerate a European ocean model originally written in Fortran by use of GPUS.
- Each module ported to GPU grew in size from ~400 lines of code to ~1000 lines of code
- Total project took approx 6 months
- Four routines in total were ported to GPUs. Of those, speed-ups were achieved on three routines. Maximum speed-up was around 25%
- Conclusion of the project participants was that the gains in performance did not justify the effort expended

Cost models were not used in this exercise

Parallel Programming – Real World Examples

These examples were discussed at the 3rd UK GPU Computing Conference in December 2011. In both cases, efforts made to accelerate existing programs using GPUS were presented

BarraCUDA – Fast Sequence Mapping Software using GPUs

- University of Cambridge Metabolic Research Laboratories
- Attempts to accelerate genome sequencing by developing software targeting many-core architectures
- Porting an existing algorithm to GPUs – no information regarding development time
- First attempt failed to produce performance gains
- Second attempt saw partial success in that a 10% speed-up over the performance of an 8 core CPU was achieved (not a like-for-like comparison)
- Research team expected better than this – a 10% speed up was not considered satisfactory based on the effort involved

Cost models were not used in this exercise

Development of Cost Models

Two approaches to the development of a cost model:

- Theoretical development
- Measurement based development

In either approach the aim is to produce a function to predict the execution time of the algorithm

Development of Cost Models

Theoretical development is based on an analysis of the characteristics of the algorithm and the hardware that it is intended to be run on. For example, the manufacturer of the hardware may produce documentation detailing how long certain operations should take on their hardware e.g.:

- Accessing global memory will take between 400-600 cycles
- Accessing shared memory will take 4 cycles
- Adding two integers will take 4 cycles
- Multiplying two integers will take 16 cycles

The work done by Hong & Kim and Kothapalli et al takes this approach.

Development of Cost Models

Measurement based development focuses on developing benchmarks for standard application operations and measuring how long these take to run. Such operations could include:

- Addition of two integers/floats
- Multiplication of two integers/floats
- Comparing two integers/floats
- Read from memory
- Write to memory
- Transferring data between devices

```
int z, x = 42, y = 24;
start_time = clock()
loop n times
    z += x + y
stop_time = clock()
duration = stop_time - start_time
```

Once the measurements are captured, models are then derived using curve fitting

Why Use a Measurement Based Approach

A number of factors can affect the performance of an algorithm, not all of which can be captured by a theoretical model:

- OS operations
- Code start up times
- Data transfer overheads
- Memory contention/latency
- Task scheduling mechanisms
- Architectural differences
- Unknowns

Some theoretical models (Hong & Kim) attempt to factor in some of these (memory latency)

Running & measuring benchmarks gives a clear view of likely performance

The theoretical model bases its predictions on what should happen. The measurement based model bases its predictions on what is actually observed to happen

Developing a Measurement Based Model

Measurements will take into account unknown factors. A model based on measurements would be developed iteratively starting with:

$$f(x) = 4x + 100x + K$$

K = unknown factors

As a new factor is identified and measured this could be added to the model.

For example, if it was identified that the cost of memory contention is 10 cycles for every 1000 accesses then the model could be expanded to:

$$f(x) = 4x + 100x + \frac{x}{100} + K$$

K = remaining unknown factors.

As the model is refined, the impact of K on it will be reduced

The Development Approach

The approach taken to develop models will use measurement experiments as follows:

- 4 devices/architectures
  - Drygalski
  - Kryten
  - Curieuse 1
  - Curieuse 2
- Measuring:
  - Operations
  - Memory hierarchy
  - Standard functions
- Unit of measurement - cycles
- Results analysis
- Curve fitting

Ensuring Benchmarks Are Effective and Valid

First benchmark – adding two integers

```
int z, x = 42, y = 24, n = 1000;
start_time = clock()
for (i = 0; i < n; i++)
    z += x + y
stop_time = clock()
duration = stop_time - start_time
```

Problem – regardless of value of *n*, duration remains the same!

Don Knuth - Analysis of assembly code allows us to see what is actually going on rather than what we believe to be happening

Assembly listing showed that:

- Compiler performing addition operation
- At runtime, result of that addition was multiplied by n

Runtime measurement gives us the time to do that multiplication – nothing more

Ensuring Benchmarks Are Effective and Valid

CUDA compiler produces intermediate assembly code listing in PTX format

- PTX is described by Nvidia as a low-level Parallel Thread eXecution virtual machine and information set architecture.

- Mainly used for optimising kernels or developing cross-platform libraries

- Can also be used to abstract away from underlying GPU architecture

- In this case, used to analyse the effects of compiler optimisations on benchmark code

(Full PTX listing not shown here due to its verbosity)

Ensuring Benchmarks Are Effective and Valid

Analysis of the compiled benchmark code at the assembler level ensures that the benchmark is doing what is intended

Initial benchmarks were affected by compiler optimisations resulting in invalid results:

```
$LDWbegin__Z12runBenchmarkPjPiii:

    ...

    ld.param.s32   %r3, [__cudaparm__Z12runBenchmarkPjPiii_numKernelLoops];
    mul.lo.s32     %r5, %r3, 66;

    ...
```

Revised code produced the following:

```
$LDWbegin__Z12runBenchmarkPjPiii:

    ...

    @%p1 bra  $Lt_0_3330;
    ld.param.s32   %r3, [__cudaparm__Z12runBenchmarkPmPiiiii_numKernelLoops];
    mov.s32   %r5, %r3;
    ld.param.s32   %r6, [__cudaparm__Z12runBenchmarkPmPiiiii_firstInt];
    ld.param.s32   %r7, [__cudaparm__Z12runBenchmarkPmPiiiii_secondInt];
    add.s32   %r8, %r6, %r7;

    ...
```

The Experimental Approach – Curve Fitting

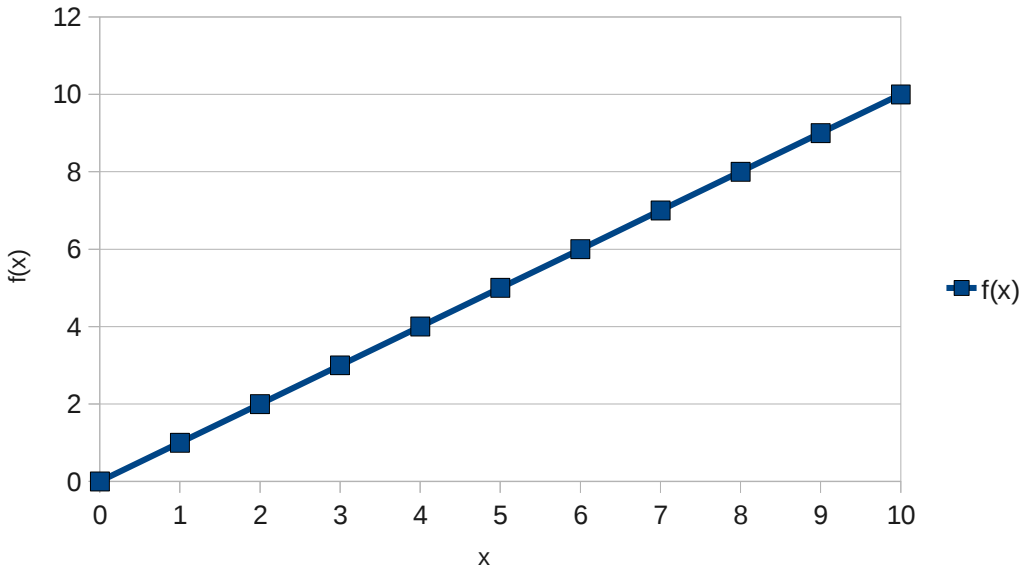
The development of a mathematical model (describing a curve) which best fits a series of data points

Matlab provides functionality to produce such models

That model then forms the basis of the cost model for the operation which originally produced the data points

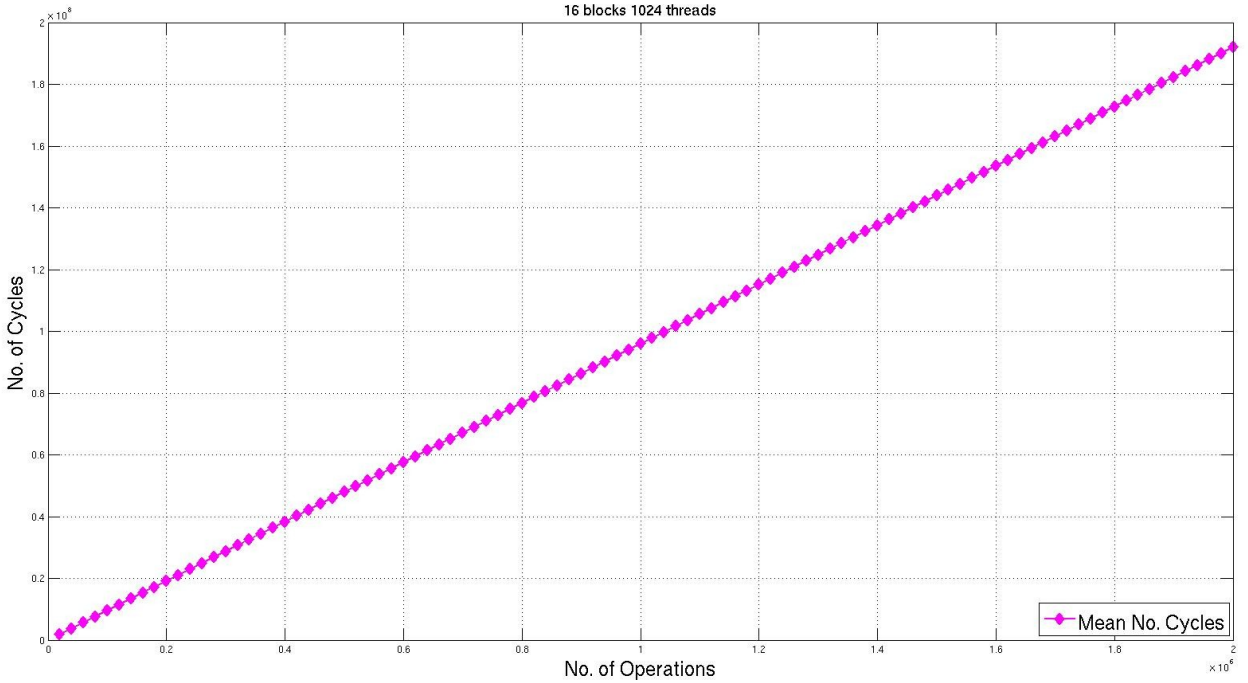
A (very simple) example:

The function f(x) = x will produce a straight line



The Experimental Approach – Curve Fitting

The graph produced by the addition of two integers:



The Experimental Approach – Curve Fitting

A linear model to describe that graph (seen on previous slide):

$$f(x) = Ax + B$$

Where

A = 42
B = 215

If a 95% confidence interval is used then the upper and lower bounds of A and B can also be specified:

A (41.5, 42)
B (210, 217)

Methodology

The methodology used for each experiment is as follows:

- Benchmark code written in C/CUDA to perform specific operation
  - Variation in each benchmark is the number of loops (10,000 – 1,000,000 in increments of 10,000)
  - Each variation of each benchmark is run 10 times
- Results are analysed using Matlab and a model for each benchmark is developed
- Model is tested with a utility written in Java to apply the model to every benchmark variation and compare the recorded actual result with the result predicted by the model
- Each experiment is run overnight between midnight & 4am (to avoid GPU use conflicts)
- Each experiment is run at least 5 times
- To date over 150 experiments have been run on three different architectures

Results to Date

- Models have been developed for six different operations on two different architectures.

- This development has included testing those models to ensure that they meet the required accuracy levels

- Some finalisation is required to ensure that differences in results between experiments are supported

- Further experiments have been developed to handle different input parameters to those operations (varying the number of threads) these are expected to be run and analysed in the next 1-2 weeks

- Work has begun on development of experiments to produce models for different aspects of the memory hierarchy

Next Steps

- Investigate impact of caching/kernel optimisation as part of memory hierarchy modelling
- Produce models for higher level functions (map, reduction, scan)
- Incorporation into TwoL (in conjunction with Chemnitz & Bayreuth)
- Expansion into more complex algorithms/applications
- Publish