Imperial College London
Department Of Computing

Runtime Compilation with NVIDIA Cuda as a Programming Tool

3rd Year JMC BSc
Final Year Project Report

Tristan Perryman

Supervisor: Paul Kelly

Second Marker: Tony Field

Abstract

In recent years graphics processing power has been increasing and today's graphics cards offer many processors on one card with performance potentially equalling that of hundreds of processors by utilising a SIMD architecture for each processor. As a result of this a relatively new and very active area of research has emerged – that of General Purpose Graphics Processing Units (GP-GPU). Recently major graphics card vendors have started to produce GP-GPU graphics cards. In particular, NVidia have launched their new Cuda C extension language for programming their range of graphics cards. We describe an extension to the TaskGraph C++ library (which is designed for type-safe runtime compilation) that adds support for runtime compilation of the NVidia Cuda language. We also describe some generic extensions to the TaskGraph library which are useful even for TaskGraph users who do not wish to use Cuda such as pointers and recursion. Finally, we demonstrate through the creation of four main examples, the benefits that this extension provides. Of particular interest, we demonstrate a specialised separable convolution filter which can obtain over a 300% speed increase over a similar GPU-based separable convolution filter under certain circumstances and we also demonstrate a specialised ray tracer which can obtain up to a 30% speed increase over a similar GPU-based ray tracer under certain circumstances.

Acknowledgements

I would like to thank the following people for their help throughout the duration of my project:

- Paul Kelly, my supervisor, for his continual guidance, ideas, expert knowledge and assistance with my TaskGraph questions.
- Michael Mellor for helping me to fix a nasty TaskGraph structures bug.
- Francis Russell for helping me with my TaskGraph build problems.
- Lee Howes for his expert knowledge on Cuda and for helping to set up the computer for my demonstration.
- Jay Cornwall for his advice and expert knowledge on Cuda.

Contents

1 Introduction

In this report we discuss our extension to the TaskGraph C++ metaprogramming library which adds support for runtime compilation of the NVidia Cuda language. This makes it possible to runtime compile fast code for NVidia's latest GPUs utilising metaprogramming techniques. Furthermore, some additional generic extensions which are not specific to Cuda, but are often necessary or useful when writing TaskGraph Cuda programs, are also discussed. Finally, the extension is evaluated with the aid of several examples that are designed to demonstrate the benefits, particularly performance benefits, from runtime specialisation.

This section begins the report by discussing the motivation, the original goals, and the final achievements.

1.1 Motivation

For a long time ways of optimising software to run as fast as possible and writing programming languages that are easy to use, yet which are still powerful and efficient has been a major area of computer science. On the other hand, optimising and designing hardware to be as fast as possible has also been central to getting computers to where they are today.

Recently, however, there has been a huge shift towards massively parallel architectures. On the one hand there is the talk of mainstream CPUs from the likes of Intel with 80 cores [35], on the other, companies such as NVidia already sell GPUs, which in applications well suited to *stream processing* (i.e. algorithms which can be adapted to be efficient in SIMD architectures) have the equivalent processing power of 128 processing units [18]. One of the advantages of GPUs over CPUs is their efficiency at processing certain types of algorithms, in particular, algorithms with high arithmetic intensity (so as to ensure memory latency is reduced) which involve a lot of floating-point vector maths operations but with a minimal amount of branching (so as to ensure that SIMD parallelism is maintained) [8].

As a result of these recent developments an ever expanding area of computer science is that of investigating ways of harnessing the full potential of GP-GPUs (General Purpose Graphics Processing Units), which allow the programmer to write their own code which executes on the GPU. Thus, our investigation was motivated by a desire to investigate possible benefits of runtime compilation on a GP-GPU. In particular, NVidia have recently released their Cuda C extension language and thus, while some research has been conducted on GP-GPU shader languages (see section 2) very little research has been done with relation to this language. We are therefore here to fill a void by investigating the performance benefits and ease of coding benefits that runtime compilation with NVidia Cuda provides.

1.2 Goals

The intention of this project is to investigate methods of runtime compilation for NVidia's new Cuda C extension language. In particular, this project has the following aims:

- **Demonstrate performance benefits to Cuda through runtime specialisation** (such as via loop transformations)
- **Demonstrate that it is possible to add support for all of Cuda's low-level features** (which are essential for optimisation) whilst maintaining TaskGraph's strong type-checking capabilities.
- **Demonstrate a single source model** whereby both Cuda and host C++ code can coexist in the same source file.
- **Allow the TaskGraph user to create clean, legible, Cuda-style code** without additional complexity resulting from TaskGraph being a C++ library rather than a written-from-scratch programming language.

1.3 Achievements

TaskGraph has been successfully extended to add support for the majority of the NVidia Cuda C-extension language. Also added are numerous “generic” extensions and architectural changes to TaskGraph, which, while not specific to TaskGraph's Cuda support were implemented because they were useful for either neatness or speed, or essential to allow Cuda to operate correctly. Finally, some examples are provided which demonstrate the performance gains, type-checking, and code legibility that the library extensions provide.

1.3.1 Key Achievements

- **TaskGraph Cuda extension** to allow NVidia Cuda code to be runtime compiled with some syntax and type safety properties.
- **Generic TaskGraph extensions** that are not specific to Cuda but which are sometimes necessary for Cuda support and which help to improve TaskGraph's capabilities and allow for neater code in non-Cuda examples.
- **Four main examples** intended to show off various benefits provided by the extension, including three Cuda examples (matrix multiplication, separable convolution filter and a ray tracer) and one non-Cuda example (expressions interpreter). Of particular interest, the separable convolution filter is capable of giving performance gains of over 300% in some cases. Similarly, the ray tracer is capable of giving performance gains of over 30% in some cases.

Below, a more detailed listing of the individual changes that have been made to the TaskGraph library is given:

1.3.2 Generic Extensions

- **Pointer support -** TaskGraph now has support for handling pointers, it can both take references of variables and store them in a pointer or dereference pointers to obtain access to the underlying data. It also allows pointers to be accessed as though they were an array (as in normal C). Furthermore, conveniently, the TaskGraph syntax for handling pointers and references has been designed to be exactly the same as that for normal C.
- **Improved function call support** - TaskGraph already had some support for calling functions, however, it was necessary for the user to explicitly declare the function to be called as a variable before it could be used, it was also necessary to explicitly specify the number of parameters as a numeric value and also specify the types of the parameters, which was not very convenient. In our improved version the user simply writes tCall(char* funcname, params...), where funcname is the name of the function and params are the parameters passed to the function.
- **Syntactically safe method of function calling** – While all calls to external functions must occur by passing a string of the function name to a tCall function, it is, however, possible to call other TaskGraphs by simply passing the TaskGraph through to the tCall function, hence, avoiding 'dodgy' string based function calls.
- **Function call dependency resolving** - When a call is made to another TaskGraph from within a TaskGraph then all its dependent TaskGraphs are compiled in with the dependent TaskGraph at compile time.
- **Recursion** - It is even possible for a TaskGraph to call itself recursively
- **Support for adding include files into a TaskGraph -** This is important for calling external library functions without the programmer having to explicitly declare them to the TaskGraph code.
- **Parameter passing changes** - TaskGraph now passes parameters in a way more akin to normal C

than before. Previously, if a TaskGraph function is called with e.g. T(a) then a reference to 'a' would be passed rather than 'a' itself. The improved behaviour would now pass the actual contents of 'a'. It is, however, still possible to pass by reference by writing, e.g. T(&a), in which case the TaskGraph declaration would have to be told to expect a pointer variable.

- **Compiler options passing** - Allows the TaskGraph user to pass options to the compiler

- **Indeterministic control flow instructions** - These are essentially all the standard TaskGraph control flow instructions (e.g. tFor, tIf, tWhile etc) but they accept an extra parameter, which means that the control flow instruction only appears in the code if the parameter evaluates to true, otherwise the code contained inside the control flow instruction will still appear in the code but the control flow instruction itself will not. This could allow for more flexible meta programming in some cases.

1.3.3 Cuda Specific Extensions

- **Support for invoking the NVidia Cuda Compiler on a TaskGraph**

- **Support for Cuda Global Calls** - These are function calls to the Cuda kernel that use a tCudaGlobalCall function that works in a similar way to tCall (see above) except it takes additional arguments that would normally appear using angle brackets if writing in Cuda natively.

- **Cuda built-in structure support -** Cuda uses some 37 built-in vector structure types with 1,2,3 or 4 components and with different component types (int, uint, float etc) hence each of these needed to be implemented in TaskGraph.

- **Cuda built-in variable support** - Namely: gridDim, blockIdx, blockDim and threadIdx which are used by the Cuda kernel to determine information about its "execution environment".

- **Support for special variable type qualifiers -** Namely, __shared__ and __constant__

- **Ability to declare a TaskGraph as a cuda kernel TaskGraph** – This is needed since Cuda kernels must specify a __global__ directive in their declaration.

- **Single source model** - Both Cuda and host C++ code can now coexist in the same source file

1.3.4 Examples

Generic Examples

- **TaskGraph expression interpreter example** - This is a non-Cuda example of how our new and improved function calling mechanisms can help to improve non-Cuda TaskGraph code.

Cuda Examples

- **TaskGraph Cuda matrix multiplication example -** This was mainly written as a test example to show the TaskGraph extension working correctly with a real example, thus, it does not show any particular advantages over normal Cuda but is probably the simplest TaskGraph Cuda example that we have built.

- **TaskGraph Cuda separable convolution filter example** - This specialises at runtime by unrolling the loops to different filter sizes, this gives a noticeable performance gain over using normal loops, it can even specialise to specific filters giving an even higher performance gain in some cases of over 300%.

- **TaskGraph Cuda ray tracing example** - This is a real-time TaskGraph Cuda ray tracer which offers performance gains over a standard Cuda ray tracer by specialising to the objects in a scene, it can also specialise to both the eye and objects in the scene, for an additional performance gain of up to around 30%.

1.4 Very Simple TaskGraph Cuda example

As a brief introduction to get a taster of the work that has been carried out here, a very simple TaskGraph Cuda example which does vector scalar multiplication on some 1024 floating point 3D vectors is demonstrated below. The example also specialises to the scalar which the vectors are multiplied by. It does this by replacing the variable scalar value by a constant in the generated code. The reader is not expected to thoroughly understand the example at this stage, however it should provide some insight into the direction that this report is heading.

```
#include <stdlib.h>
#include <TaskGraph>
#include "TaskUserFunctions.h"
#include <boost/preprocessor.hpp>

using namespace tg;

#include <TaskCudaTypes.h>

struct vector3
{
  float x;
  float y;
  float z;
};

#define VECTOR3FIELDS ((float,x))((float,y))((float,z))
TASK_STRUCT(tgVector3,vector3,VECTOR3FIELDS)
#undef VECTOR3FIELDS

typedef TaskGraph<void, vector3*> cuda_host;
typedef TaskGraph<void, vector3*> cuda_kernel;

#define NUM_VECTORS 1024
#define BLOCK_SIZE 64

int main( int argc, char *argv[] ) {
  cuda_host T;
  cuda_kernel T2;
  int k=5;
  int mem_size = sizeof(vector3)*NUM_VECTORS;
  cudakerneltaskgraph(cuda_kernel, T2, tuple1(d_A))
  {
    tVar(int, bx);
    tVar(int, tx);

    bx = blockIdx.x;     // Block index
    tx = threadIdx.x;    // Thread index
    tVar(int, thisvector);
    thisvector = tx + bx*BLOCK_SIZE;

    d_A[thisvector].x = d_A[thisvector].x*k;
    d_A[thisvector].y = d_A[thisvector].y*k;
    d_A[thisvector].z = d_A[thisvector].z*k;
  }

  cudahosttaskgraph( cuda_host, T, tuple1(h_A) ) {

    tInclude("stdio.h");
    tInclude("cutil.h");
```

```
    tCall("CUT_DEVICE_INIT");

    // allocate device memory
    tVar(vector3*, d_A);
    tCall("CUDA_SAFE_CALL", tCall("cudaMalloc", cast<void**>(&d_A), mem_size));
    // copy host memory to device
    tCall("CUDA_SAFE_CALL", tCall("cudaMemcpy", d_A, h_A, mem_size, cudaMemcpyHostToDevice));


    // setup execution parameters
    tVar(dim3, threads);
    threads.x = BLOCK_SIZE;

    tVar(dim3, grid);
    grid.x = NUM_VECTORS / threads.x;

    tCudaGlobalCall(T2, grid,threads,0,d_A);

    // copy result from device to host
    tCall("CUDA_SAFE_CALL", tCall("cudaMemcpy", h_A, d_A, sizeof(float3)*NUM_VECTORS,
cudaMemcpyDeviceToHost));
    tCall("CUDA_SAFE_CALL", tCall("cudaFree", d_A));
   }

   const char* args[] = {"-Xcompiler", "-fPIC", "-I", "/usr/local/cuda/include", "-I",
"/home/tp105/NVIDIA_CUDA_SDK/common/inc/", "-lGL", "-lGLU", "-lcutil", NULL};

   T.compile(tg::NVCC, true, args);

   // set seed for rand()
   srand(2006);

   // Allocate memory
   vector3* h_A;
   h_A = (vector3*)malloc(mem_size);

   // Initialize array of vectors to some random values
   int i;
   for(i=0; i < NUM_VECTORS; i++)
   {
     h_A[i].x = rand() / (float)RAND_MAX;
     h_A[i].y = rand() / (float)RAND_MAX;
     h_A[i].z = rand() / (float)RAND_MAX;
   }
   printf("BEFORE: %f ", h_A[2].y);
   T(h_A);
   printf("AFTER: %f ", h_A[2].y);
}
```

The basic principle of this example is that there are two TaskGraphs (i.e. functions which are generated at runtime). One is a Cuda kernel (which runs on the GPU) and the other is the Cuda host code (which runs on the CPU and performs various setup functions and calls the kernel). Inside these functions, code which is similar but not identical to standard Cuda code is used in order to ensure that the code which will be generated at runtime is distinguished from code which is generated at compile time.

The two TaskGraphs are generated in main(), after which they are compiled with the T.compile() call. A random set of 1024 vectors is then generated and passed to the Cuda host code via the call "T(h_A);". The Cuda code which was just compiled is then executed and it multiplies all the vectors by 5 (since k = 5) and overwrites the original values with the new set of values.

1.5 Report Structure

Background
This section will discuss relevant previous work done on the topic of runtime compilation, GPUs and massively parallel architectures. It will also discuss some background directly relevant to understanding this report.

Design of the TaskGraph extension
This section will discuss the design of the TaskGraph extension, including how the various features function from the users point of view along with some insight as to why they were designed that way. It will not, however, cover a great deal of technical detail as the already existing TaskGraph implementation constrained many of the low level design details.

Implementation of the TaskGraph extension
This section will discuss the challenges of extending the TaskGraph library, we will cover the various challenges of modifying the library to add Cuda support including some of its architectural limitations and how they were worked around.

Evaluation
This section will evaluate our TaskGraph extension by implementing several examples for TaskGraph Cuda and evaluating the benefits which the extension offers these examples. Any issues with the overall implementation of the extension are also discussed.

Conclusion
This section will summarise the work done and strive to provide a balanced evaluation of our overall achievements and how closely they fit the original project goals. We will also discuss any further work that could be carried out to improve upon the work discussed in this report.

2 Background

This section discusses related works which have been done, in particular with regards to runtime compilation, parallelism and GP-GPU programming and optimisation.

2.1 Metaprogramming Tools

This section discusses the various metaprogramming libraries and languages, or more specifically, programs which are capable of modifying themselves at runtime and essentially recompiling on-the-fly.

2.1.1 TaskGraph

TaskGraph is a C++ library which allows runtime generation and compilation of C code that can then be manipulated at runtime using meta-programming techniques, such as loop transformations (unrolling, fusion etc). TaskGraph essentially forms a mini C sub-language that works by utilising various C++ features, such as operator overloading, that allows for fairly strong syntax and type-checking capabilities without needing to modify the C++ compiler. [1]

Since this report is strongly based around TaskGraph, a simple TaskGraph example will be very briefly discussed and any key relevant parts of the library explained as the discussion progresses. However, while attempts are made throughout the report to explain any relevant details of the TaskGraph library as needed, we strongly recommend that the interested reader reads the original TaskGraph documentation which is contained both with our submitted sourcecode and the original sourcecode on the TaskGraph website [33].

Observe the following sourcecode, which is taken from the addc example included in the TaskGraph distribution, but with additional comments added in order to thoroughly explain how it works:

```
/*
 * Simple taskgraph example which shows how to create
 * and execute a taskgraph.
*/
#include <stdio.h>
#include <stdlib.h>
#include <TaskGraph>

using namespace tg;

// Define a TaskGraph type that will return an int (first template parameter) and takes as a parameter an int (second
template parameter)
typedef TaskGraph<int, int> addc_TaskGraph;

int main( int argc, char *argv[] ) {

// Instantiate the TaskGraph object of our specified type
        addc_TaskGraph T;
        int b = 1;
        int c = 5;

// Begin generating TaskGraph code of type addc_TaskGraph storing it in the TaskGraph T taking 1 parameter which we
call "a"
        taskgraph( addc_TaskGraph, T, tuple1(a) ) {

// This line will create a printf("a: %d", a); in the generated code
          tPrintf("a: %d", a);
```

```
// This line will essentially create a tReturn(a + 5); statement in the generated code, with 'a' being the parameter passed
to the TaskGraph and 'c' is a constant that will be 5 in the generated code since this is not a TaskGraph variable, but a
normal C++ variable, thus it will take the value that it has at generation time which is 5.
                tReturn(a + c);
        }

// This compiles the TaskGraph object T that we have just created with the GCC compiler
        T.compile( tg::GCC, true );

// This line executes the TaskGraph passing the parameter b and prints the result
        printf( "T(b) = %d where b = %d\n", T(b), b );
}
```

One final point worth noting is that prior to the start of this project, TaskGraph also had very basic support for parallelism using the Cell architecture, which allows the user to construct separate chunks of C code at run-time and then execute each chunk in parallel as a separate thread.

2.1.2 Stanford University Intermediate Format (SUIF) Compiler

SUIF is a research compiler project built by the University of Stanford. It provides a common intermediate format for representing programs which can then be used to do a variety of compilation passes. SUIF has built-in support for a variety of such passes, including dependence analysis and a loop-level parallelism optimiser. It also allows direct access to its intermediate representation thus allowing additional optimisers to be developed for SUIF [17]. TaskGraph uses this as its backend for generating and optimising code.

2.1.3 MetaOCaml

MetaOCaml [2] is a multi-stage programming language based on OCaml. Similar to TaskGraph it supports run-time code generation and specialisation. Unlike TaskGraph, however, it is a functional programming language whereas TaskGraph is a C++ library. One of the advantages which it offers to TaskGraph is that it provides an extensive type safety system [3].

2.1.4 Sh

See section 2.3.5

2.2 CPU Parallelism Tools

This section discusses the various systems that are available for writing parallel code which are mainly aimed at CPUs although they may have some possible connections with GPU parallelism. However, on the whole GPUs often have a fairly different architecture to CPUs which can result in more specialised tools being written for GPUs.

2.2.1 OpenMP

OpenMP [4] is an API designed to make it easier to code parallel programs. The API is currently designed for C, C++ and Fortran. The API allows the programmer to specify parts of code which are to be run in parallel using preprocessor directives. When this parallel code is executed it is executed by multiple threads, each thread has a thread id which the thread can find out by calling a special OpenMP function. When the end of the parallel code block is reached all threads are synchronised and after that only the “master” thread executes the remainder of the code [5].

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

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int th_id, nthreads;
  #pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
    #pragma omp barrier
    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return 0;
}
```

Example taken from wikipedia [5]

The above example illustrates a very simple OpenMP program whereby each thread prints “hello world” while running in parallel, with only the master thread (with thread id 0) printing how many threads there are.

OpenMP also has some additional features such as parallel for loops which allow the programmer to specify whether the for loop iterations are independent of one another and hence, can be executed in parallel. It also allows for more sophisticated for loop optimisation using reduction clauses which enable a for loop to operate on data in “chunks” and then combine these “chunks” at the end. This is particularly useful for loops which may, for instance, involve summing a set of data [5].

One interesting point to note is that there have been some recent talks about attempting to create a GPU version of OpenMP.

2.2.2 Unified Parallel C (UPC)

UPC [6] is a C parallelism extension that is in many ways similar to OpenMP. Unlike OpenMP it is more efficient in distributed cluster environments whereas OpenMP only runs efficiently in shared-memory environments. The reason for this is in the way UPC handles variables as it has extensions for how data types are declared. For example, it allows for things such as shared arrays to be split into thread local chunks, which is more efficient for distributed cluster environments as there is a large overhead in transferring data between clusters which does not exist on shared-memory systems [7].

2.2.3 C^N

C^N is yet another language designed for parallelism that is probably most similar to UPC as the language is mainly designed around the concept of poly (one copy stored on each processor core) and mono (one copy exists in main memory) [9]. Unlike UPC, however, it was mainly built as a way of programming Clearspeed's processors, which have some 96 cores and localised memory on each core [9,10].

One interesting point to note is that this architecture is in some ways vaguely similar to that of a GPU whereby each processing unit generally has its own local memory along with a global shared memory system.

2.2.4 The Codeplay Sieve C++ Parallel Programming System

The Codeplay Sieve C++ Parallel Programming System is another C++ parallelism extension which has some significant advantages over OpenMP and UPC, with the main ones being:

- Both OpenMP and UPC require the programmer to provide certain guarantees about the structure of the code in order for parallelism to work correctly. Otherwise, although the code may compile fine, unreliable behaviour may occur during execution. In the sieve system this limitation does not exist, although the compiler may throw up errors on compilation if certain dependency requirements are not met.

- The sieve system will give advice on how to improve the code and enable greater parallelism, hence allowing the programmer to improve the code for greater performance.

This is done by allowing the programmer to specify a block of code with a sieve marker, the compiler will then attempt to parallelise the code by delaying so-called “side effects” (parts of the code which cannot be safely automatically parallelised by the compiler) until the end of the code block. It also allows the programmer to define classes which allow things such as computing a sum within a loop to be optimised in a similar way to OpenMP's reduction clauses (as otherwise computation of the sum would be delayed until the end, which would result in practically no parallelism).

However, as we can see, unlike OpenMP and UPC, the additional automation and added debugging ability does come at a cost as one can potentially get greater performance out of OpenMP and UPC as they allow the programmer to program at a lower level, but they are much harder to debug. [11]

2.3 Shader Languages

Shader Languages are a certain type of language specifically tailored towards allowing a graphics programmer to program the GPU. They may have uses beyond graphics programming for more general purpose efforts although their design is heavily geared towards use in graphics.

2.3.1 Stanford Shading Language

This is a research project developed at Stanford with the intention of bringing “real-time programmable shading to mainstream graphics hardware”. It was begun in 1999 and they were essentially responsible for introducing the concept of shaders as a hardware abstraction layer to make it easier to program the GPU for graphics applications without getting bogged down in the specifics of the hardware specifications of individual chipsets. Essentially, the language uses the concept of “computation frequencies” with the reasoning being that sometimes one wishes to specify, for example, the colour of a vertex, at other times one may wish to specify the value of every “fragment” (essentially pixels but possibly including extra information such as alpha blending) for say, the purposes of texturing. The language introduced four different computation frequencies which were: Constant (evaluated at compile time), Primitive Group (i.e. a group of polygons), Vertex, Fragment. This allows for relatively easy programming while giving a reasonable amount of flexibility and efficiency. [32]

2.3.2 OpenGL Shader Language (GLSL)

The OpenGL Shader Language is an extension to the well-known OpenGL API for fast hardware accelerated 3D graphics. It was originally created as an extension to OpenGL 1.4, although it is now included in the OpenGL 2.0 core GLSL allows the programmer to write programs called “shaders” which run on the GPU. OpenGL defines two main types of shaders [12]:

1) Vertex shaders which run on the OpenGL vertex processor which is responsible for performing operations on individual vertices such as position, normal or texture coordinate transformation and also lighting or colour alterations [13].

2) Fragment shaders which run on the OpenGL fragment processor which is responsible for OpenGL's texturing, colour sum and fog stages [14].

GLSL programs are essentially text strings which are passed to the video card drivers at run-time and, hence, compiled at run-time [12].

2.3.3 High Level Shading Language (HLSL)

The High Level Shading Language is very similar to GLSL [15] except it also supports Geometry Shaders which are not currently supported in the core OpenGL 2.0 specification (although there are extensions to support these). Geometry shaders allow geometric objects to be manipulated (e.g. triangles, lines etc) [16]. The main differences between GLSL and HLSL lie in the syntax used for the two languages and that GLSL is designed to be used with OpenGL whereas HLSL is designed for use with Direct3D.

2.3.4 NVidia Cg

Cg is yet another shading language very closely related to HLSL (as Cg was developed in close collaboration with Microsoft who in turn developed HLSL) and it is semantically still similar to GLSL. It has the advantage, however, that it supports both OpenGL and DirectX APIs and so is capable of compiling shader programs for both APIs. [19]

2.3.5 Sh

Sh is a C++ metaprogramming extension language written for programmable GPUs. In some respects it is similar to TaskGraph in that Sh code is written by embedding it inside of C++ as a mini sub-language using techniques such as operator overloading which conveniently allows for a certain degree of type safety. However, Sh's GPU programming style resembles largely that of a shader language and it has support for various shader backends (such as GLSL).[26]

2.3.6 RapidMind

RapidMind is essentially the commercial successor to Sh which uses a similar metaprogramming methodology to Sh but is designed to allow the user to write generic parallel code in a stream processing fashion which the RapidMind library can then dynamically compile and optimise for the specific architecture which it is to be executed on. This includes compiling the code to execute on ATI, NVidia, x86 and Cell architectures.

2.4 Stream Processing

Stream processing is a programming paradigm whereby a series of functions (usually a single kernel function) is performed on each element of a 'stream' (sequence of data items) separately. This idea is useful because it can be used when writing parallel programs since the same kernel function can run in multiple threads but each instance will be passed a different element of the stream. [20]

In recent years this programming model has proven to be particularly useful because it is a relatively good abstraction for programming massively parallel architectures whereby having to handle the interaction details of every single processor core would be an extremely time-consuming, error prone, and difficult business. In particular, architectures where this has recently become particularly prevalent include GPU architectures which usually have several processor cores, sometimes referred to as multi-processors, and within each multi-processor it uses SIMD style parallelism between the various processors inside the multi-processor [8].

This makes stream processing a very good abstraction for this type of architecture.

2.4.1 Imagine

Imagine is a complete programmable architecture, developed at Stanford as a research project, which "exploits stream-based computation at the application, compiler, and architectural level". Essentially what Stanford have done is to design a complete programmable architecture that is programmed using a stream processing abstraction. The system is essentially designed to be a card which has its own processor and communicates with the host processor through the bus. It essentially uses SIMD at the hardware level and at the compiler level uses its own special stream programming language which generates code that runs on both the host machine (for communicating with the Imagine card) and code for the Imagine processor. [21, 22]

2.4.2 Merrimac

Merrimac is yet another stream processing project developed at Stanford. It is largely based on Imagine and differs in that it is targeted directly towards supercomputing applications and uses a high-radix network (i.e. a network where routers have a large number of ports in order to reduce latency from hops between different routers) for the different Merrimac nodes on the network. Each node is in many respects similar in architecture to that of Imagine and many of the Merrimac tools (e.g. compilers) are based on those developed for Imagine. [23, 24]

2.4.3 NVidia Cuda

NVidia's Cuda is a C-based language for programming their latest range of programmable GPUs (such as the Geforce 8000 series of graphics cards). The reason for development of Cuda is because GLSL (as discussed above) provides the wrong abstraction for programming GPUs as it does not allow the programmer to fully take advantage of the massively parallel architecture of the GPU. Cuda, on the other hand, provides a reasonably low-level abstraction for gaining maximum performance from the massively parallel architecture of their GPUs. Cuda is essentially based around the stream programming paradigm and thus it is similar to projects like Imagine except that unlike Imagine it is a commercial endeavour, is readily available, and has been designed as essentially an extension to NVidia's pre-existing graphic card technologies.

We shall now briefly examine the main architectural features of Cuda, as outlined in the "NVidia Cuda Programming Guide" [8], however, it is simply not possible to regurgitate the entire contents of a 143 page manual here, so while we will try to explain Cuda sufficiently so that the reader can grasp the general concepts involved in the rest of the report we strongly advise turning to the manual for a more in-depth understanding.

Cuda programs essentially have two main components:

1) The **host code** which runs on the CPU and is responsible for calling the kernel.

2) The **kernel** which is the code that is actually executed on the GPU.

This brings us onto a brief discussion about the exact parallelism paradigm that Cuda uses. As we mentioned it is based around the stream processing paradigm and in Cuda's case the execution of the kernel is organised as a "grid of thread blocks".

A thread block is essentially a collection of threads, limited by a certain maximum number of threads. All threads within a thread block can communicate with one another through some fast shared memory and, of particular significance, threads can synchronise with one another through the use of synchronisation points which forces all threads that have reached the synchronisation point to wait until all other threads have also reached this point. Furthermore each thread can identify itself by referring to its thread ID within the block, which is a built-in runtime variable (threadIdx which is of the special Cuda structure type dim3) maintained by Cuda.

Each thread block is then organised into a grid, threads in separate thread blocks cannot reliably communicate and synchronise with one another. It is however possible for a thread to identify which block it is contained in via the block ID runtime variable (blockIdx which is also of the special Cuda type dim3).

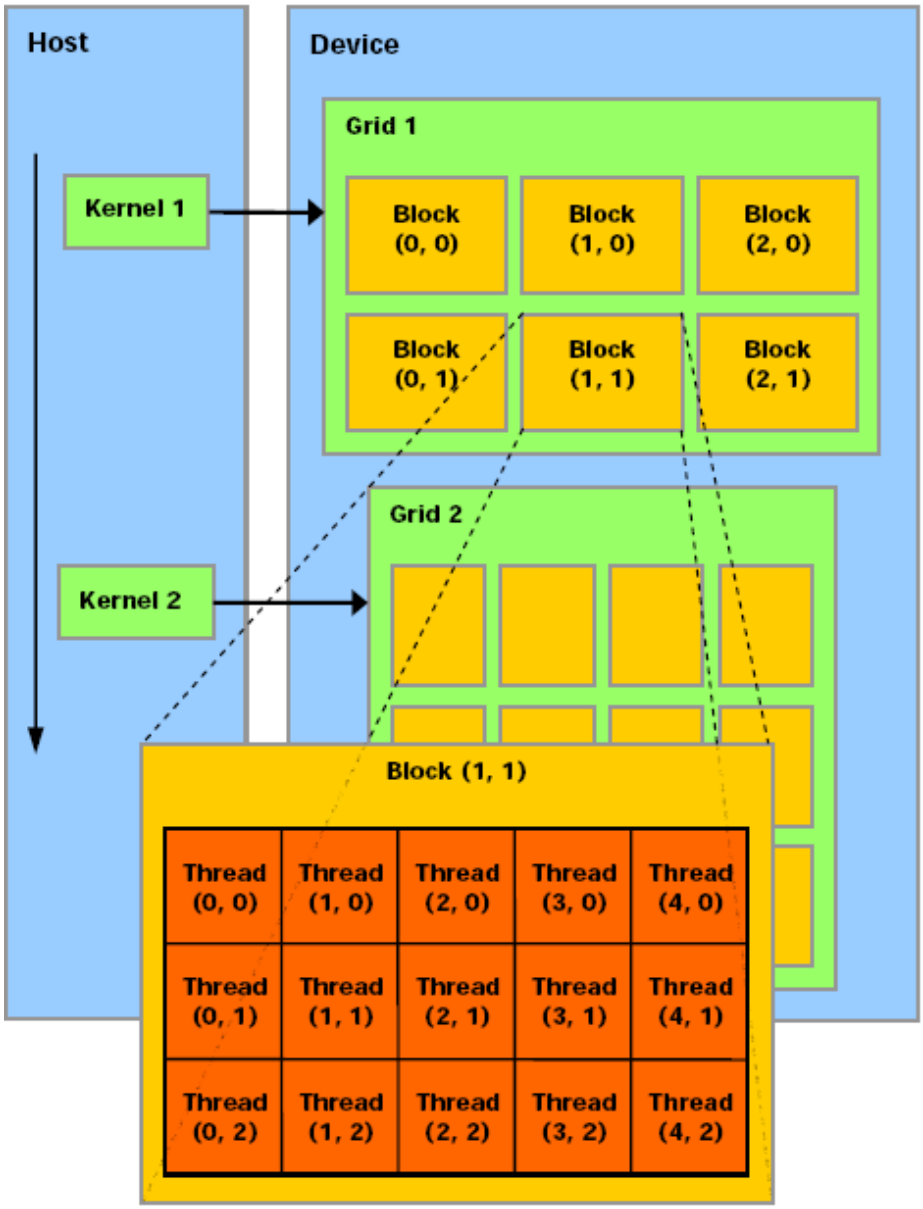


Figure 1.1: (taken from the "NVidia Cuda Programming Guide")

The size of each block and the size of the entire grid of blocks can be specified at runtime when calling the kernel using the special angle bracket notation, e.g.

```
myfunc<<<Dg, Db>>>(params...)
```

where Dg and Db are the additional parameters required by Cuda. Dg is the special Cuda structure dim3 and specifies grid dimensions, Db is also of type dim3 and specifies the dimension and size of each block.

This does of course require a brief discussion of the Cuda built-in structures. Cuda introduces several built-in structures, including dim3. These are:

char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, dim3.

Essentially the number at the end represents the number of components in each structure and the prefix to that number indicates the type of each component, e.g. ulong4 has 4 components x,y,z and w each of type unsigned long. dim3 is a special structure generally used for specifying dimensions, it is based on uint3 but any component left unspecified is initialised to 1 (since one should not specify a grid of blocks that has one of its dimensions as 0).

As was previously mentioned, Cuda has some runtime variables which are only accessible from the kernel, there are in fact four runtime variables these are namely:

- **gridDim** (of type dim3) which contains the dimensions of the grid of blocks.
- **blockIdx** (of type uint3) which contains the index number of this thread's block.
- **blockDim** (of type dim3) which contains the dimensions of this block.
- **threadIdx** (of type uint3) which contains the index number of this thread within the block.

Finally, we make note that NVidia's GPUs have various different types of memory which the Cuda programmer can explicitly specify when declaring variables:

- **Constant memory** - very fast cached memory, which cannot be modified by kernel code and is only 64kB
- **Shared memory** - very fast on-chip memory which is shared between all the threads of a block. However, it can suffer from *bank conflicts* where multiple memory accesses are addressed to the same bank reducing performance, it is also limited to 16kB per chip.
- **Texture memory** - This is also cached memory, although it is generally not quite as fast as constant memory.
- **Global memory** - This is uncached memory and is very slow although there is plenty of it and often data will be transferred from host to global memory and the kernel code will then load the data from global memory to shared memory in order to carry out its operations.

So, to summarise, primarily the Cuda language has C extensions which allow for both execution of the kernel and interaction between the host and kernel code. It also adds various new data types which are used for programming the GPU, along with the ability to specify which type of memory the data is to be stored in.

2.4.4 Brook

Brook is a stream programming language developed at Stanford which extends the C language. It is in many respects similar to Cuda, except that the concept of a stream is made somewhat more explicit with the way variables are passed from the host to the kernel. In Brook, the user declares a stream data type which indicates that the items in the stream can be operated on in parallel by the kernel [25]. This is in contrast to Cuda in which it is necessary to explicitly allocate device memory, copy an array over, specify the number of blocks/threads and then in the kernel one can use Cuda's builtin thread/block identifier variables to determine which part of the array to access [8]. Clearly Cuda's method is somewhat more involved, although potentially more flexible than Brook in terms of optimisations.

2.4.5 ATI FireStream

ATI FireStream is essentially ATI's alternative to NVidia's Cuda, which at the time of starting this project was in “pre-production alpha release” mode. In other words, an alpha version of the SDK was available in order to enable potential developers to begin experimenting with the compilers and libraries for the technology. The

processors themselves were not due to begin marketing until Q1 2008. ATI FireStream is essentially based around the Brook stream programming language. [27]

2.5 Applications of GP-GPU

This section is finalised by briefly discussing some examples of some very promising areas where GP-GPU (General Purpose GPU) programming has been successfully applied.

2.5.1 OpenVIDIA : GPU accelerated Computer Vision Library

The OpenVIDIA project is an intriguing and potentially very useful example of one of the many potential applications for today's high powered GPUs. The project takes advantage of OpenGL and Cg in order to perform various computer vision tasks in real-time. From the project's home page [28], its key features are as follows:

- Edge and Corner Detection
- Feature Based Corner Tracking
- Skin Tone Tracking
- Color Object Tracking
- Contrast/Brightness Image Enhancement
- Video Orbits Image Compositing Algorithm

The project also has support for what they call “Mediated Reality”, which they define as a “computationally enhanced version of reality that mixes both the real and virtual worlds”. This results in a couple of additional, and very interesting, features (as taken from their home page):

- 2D virtual tags to label the real world around the user
- 3D objects created in 3D Studio or Cal3D to be inserted into the real world (still under development)

This is of course all done in real-time as a result of their utilisation of the massively parallel architecture of GPUs.

2.5.2 Building a Million Particle System

This paper discusses an implementation of a million particle system using the floating point programmable pixel pipeline, generally available with the various shading languages (GLSL/HLSL/Cg etc). It can only be seen why this is such an achievement when looking at how many particles can be handled using the CPU to do most of the work. CPUs can generally only handle up to around 10,000 particles. This gives a huge speed boost of around 100x and hence shows how much some applications can benefit from the use of a GP-GPU [29].

2.6 Conclusion

This section has discussed a variety of interesting projects all of which are very closely related to our work. It has been shown that there are a variety of subtly different parallel programming paradigms for CPUs which have potential to be applied to GPUs, and two main programming paradigms specifically for GPUs - shaders and stream processing. Although in some respects it can be argued that shaders are a type of stream processing language, they do vary quite significantly from the more generic and powerful alternatives such

as Cuda. Also discussed was a more in-depth look at TaskGraph and Cuda, which this project is based on, and a discussion of some of the various applications that GP-GPU benefits in which it was shown that it can offer very substantial performance advantages.

It is interesting to note that Sh and RapidMind are very similar to this project, however, they are both a relatively high level abstraction for programming GPUs and as such do not allow for the potential speed and flexibility of languages such as Cuda. This is because Cuda exposes a lot of low level details which many other languages do not. This potentially makes it more awkward to program, however, the performance advantages can be quite significant.

3 Design of the TaskGraph Extension

This section discusses the design of the TaskGraph extension. It begins by discussing the general design considerations that were taken into account when designing the TaskGraph extension, it then moves on to discuss the various generic extensions followed by the Cuda extensions including how they work from a users perspective, along with some discussion of the design alternatives.

3.1 General Design Considerations

The main aim for the design of the TaskGraph extension was to implement functioning Cuda support so as to make it possible to code functioning examples in the TaskGraph language. However, there are also a variety of additional considerations which were taken into account:

- Design the language extension features such that they are clean, legible and convenient to use
- Ensure consistency of the TaskGraph language so that both Cuda code and non-Cuda code do not differ except where necessary
- Ensure consistency between Cuda and TaskGraph Cuda languages so that the difference between coding normal Cuda and TaskGraph Cuda is kept to a minimum.
- Ensure that the extension is designed around the current TaskGraph implementation so as to reduce unnecessary workload in extending TaskGraph which result from some things being difficult to implement given the current TaskGraph implementation.
- Take into account that some additional features useful/required for Cuda may also be useful for non-Cuda TaskGraph applications. Thus, it was important to base our Cuda design around allowing both Cuda and non-Cuda support for some features.
- Ensure the type-safety properties of TaskGraph are maintained as much as possible.

3.2 Generic TaskGraph extensions

3.2.1 Pointer Support

Some sort of pointer support was an essential feature for Cuda since often a Cuda programmer will need to allocate device memory, assign it to a pointer and then pass it to the Cuda kernel, and, at some point, the programmer may then wish to read back the data from device memory by allocating it to a pointer.

An alternative approach would have been to have TaskGraph handle the pointers “behind the scenes” and give the programmer an abstraction such that they can just pass an array to the kernel function which would then return the result as an array. This would be nice in terms of its safety properties in that there is less ability for the user to write all over the memory address space and cause segmentation faults. However, it is our view that since TaskGraph is a language written inside of C++ there is no point in trying to add stronger safety properties than its parent language supports as the programmer could still write all over the memory address space if they desire. Furthermore, such an approach could potentially limit the amount of flexibility and thus reduce the ability of the user to write efficient code which is a major aspect of the motivation for both this project and the reason why Cuda was written.

Thus, our pointer support additions have the following features:

- Support for dereferencing pointers by accessing them as though they were an array, e.g. myptr[7] = 5;

- Support for allowing TaskGraphs to return pointers and accept them as parameters.
- Support for obtaining a reference to a variable, e.g. myptr = &a;

3.2.2 Parameter Passing Changes

One of the major changes to the TaskGraph library was that we decided to change the way parameters were passed. Previously, when parameters were passed to a TaskGraph it would actually pass a reference to the specified parameters to the TaskGraph function. The following code snippet illustrates how this works:

```
#include <stdio.h>
#include <stdlib.h>
#include <TaskGraph>

using namespace tg;

typedef TaskGraph<int, int> example_TaskGraph;

int main( ) {

	example_TaskGraph T;
	int b = 1;

	taskgraph( example_TaskGraph, T, tuple1(a) ) {
			a = 7;
	}

	T.compile( tg::GCC, true );
		T(b);
	printf( "b: ", b );
}
```

In the above example, a TaskGraph is declared which returns a value of type int and accepts a parameter of type int. The variable 'b' is then initialised to 1 and all the TaskGraph does is set 'a' to 7. If this was a normal C function then the TaskGraph would be expected to essentially do nothing when executed, however, when the TaskGraph is executed with 'T(b)' the parameter 'a' becomes a reference to b, thus, b will be set to 7 and so our printf call will print “b: 7” instead of the “b: 1” which would usually be expected with C.

This feature can sometimes be nice in the absence of pointer support since it allows the programmer to directly modify the parameter variables if they so desire, however, it causes problems with Cuda support. To understand why, suppose a reference is passed to a Cuda kernel from host code, the kernel is of course code being executed on the device and is thus inside a different address space, this means that the reference is no longer valid when inside the kernels address space. This issue left us with two possible options:

- Use a different parameter passing method for Cuda kernels, breaking consistency and potentially causing confusion for the programmer
- Modify the parameter passing method to function in the normal C style ensuring consistency but potentially breaking compatibility with older TaskGraph dependent code. This would also require the addition of pointer support (as discussed in the previous section) in order to still allow by-reference parameter passing.

It was decided to opt for the latter option as TaskGraph is essentially a research project and thus breaking backwards compatibility is not quite as bad an option as it would be for some other projects. Also it had already decided that adding pointer support would be essential for other reasons (see 3.2.1).

3.2.3 Improved Function Call Syntax

The function call syntax has been improved so that calling external library functions in TaskGraph is more legible and less involved. While this improved legibility is not an absolute necessity it is nonetheless important for Cuda since Cuda requires the programmer to call a large number of different functions.

Before discussing the modifications, let us look at the approach that was originally used for calling TaskGraphs, observe the following code snippet:

```
#include <stdio.h>
#include <stdlib.h>
#include <TaskGraph>
#include "TaskUserFunctions.h"

using namespace tg;

typedef TaskGraph<int> addc_TaskGraph;

int main() {

        addc_TaskGraph T;

        taskgraph( addc_TaskGraph, T, tuple0() ) {
                tVar(int,mylen);
                static TaskFunction1<int, char*> func ( "strlen" );
                mylen = func.call ( "hey" );
                tReturn(mylen);
        }

        T.compile( tg::GCC, true );
        printf( "Length: %d", T() );
}
```

The TaskGraph code above first defines a function "strlen" using TaskFunction with a suffix of 1 (indicating that it takes 1 parameter), it then proceeds to specify as a template the parameter/return types that the function takes, the return type is specified as int and the first parameter as char*. In the next line the function is then called, passing parameter "hey" and the return value is then stored in mylen.

Clearly, it would be ideal if this call could all occur on one line, furthermore, it is not nice for the programmer to have to specify the parameter types for the function being called nor is it nice for the programmer to have to specify a numeric value for the number of parameters. An alternative to the mechanism above would be for the TaskGraph to have an inbuilt function which could then be called in a manner which is the same as any normal C function call. Observe the following example:

```
        taskgraph( addc_TaskGraph, T, tuple0() ) {
                tPrintf("hello world");
        }
```

The obvious disadvantage of this method, however, is that one cannot call functions which the TaskGraph library does not have a wrapper for, unless the user writes the wrapper themselves, but this is again not particularly desirable.

As a result of these issues, a new method of function calling was developed which involves simply passing the function name string and its parameters to a tCall function, tCall(char* funcname, params...). This is best explained by observing the modification of our above example to use the new method:

```
        taskgraph( addc_TaskGraph, T, tuple0() ) {
                tVar(int,mylen);
                mylen = tCall("strlen", "hey");
```

```
        tReturn(mylen);
}
```

This is a much cleaner solution.

3.2.4 tInclude

There is still one last problem with our improved function call syntax above (see previous section). Since the compiler is not told the parameter/return types of the strlen function it has no idea as to what function definition to export. For this reason, an additional tInclude(char* headerfile) TaskGraph function was created which allows one to include header files in a similar manner to normal C. Now observe our final example:

```
taskgraph( addc_TaskGraph, T, tuple0() ) {
        tVar(int,mylen);
        tInclude("stdlib.h");
        mylen = tCall("strlen", "hey");
        tReturn(mylen);
}
```

This syntax is much more familiar and natural to the normal C programmer than the original version, it also involves less typing and greater legibility, especially when one uses a lot of functions from the same include file (which experience tells is relatively common).

3.2.5 Syntactically Safe Method of TaskGraph Function Calls

One of the issues which has not been directly addressed before is how to call other TaskGraphs directly from within a TaskGraph. The original TaskGraph implementation did support a method of calling other TaskGraph functions by using a proxy function which calls back to the statically compiled code, which would then call the TaskGraph function. However this is a rather ugly hack and could potentially cause a performance hit and, in the case of Cuda, it is absolutely vital that the host code can call the kernel TaskGraph directly. If the host code doesn't call the kernel TaskGraph directly then one would need to use the Cuda compiler to compile some of the statically generated code (namely the proxy function) which would break the single source model and again, could cause performance hits, thus it is an undesirable solution. Furthermore, it still involved passing a string of the function name, which is again not the ideal solution as it bypasses any syntax safety checks (e.g. checking whether the function being called actually exists).

The method described here, however, is very elegant in that not only can a TaskGraph be called directly but it is not necessary for a string consisting of the function name to be passed either - the TaskGraph is simply passed through to the usual tCall function (see section 3.2.3). Hence, the syntactically safe part of our method. The following code snippet illustrates how our method works:

```
#include <stdio.h>
#include <stdlib.h>
#include <TaskGraph>
#include "TaskUserFunctions.h"

using namespace tg;

typedef TaskGraph<int> example_TaskGraph;
typedef TaskGraph<int, int> add_TaskGraph;

int main() {

        example_TaskGraph T;
        add_TaskGraph T2;

        taskgraph(add_TaskGraph, T2, tuple1(a))
```

```
	{
		tReturn( a + 1);
	}

	taskgraph( example_TaskGraph, T, tuple0() ) {
		tVar(int,result);
		result = tCall(T2, 5);
		tReturn(result);
	}

	T.compile( tg::GCC, true );
	printf( "Answer: %d", T() );
}
```

In the above code two TaskGraphs are declared, T and T2. When T() executes then a function call is made with tCall to T2 which passes 5 through as a parameter. T2 then adds 1 to 5 and returns. The result of T2 is then stored in 'result' and returned. Thus our printf routine will print: "Answer: 6"

3.2.6 Recursion

A point which is worth a brief mention is that TaskGraph functions are even capable of calling themselves recursively. The following code snippet illustrates this:

```
#include <stdio.h>
#include <stdlib.h>
#include <TaskGraph>
#include <TaskUserFunctions.h>

using namespace tg;

typedef TaskGraph<void, int> recursion_TaskGraph;

int main() {
	recursion_TaskGraph T;

	taskgraph( recursion_TaskGraph, T, tuple1(a) ) {
	 tPrintf("a: %d", a);
	 tIf (a > 0) tCall(T, 0);
	}

	T.compile( tg::GCC, true );
	T(1);
}
```

When T(1) is called the TaskGraph T is executed and prints "a: 1", it is then called again through a recursive call and prints "a: 0", it then terminates. This is a simple example of recursion in TaskGraph.

3.2.7 Resolving Function Call Dependencies

A topic which is worth mentioning is how the issue of function call dependencies was resolved, for example, a TaskGraph T1 may call another TaskGraph T2, if T1 is then compiled, the question is what happens to its dependency on T2 as T1 cannot be executed unless T2 is compiled as well.

Our method of solving this issue is that a TaskGraph that is dependent on other TaskGraphs can simply be compiled and executed in the usual way and the TaskGraph library will scan through both its dependencies, and its *dependencies dependencies* and so on and thus the TaskGraph will be compiled along with all the TaskGraphs it is dependent on.

3.2.8 Compiler Options Passing

An additional feature that was added allows the programmer to pass options to the compiler, for example, suppose a TaskGraph T has been created then it can be compiled with certain compiler options as follows:

```
const char* args[] = {"-Xcompiler", "-fPIC", "-I", "/home/tp105/NVIDIA_CUDA_SDK/common/inc/", "-lGL", "-
lGLU", "-lcutil", NULL};
T.compile(tg::NVCC, true, args);
```

i.e. args is an optional parameter to T.compile which stores an array of arguments. This list must be terminated by NULL since otherwise C++ cannot determine how many arguments there are.

This could have been done differently by allowing the user to place the arguments in the T.compile call, however, due to limitations in C++ with variadic functions it would still be necessary for the NULL parameter. Furthermore, since there can be a lot of arguments it is probably cleaner to simply store the arguments in a separate list.

3.2.9 Indeterministic Control Flow Instructions

Indeterministic Control Flow Instructions are a useful feature, which could make TaskGraph easier to program with in some cases. They are designed for situations where it cannot be determined at compile time of the generator whether a particular TaskGraph control flow instruction should exist or not. Yet regardless of whether or not the instruction exists the user wants the statements in its body to exist. For example, consider the following TaskGraph code which loops over the counter test and increments it by 1 from 0 to 5:

```
tVar(int, test);
tFor(test, 0, 5)
{
        ...
}
```

A user may want to be able to determine whether the above loop should be unrolled or not at runtime and then this could influence some of the statements in the loop. For example, we might be specialising and if the loop is unrolled then we may want some of the statements to refer to different variables (perhaps standard C++ variables rather than TaskGraph variables so they can be compiled in as constants).

One could be forgiven for thinking that the problem is easily solved - simply add an if statement in front of the tFor to determine whether a normal for loop should be used or a tFor loop, e.g.

```
bool specialise;

...
if (specialise) tVar(int, test); tFor(test, 0, 5) else for (int test=0; test <= 5; test++)
{
        ...
}
```

However, this syntax is not allowed by C++ as the braces must match one loop or the other, not both. Clearly, a simple solution that would actually work would be to have multiple versions of the code with the minor changes added, for example:

```
bool specialise;
...
if (specialise)
{
        tVar(int, test);
        tFor(test, 0, 5)
```

```
	{
		...
	}
}
else
{
	for (int test=0;test <= 5; test++)
	{
		...
	}
}
```

However, if the content of these loops is anything other than trivial and especially if there might be multiple parts of the code where the existence of the control flow instructions cannot be determined at compile time (which could even be nested inside these two loops, resulting in some four cases to be taken care of), then asides from looking very long this code also becomes fairly difficult to maintain as there is a lot of redundant code and small modifications require changes to many parts of the code.

Thus, our solution adds a separate bunch of statements which add the suffix "Maybe" to TaskGraph's standard control flow instructions, namely creating the instructions: tForMaybe, tForStepMaybe, tForDownMaybe, tForDownStepMaybe, tWhileMaybe, tIfMaybe. These instructions take the same parameters as the normal control flow instructions except for one additional parameter at the end which takes a C++ boolean data type (i.e. it must evaluate to bool at code generation time). If this value is true then the control flow instruction will appear as usually expected in the generated code, if it is false then the control flow instruction does not appear in the generated code but all statements inside its braces will appear in the generated code.

Thus, our solution to the problem discussed above could be:

```
bool specialise;

...
tVar(int, test);
tForMaybe(test, 0, 5,!specialising)
{
	for (int test2=0;test2 <= 5 && (specialising || test2=0); test2++)
	{
		...
	}
}
```

Examining the above code carefully, it should be evident that if specialised is true then tForMaybe will not generate a for loop in the TaskGraph generated code and the inner C++ for loop will loop 6 times, hence unrolling the loop. If specialised is false then tForMaybe will appear in the TaskGraph generated code, looping 6 times, and the inner C++ for loop will only loop once.

Of course, the above design is not perfect as it has several flaws. Firstly, the loop variable for both loops must be different, if the loop variable is referred to frequently in the code inside the loops then there is likely to be a lot of special case code for handling whether the loop is specialised or not in order to determine which loop variable to use. Furthermore, the declaration for the normal C++ for loop is a little contrived as it requires a fairly complicated boolean expression to ensure that it operates correctly. However, while these are issues it does still have the potential to reduce the amount of redundant code and in the very least makes it easier to tweak as both specialised/unspecialised versions of the same statement will be next to one another rather than on different pages of the screen. Furthermore, while this has been used for loop specialisation in both our simple example here and the ray tracing example (see section 5.4) it might be that there are other uses for it which could actually result in cleaner code.

3.3 Cuda Specific TaskGraph extensions

3.3.1 NVidia Cuda Compiler

The design decision here was fairly obvious - ensure consistency with the original TaskGraph implementation since it provided support for different compilers. For example, originally the GCC compiler could be called with:

```
T.compile(tg::GCC, true, args);
```

So, to enable Cuda support, a tg::NVCC compiler specifier was added so the Cuda compiler can be used with the following statement:

```
T.compile(tg::NVCC, true, args);
```

3.3.2 Cuda Kernel, Host and Global Calls

Any Cuda program consists of the host code which runs on the CPU and the kernel code which the host code will call and runs on the GPU. The mechanism for host code calling kernel code is done via global calls which are special calls made from the host code to execute some piece of kernel code [8].

It was decided that the best way to tackle this would be to allow the user to create two TaskGraphs, one would be a kernel TaskGraph, the other would be a host TaskGraph. The user can then use a tCudaGlobalCall function to generate a global call from the host TaskGraph to the kernel TaskGraph. It would also be essential that the user can specify, when a TaskGraph is declared, whether it is a kernel TaskGraph or a host TaskGraph in order to allow the library to determine whether or not to put a __global__ type qualifier (which tells the Cuda compiler whether the function is kernel or host code) in front of the function name in the generated code. This is done by creating a cudakerneltaskgraph macro and a cudahosttaskgraph macro, which operate in the same way to the usual taskgraph macro used to declare TaskGraphs, except that they specify that the TaskGraph is either a kernel TaskGraph or a host TaskGraph.

There is also one final design issue to address with this system. When global calls are made not only must the usual parameters be specified but also some additional parameters must be specified (see section 2.4.3). In standard Cuda these are normally specified by having the function call in the form:

```
myfunc<<<Dg, Db, Ns>>>(params...)
```

where Dg, Db and Ns are the additional parameters used by Cuda, enclosed in special angle brackets. Dg is of type dim3 and specifies grid dimensions, Db is of type dim3 and specifies the dimension and size of each block and Ns is an optional argument of type size_t which specifies the number of bytes in shared memory that is dynamically allocated for each block, this normally defaults to 0 [8].

Since C++ does not support such angle bracket syntax our method works in essentially the same way the tCall function works (see sections 3.2.3-3.2.7). The additional parameters are handled by making all three compulsory (so if the user doesn't care about Ns then they can simply set it to 0). So our syntax becomes of the form:

```
tCudaGlobalCall(T, Dg, Db, Ns, params...);
```

Thus, the final design is illustrated with the following code snippet:

```
using namespace tg;

typedef TaskGraph<void> cuda_host;
typedef TaskGraph<void, int> cuda_kernel;
```

```
int main( int argc, char *argv[] ) {
	cuda_host T;

	cuda_kernel T2;
	cudakerneltaskgraph(cuda_kernel, T2, tuple1(myparam))
	{
		/* ...code... */
	}
	cudahosttaskgraph(cuda_host, T, tuple0())
	{
		tVar(dim3, Dg);
		tVar(dim3, Db);
		tVar(int, Ns);
		tVar(int, param1);
		/* ...code... */
		tCudaGlobalCall(T2, Dg, Db, Ns, param1);
		/* ...code... */
	}
	/* ...code... */
	T.compile(tg::NVCC, true);
	T();
}
```

This code generates the two TaskGraphs and then calls compile on the host TaskGraph, T. As mentioned in section 3.2.7 this will resolve all the dependencies between the two TaskGraphs, so since T makes a global call to T2, both of these TaskGraphs will be compiled together automatically. Next, T() is executed, which will then make a global call to the kernel T2 passing the special compulsory Cuda parameters Dg, Db and Ns which are of types dim3, dim3 and int respectively (the way these parameters are declared will be discussed in the next section). It also passes the user-defined parameter param1, which is required by the declaration of T2.

3.3.3 Cuda Built-in Structure Support

TaskGraph already had support for standard structures, which worked in a very similar way to normal C. However, the issue with Cuda is that it has built-in structures which the user does not declare. Hence, so as to ensure that our design is similar to Cuda in this regard, and also to ensure that we can easily prevent the built-in structures being declared in the generated code, our design works in exactly the same way as normal TaskGraph structures but all Cuda structures are pre-declared.

The following code snippet is provided to briefly illustrate how the structure support works:

```
	/* ...code... */
	cuda_kernel T2;
	cudakerneltaskgraph(cuda_kernel, T2, tuple1(myparam))
	{
		/* ...code... */
		tVar(dim3, test);
		test.x = 5;
		test.y = 3;
		test.z = 7;
		tVar(dim3, test2);
		test2 = test;
		/* ...code... */
	}
	/* ...code... */
```

"test" is declared as the Cuda built-in structure "dim3", this structure has components x,y and z (all of which

are of type uint) so for illustration purposes the code above then assigns various values to each of these components. Finally, it then declares another dim3 variable, test2, and assigns test to test2.

3.3.4 Cuda Built-in Variable Support

As mentioned in section 2.4.3 Cuda has support for four built-in variables [8], accessible from the kernel only, namely:

- gridDim (of type dim3) which contains the dimensions of the grid of blocks.
- blockIdx (of type uint3) which contains the index number of this thread's block.
- blockDim (of type dim3) which contains the dimensions of this block.
- threadIdx (of type uint3) which contains the index number of this thread within the block.

Clearly each of these must be implemented, the design for how structures works was discussed in section 3.3.3, here a similar method is used, but not only does the TaskGraph programmer not have to declare the structure, they need not declare the built-in variables either and they may be used straight away inside the Cuda kernel.

3.3.5 Cuda Variable Type Qualifiers __shared__ and __constant__

There are two essential Cuda variable type qualifiers that we have implemented:

- __shared__ is used to declare a variable that resides in the shared memory space of a block, that is, it can be accessed by any thread within the block. [8]
- __constant__ is used to declare a variable that resides in constant memory. One of the reasons why constant memory was important is that it is cached and the cache works such that in some cases it can be as fast as a read from a register. [8]

This design works by creating modified versions of TaskGraph's tVar macro (which is used to declare TaskGraph variables), namely, tVarShared and tVarConstant. These work in exactly the same way as tVar except they create shared and constant versions respectively of the variable passed as a parameter. For example, tVarShared(uint3, test) will create a shared variable called test of type uint3.

One important point to note however, is that while shared variables are relatively straightforward in that they can be used just like normal variables for the most part, constant variables are somewhat more awkward. In particular, constant variables in Cuda have to be global variables rather than local variables and typically the host code will call cudaMemcpyToSymbol to copy some data from host memory to the constant memory variable. This means that essentially two TaskGraphs need to be aware of the same variable. This is further complicated by the fact that the total amount of constant memory occupied by variables cannot exceed 64k and Cuda seems to provide no way of deallocating constant memory. Furthermore, its documentation remains rather vague on what happens if multiple applications use 64k of constant memory, or, for that matter, what happens if multiple DLLs each use 64k of constant memory? Will the drivers swap out the constant memory correctly or will it cause a crash?

Our solution, therefore, requires the user to declare the constant variable in all the TaskGraphs in which it will be used, when the TaskGraphs are all compiled together the TaskGraph library will ensure that it declares constant variables of the same name in different TaskGraphs only once, thus allowing a mechanism of referring to the same variable in different TaskGraphs. Unfortunately this does not resolve the potential issue of running out of constant memory, however, it is perhaps the best solution to what appears to be a fairly poor design decision on the part of NVidia.

There is, however, one final issue to contend with. The TaskGraph syntax allows the user to declare arrays by writing e.g. tVarShared(float[50], test) which will declare an array test consisting of 50 floats. However, we may wish to be able to determine the size of the array in the generated code at runtime of the generator,

since that is, after all, the whole point of metaprogramming. This poses a problem since the way C++ works means that it will not accept a variable parameter for the size of the array. Thus, we are forced to use some old code which was in TaskGraph that uses a nastier declaration but, for instances where the size of the array can be variable, is the only way to do it. The construct is:

tArrayFromList(type,x,ndims,sizes)
where type is the type of the elements in the array (e.g. float), x is the name of the variable to be created, ndims is an integer specifying the dimensions and sizes is a pointer to a normal C++ array which stores the size of each dimension in the array.

Thus, using this specification we came up with two new constructs, tSharedArrayFromList and tConstantArrayFromList which operate in exactly the same way as tArrayFromList except they create a shared array and a constant array respectively. Asides from the syntactical difference, tSharedArrayFromList and tConstantArrayFromList are exactly the same as tVarShared and tVarConstant.

3.4 Conclusion

This section has provided an overview of the design of the TaskGraph extension. Details of how the extension is intended to work from a users perspective have been discussed along with some discussion of our design considerations including the design alternatives and limitations that may exist with C++/Cuda. The next section will discuss technical details of how the extension was actually implemented.

4 Implementation of the TaskGraph Extension

This section discusses the various challenges of implementing the TaskGraph extension and how they were overcome. It discusses in detail how the design features discussed in section 3 were implemented.

4.1 Key Challenges

One major challenge to take note of is that TaskGraph uses SUIF (see section 2.1.2) as its backend for generating the source code which is then passed to the appropriate compiler at runtime. However, Cuda makes several changes to the standard C source code (see section 2), thus, not only did the TaskGraph library need to be modified but its SUIF backend needed modifying as well.

The other major challenge is that TaskGraph uses a lot of complicated macros and C++ templating in order to allow for its type-safe C sub language design. This meant that we had to try to understand these fairly thoroughly and extend them to support the features that needed to be implemented.

4.2 Generic TaskGraph extensions

4.2.1 Pointer Support

Prior to our work, TaskGraph already had a basic TaskPointer class in place, however, this class was incomplete and essentially did nothing except for allowing declaration of a pointer with the tVar macro. This class is ultimately the class that gets instantiated when declaring a pointer variable and thus is the class that gets manipulated when trying to access the pointer.

In order to add pointer support, this class needed to be modified to override the [ ] operators, thus allowing array style pointer dereferencing. Some additional changes were also made, partly using code from TaskScalarVariable, to ensure that pointers can be assigned to other pointers and especially to the return value of a function call (as this is necessary for Cuda malloc routines).

The final modification was to both the TaskScalarVariable and TaskPointer classes to add support for obtaining a reference to a variable, this was done by simply overriding the & operator thus allowing one to obtain a reference to a variable in the same way as normal C. This also required some additions to the SUIF TaskIR class (which is essentially TaskGraphs interface to the SUIF backend) and the coreTaskGraph class (which inherits from TaskIR) in order to allow SUIF to correctly add the reference into the generated source code. It should be emphasized, however, that this did not require any modifications to the SUIF backend.

4.2.2 Parameter Passing Changes

In order to change the parameter passing mechanism to a style more akin to normal C (see section 3.2.2) several changes had to be made to the TaskGraph code.

The first issue was that the way TaskGraph generated the C source code needed to be changed. Previously, the runtime compiled TaskGraph code just defined a void** parameter in the function declaration and then did various pointer accesses to store the correct parameter in the correct variable. Thus, the SUIF TaskIR class needed to have several functions modified. Firstly the code was changed to move the local SUIF function declaration variable (i.e. the SUIF variable which declares the function in the generated code) to class scope in TaskIR so that the parameters can be updated in the actual function definition AFTER TaskIR was instantiated as the actual parameter type information is only passed to TaskIR after it has been instantiated (since the previous method only used void** and thus did not need to know anything about the parameters at initialisation). addVariableDefinition and addArrayDefinition were then changed to tell SUIF to handle the parameters in the format required in the actual function definition rather than using the previous method of typecasting them from void** and then using pointer accesses to get the correct variable.

The second issue was in how the parameters were passed to the TaskGraph runtime compiled function when the function was executed. The previous method essentially built up an array of pointers to the parameters and then passed the array to the TaskGraph runtime compiled function. This needed to be changed so that the parameters are passed through directly. The solution to this involved some restructuring of the TaskGraph code since it was then necessary to know the entire parameter list when the TaskGraph runtime compiled function is actually called (whereas before only the void** pointer needed to be known). We moved the execute() function (which is essentially called when executing a TaskGraph) from the ReturnTaskGraph to the SubTaskGraph class since various information on the parameters was needed which were much easier to obtain within that class. Specifically, we needed to pass around more template arguments for the parameter types between the various classes. This was essential since execute() calls a function callback pointer to the runtime compiled TaskGraph function and, while originally this pointer could be a fixed type, it now needs to be typecasted to accept the correct number of parameters and type of parameters.

It should be noted, that some of the changes that occurred here have probably broken Cell support (which TaskGraph originally supported) as some of the parameter passing code had various special cases for Cell and we largely ignored these parts of the code. This likely means that TaskGraph will no longer compile for Cell, however, as this investigation is not about Cell and the author has no previous experience with Cell it was decided that it would be simply too time consuming to attempt to fix the code to work correctly with the Cell architecture.

4.2.3 Improved Function Call Syntax

In order to create the tCall style function syntax (see section 3.2.3) we had to use several clever tricks.

```
tCall(char* funcname, params...);
```

Firstly, it was observed that the tCall function essentially needed to be a variadic function, that is, it must take a variable number of parameters. C++ has support for variadic functions, however, the way TaskGraph works is that each parameter is passed through as a TaskExpression object, this is somewhat problematic as the variadic functions are really leftovers from C and thus do not support passing objects, which is a C++ only construct. This problem can be overcome if a pointer is passed to the TaskExpression object, however this requires the use of the & symbol in the function call so as to ensure that a pointer to the object is passed rather than the object itself, which is obviously an undesirable method. The other method that was considered is that parameters can be declared as reference types in C++ which avoids the use of the & operator in the function call, however, C++ did not seem to support this either for use in variadic functions.

The other issue with variadic arguments is that there is no way to determine the number of arguments that have been passed, the only way is for one of the arguments in the function call to specify how many arguments are to be passed, but again this is obviously not a desirable solution.

Thus our final solution is to declare an internal tCall function which uses the pointer method of variadic arguments and accepts as its first parameter the number of arguments as an integer. Then, our actual exported tCall function, which the TaskGraph user calls, is a series of repeated inline functions using function overloading so the first function would accept one argument, the next function accepts two arguments and so on up to a reasonable limit (our code supports up to 10 arguments as this was adequate for our investigation, although more can easily be added if necessary, and probably should be if used in a production environment). The contents of the exported tCall function is then simply an appropriate call to the internal tCall function which conveniently hides all the nasty syntax from the user.

This is best illustrated with the following code snippet from our implementation:

```
...

TaskExpression coreTaskGraph::tCall_internal(int argnum, const char* funcName, ...) {
/* ...code... */
}
```

```
...

__inline__ TaskExpression tCall(char *funcname)
{
 return CURRENT_TASKGRAPH->tCall_internal(0, funcname);
}

__inline__ TaskExpression tCall(char *funcname, const TaskExpression& a1)
{
 return CURRENT_TASKGRAPH->tCall_internal(1, funcname, &a1);
}

__inline__ TaskExpression tCall(char *funcname, const TaskExpression& a1, const TaskExpression& a2)
{
 return CURRENT_TASKGRAPH->tCall_internal(2, funcname, &a1, &a2);
}

__inline__ TaskExpression tCall(char *funcname, const TaskExpression& a1, const TaskExpression& a2, const
TaskExpression& a3)
{
 return CURRENT_TASKGRAPH->tCall_internal(3, funcname, &a1, &a2, &a3);
}
```

The above explicitly defines tCall for up to 3 arguments. CURRENT_TASKGRAPH is a pointer to the current TaskGraph (which is an object that inherits from coreTaskGraph) since tCall_internal resides in the coreTaskGraph class as it is dependent on data within that class.

Once the interface was sorted, there were a few issues with the internals of making the function calls work correctly. For one, SUIF needed to know the type of not only the parameters in the call but also the parameters in the declaration of the function being called. SUIF needs this for a couple of reasons:

- SUIF may need to declare a prototype of the function, however, as users will generally be calling external functions contained in include files this functionality was not needed and thus this meant that a function needed to be added to the TaskIR class, namely, createFunctionUndeclared which will add a new function declaration variable for SUIFs sake (as this is a necessity before the function can be called) but will ensure SUIF does not actually declare it anywhere in the generated source code.

- SUIF may do some typecasting behind the scenes if C requires an explicit typecast for the parameter to be accepted without errors. However, since the type of the function definition is not known by TaskGraph, it would be preferable to be able to just pass the parameters as is with no typecasting but give the TaskGraph user the option of typecasting if they should deem it necessary.

Thus our solution is to simply lie to SUIF and tell it that the type of the parameters in the function declaration are the same as the type of the parameters in the call. This ensures that SUIF does no typecasting as we desired, although it does still allow the user to explicitly typecast using TaskGraph's cast<type>(expression); function, which will typecast the parameter to the appropriate type and then pass it through, with its new type, to the tCall function, just as desired.

The final issue which we encountered was in the way TaskGraph and SUIF handle expressions. Originally, when function calls or standard assignment statements were generated these were gradually built up as an expression, e.g. using the '+' operator would combine two expressions (which could just be variables, for instance) to generate another expression. However, this would not actually add the statement into SUIF's AST (Abstract Syntax Tree), thus, it would not appear in the generated source code. In order for it to appear in the generated source code an assignment '=' operator would have to be used and this would generate the new expression and declare the whole statement as a SUIF AST node. This example does not directly apply to function calls, of course, however, it introduces a problem. Suppose, when generating the function call, it does not know whether it is going to be just called on its own or assigned to another variable (or passed to another function even) then how does it know whether to add itself to SUIF as an actual AST node (as needed in the former case) or simply generate the new expression and return it (as needed in the latter case)?

Our solution to this problem is to create the expression for the function call twice, one expression will simply be returned, the other will generate the SUIF AST node. It is necessary to create the expression twice since SUIF did not seem to like assigning the same expression to different things (i.e. assigning it to another expression and a new AST node). Once this has been done then if the returned expression does get used with some TaskGraph function that generates a SUIF AST node (e.g. assignment operator/tCall function/for loop/while loop/...) then TaskGraph knows that it was wrong to generate the SUIF AST node and so it can then remove this node from the tree within that TaskGraph function.

In theory, SUIF was supposed to have support for removing nodes from the tree, however, it appeared that there was a bug in SUIF which prevented this from working correctly. It seemed that pointer addresses to the node to be removed would mysteriously change when passing them through to SUIF's remove function. It was found that this was possibly due to the unusual class inheritance mechanism used which appeared to do some typecasting behind the scenes that possibly results in the pointer address changing. Due to the difficulty involved with solving this we decided to simply create a remove_end function which simply removes the last node added (thus, avoiding the requirement for passing a pointer to the node to SUIF) as this is sufficient for our needs.

So, whenever a TaskGraph function is entered that generates a SUIF AST node the function needs to be able to check whether the TaskExpression that has been passed to it has generated SUIF AST nodes for any tCalls that are within it, so it can then delete them as necessary (using SUIF's remove_end() function). This is done by assigning TaskExpression objects a special tCall_flag which is an integer. tCall_flag is then manipulated as follows:

- For TaskExpressions generated by tCall it is set to 1.
- For TaskExpressions not generated by tCall and not generated from another TaskExpression, it is set it to 0.
- For any unary operators that do not generate SUIF AST nodes this flag is simply copied into the new TaskExpression generated by the operator.
- For binary operators, the sum of the tCall_flags in both expressions is added together and put into the new expression. This is because there could be a tCall in both expressions, thus two or more SUIF AST nodes would have been generated.
- Finally, when we reach a function that actually generates a SUIF AST node we repeatedly remove the last AST node tCall_flag times as this will represent the number of tCalls that occurred in the current expression.

This method then ensures that tCalls can appear both on their own and in another expression without generating duplicate tCall statements.

4.2.4 tInclude

In order to implement a tInclude function, we needed to modify SUIF so that it would add the necessary #include<"foo.h"> lines to the top of the source file, where foo.h, is of course, a user-defined header file.

It was felt that the easiest way to do this was to store a header_string in TaskIR, which would consist of all the various #include lines and then pass this to s2c_main (which is the SUIF function that generates the actual source code). We then modified s2c_main to accept this additional parameter and print the header_string to the top of the output file. Thus, all that the tInclude function needs to do is add the #include<foo.h> line, along with the new line character, to TaskIR's header_string.

4.2.5 Syntactically Safe Method of TaskGraph Function Calls

In order to allow one TaskGraph to call another TaskGraph in a syntactically safe way all that needed to be done was to create yet another set of overloaded versions of our tCall function (see section 4.2.3). The only

difference between the version which calls external functions and the version that calls TaskGraphs is that the version which calls external functions has as its first parameter the function name, which is of type char*, whereas the version which calls TaskGraphs takes a TaskGraph as its first parameter. Thus, we simply copy all the exported tCall functions that were discussed in section 4.2.3 but with only two changes. Firstly, we change the function name parameter to be a parameter of type coreTaskGraph&, which is a reference to the TaskGraph we want to call. Secondly, we grab the actual function name string from the coreTaskGraph object which was passed as a parameter and then pass that to our internal tCall function.

4.2.6 Recursion

There were no major issues with implementing recursion since it pretty much came as a result of our other function call implementations above. However, it did require special testing as some minor bugs causing duplicate functions to be generated occurred when using recursion.

4.2.7 Resolving Function Call Dependencies

There needs to be a way of resolving function call dependencies (see section 3.2.7) if TaskGraphs are to be allowed to call other TaskGraphs so as to ensure that the appropriate TaskGraphs are compiled together. There are essentially two main stages here:

- In the first stage it is necessary to somehow build a list of all the TaskGraphs that the calling TaskGraph, T1, is directly dependent on (that is, TaskGraphs which it calls itself).

- In the second stage, it is necessary to find all the TaskGraphs which T1 is indirectly dependent on (that is, for instance, TaskGraphs that are called by TaskGraphs that T1 calls). This essentially involves recursively scanning through the TaskGraphs and should become clearer in our pseudo code later on.

Firstly, let us discuss how the first stage is implemented. Each time a TaskGraph T1 makes a tCall statement to TaskGraph T2 then a reference is stored to T2 in T1, thus by the end of the TaskGraph definition a complete dependency list will have been built.

Our chosen data structure for storing these references is an STL (Standard Template Library) set structure. STL data structures are used because they are more convenient to use than going through the effort of coding the data structure manually. The advantage of a set data structure over, say, a list is that it does not store duplicate elements. Suppose the TaskGraph T1 makes two or more tCalls to the same TaskGraph T2 then we do not wish to store a reference to T2 twice as this could cause complications later on. Thus, the set data structure makes perfect sense.

Once the first stage is implemented, a discussion of the second stage is necessary. Essentially the second stage is done at compile time once all dependencies have been generated. This could have been done earlier on, however it makes no significant difference either way. The idea is that the code iterates through the list of dependencies and then recursively scans through each TaskGraph's dependencies in the dependency list.

This is best illustrated with the following pseudo code:

```
buildDependencyList(coreTaskGraph* T1, STL_SET &fullDependencyList)
{
  for (T2 = each taskgraph in T1.dependencyList)
  {
    Add TaskGraph T2 To fullDependencyList (provided it isn't already there)
   if (T2 was not already in the fullDependencyList)
   {
     buildDependencyList(T2, fullDependencyList);
   }
  }
}
```

In the above pseudo code, dependencyList is the dependency list as generated in the first stage. fullDependencyList is the dependency list that will result from the second stage. The code iterates through all the TaskGraphs in the dependency list and adds them to the fullDependencyList, if the fullDependencyList did not already have T2 then we recursively iterate through T2's dependency list and thus store all of its dependencies in fullDependencyList.

This completes the generation of the TaskGraph's function dependencies. When the compilation is done it then adds all the dependencies into a TaskGraphGroup class (which already existed prior to this extension) which will then put all the TaskGraph's into the same source file and compile them together.

4.2.8 Compiler Options Passing

This was fairly straightforward to implement as TaskGraph internally already had mechanisms for passing arguments (since this was needed to specify things such as object and source filenames and tell the compiler to link as a DLL). Thus, all that was needed was to ensure that the options were passed through to the appropriate compiler and then concatenate the user defined argument list with the default argument list (since TaskGraph must internally specify some options itself, such as object and source filenames).

4.2.9 Indeterministic Control Flow Instructions

TaskGraph's control flow instructions are all macros in TaskConstructs.h. So, we implemented our new variations of these by copying the various macros corresponding to the tFor, tForStep, tForDown, tForDownStep, tWhile, tIf and renaming them to tForMaybe, tForStepMaybe, tForDownMaybe, tForDownStepMaybe, tWhileMaybe, tIfMaybe. Each macro is very similar and was thus modified in a very similar way as well. Observe the tFor macro:

```
#define tFor(var,from,to) for ( tg::BlockEnder<tg::FinishBlockEnder> __ender(CURRENT_TASKGRAPH-
>startForBlock ( var, from, to, 1, opLESSTHANOREQUALTO ) ); __ender.hasDone(); __ender.done() )
```

In fact each macro, including tIf and tWhile use a for loop inside the macro. The for loop has nothing to do with the actual construct but rather it enables one to use braces in the normal way and allows it to make a function call to tell TaskGraph when it has finished constructing the tFor loop block and thus can then return to the original scope when it encounters the closing brace. This works since the for loop here will call __ender.done() when it completes the first iteration, which is what we want, it then calls __ender.hasDone() which will return false causing the loop to exit so the for loop will only ever do one iteration.

The next step was to modify this to accept our special parameter, which we will call createStatement, so that it will only add the construct if createStatement is true. The key thing to observe is that at the start of the for loop it creates a variable __ender of type tg::BlockEnder. This needs to be prevented from doing anything if the parameter is false, it is also necessary to prevent __ender.done() from doing anything since obviously it shouldn't finish creating a TaskGraph control flow block that hasn't actually been created. Furthermore, due to the limitations of C++, it is not possible to add the check inside of the for loop.

So, in order to implement the modifications it was necessary to modify the class BlockEnder's constructor and its done() function, it was also necessary to modify the function startForBlock inside of the coreTaskGraph class (as this was also called in the loop above, and should be prevented from doing anything). This was done by simply passing the createStatement parameter through from our macro to both BlockEnder's constructor and the startForBlock function. BlockEnder's constructor is then modified so that if createStatement is false then none of the code inside it is executed, except for the one line which ensures that __ender.hasDone() will return true so that the contents of the loop are executed once. A similar thing is also done to startForBlock except it has some code in it to remove tCalls as per one of our previous modifications (see section 4.2.3) and clearly that code must always be executed otherwise we will end up with tCalls all over the place, however, the rest of the code in that function is only executed if createStatement is true.

This essentially completes our implementation for the Maybe macros. As was said before, all the macros are essentially modified in a very similar way, so we will not discuss how exactly the other macros were modified. However, for completeness, we show our final resulting macro for the tForMaybe after all other modifications

have taken place:

```
#define tForMaybe(var,from,to,createStatement) for ( tg::BlockEnder<tg::FinishBlockEnder>
__ender(CURRENT_TASKGRAPH->startForBlock ( var, from, to, 1, opLESSTHANOREQUALTO,createStatement ),
createStatement ); __ender.hasDone(); __ender.done() )
```

4.2.10 Structure Bugs

During testing we found a couple of bugs in the structure code which needed fixing:

- The first prevented the same structure from being used in two TaskGraphs, causing the program to crash if the user did this. This issue was kindly resolved by Michael Mellor.
- The second was caused by SUIF adding "int : 32;" lines to its structure declarations in the generated code, this appears to be some obscure C syntax to do with padding. This meant that, since the version of the structure held by the C++ generator code did not have this padding then when the structure from the C++ generator code was passed to the TaskGraph generated code then the data in the structure became mangled as they were essentially different structures. This issue was resolved by locating the appropriate line in SUIF which prints "int : 32;" and commenting it out as there appeared to be no other way to fix it other than hacking SUIF.

4.3 Cuda Specific TaskGraph extensions

4.3.1 NVidia Cuda Compiler

Conveniently, TaskGraph already had a reasonably modular architecture for supporting different compilers since it already supported several compilers. Hence, in order to add support for the NVidia Cuda Compiler, essentially the code for calling one of the already existing compilers just needed to be duplicated and modified to invoke the Cuda compiler correctly. It was also necessary to create a NVCudaCompiler class and add it to TaskCompilers.cc, then a NVCC enumerator needed to be created and the compile function in TaskGraph.cc needed to be modified in order to allow the user to tell TaskGraph to use the Cuda compiler (see section 3.3.1 for details of how this is done). It was also necessary to modify the configure scripts to add an NVCC_PATH constant which stores the path to the nvcc executable. The biggest problem was mainly in finding the correct default options to get Cuda to compile as a DLL since there was no clear documentation on this other than the command "nvcc --help". However, by using a certain amount of trial and error we eventually got it to function correctly.

4.3.2 Cuda Kernel and Host TaskGraphs

As mentioned in section 3.3.2 Cuda kernel and host TaskGraphs are declared using cudahosttaskgraph and cudakerneltaskgraph. TaskGraph of course already had a taskgraph macro for normal TaskGraphs, so we just copied this macro and renamed it to cudahosttaskgraph and cudakerneltaskgraph. This macro then ensures that a flag, isCudaCode and/or isCudaKernel, is stored in with the TaskGraph. These can then be used to determine whether or not to generate normal C sourcecode, Cuda sourcecode or Cuda Kernel sourcecode. There are differences between how TaskGraph should generate normal C sourcecode and Cuda host sourcecode, which are detailed in the sections below. For now, however, we shall look at the key difference between Cuda kernel sourcecode and Cuda host sourcecode: *__global__ function type qualifiers*

The __global__ function type qualifier must be included in all Cuda kernel function declarations to tell Cuda that the function is a kernel function as opposed to a host function. This posed a challenge in that SUIF had to be modified to support the __global__ function type qualifier. This was fixed by modifying SUIF's proc_sym class to add a function set_cuda_qualifiers(int qualifiers) which takes a set of qualifier flags and stores them in a variable inside the class. Then, TaskIR is modified so that when it creates the function definition it simply calls procSym->set_cuda_qualifiers(CUDA_GLOBAL). This will then set the flag indicating that it is a Cuda

global function. All that was then needed was to locate where the string for the function definition was actually output in SUIF and prepend it with "__global__ " if the CUDA_GLOBAL flag was enabled for the qualifiers in procSym.

4.3.3 Cuda Global Calls

In order to call Cuda kernel functions it was necessary to add support for Cuda global calls. The final design for this is detailed in section 3.3.2. The implementation for these was carried out by using the code for the implementation of standard TaskGraph function calls detailed in sections 4.2.3 and 4.2.5 but extending it to support the extra features.

The first, relatively simple issue, is that the first three parameters must be the "special" Cuda global call parameters (as discussed in 3.3.2). Thus each tCudaGlobalCall function is made to always require the extra three parameters at the start of the parameter list.

The second issue is that these parameters must be enclosed in angle brackets. This required special modifications to the SUIF backend to ensure that it would generate the correct code with the angle brackets. This was done by making the tCudaGlobalCall function tell TaskIR that it is creating a Cuda global call when it calls TaskIR to create the function call, TaskIR would then in turn call a set_cuda_qualifiers(int qualifiers) function (a similar approach to what we did in section 4.3.2) on the in_cal class (as this is the SUIF class which stores the function call), passing the CUDA_GLOBAL flag through to it. This then required locating the code which generates the string representation of the function call in order to modify it to use angle brackets. Unfortunately, this was not quite as simple as one would hope as it turns out SUIF would generate a ctree class node from the in_cal class and the code to actually generate the string would be in ctree which has no reference to the information held in in_cal. Thus, it was necessary to ensure that when the ctree node is generated from in_cal it copies the cuda qualifiers information across into the ctree class. Then, when the ctree class prints the string for the function call a check is added to see if the CUDA_GLOBAL flag is set, if it is, then the first three arguments in the parameter list will be put into angle brackets and the rest of the parameters will be inside the normal brackets. If it is not, then the function call will be printed in the same way as before.

4.3.4 Cuda Built-in Structure Support

TaskGraph already had support for structures, so it was only necessary to make sure the implementation pre-declares the structures so that the user does not need to explicitly declare structures that should be built-in. It is also important to ensure that SUIF does not generate structure declaration code in its output as this would cause conflicts with Cuda's built-in structures (infact, it seems, Cuda may allow one to override these, but the effects of doing such a thing would only cause problems).

The structures are pre-declared by adding a simple include file, TaskCudaTypes.h, which declares all the structures needed for Cuda using the standard TaskGraph method of declaring structures. Each structure takes the same name as it would in Cuda as there should be no naming conflicts (since C++ is unaware of Cuda data types).

The rather more difficult issue is that it was necessary to stop SUIF from declaring the Cuda structures in the generated code. This was done by modifying the createStructType function in TaskIR so that when the structure is actually created it does a string compare to check if the structure's name matches the name of a Cuda structure, if it does, then it passes an additional parameter when instantiating SUIF's struct_type class indicating that it is a Cuda structure and thus SUIF had to be modified to accept this additional parameter and only output the structure declaration if it is not present. This last part was handled in a similar manner to our previous modifications to SUIF - we simply needed to ensure that it stored this flag in the struct_type class and when the structure was printed to disk the code would check if this flag is set, if it is not, then it prints the structure declaration to disk, otherwise it does not. It was also necessary to prevent SUIF from printing the "struct " keyword in front of variable declarations as this is a C standard which Cuda did not seem to like for its built-in structures. This was done by adding another check for the flag but in a different place.

Clearly, our method of getting TaskIR to determine whether the structure is a Cuda structure or not is not

ideal as string comparisons are a fairly poor way to handle the problem. However, it would have been fairly awkward to modify the code base so that string comparisons do not have to be made, as it would likely require much of the structures code to be duplicated and modified so as to create a special Cuda structure version of the code. Thus, since the intention of this project is to investigate the possibilities that exist for runtime compilation with Cuda rather than write perfect code it was decided that there was no strong reason to justify the extra time in trying to offer what is likely to be only a small performance boost when generating a large number of structures.

As one final point, an issue that did arise is that when structures are declared in TaskGraph it has both a real structure name and an internal TaskGraph structure name. Unfortunately, the structures code would give the structure its TaskGraph name in the generated code, rather than its real name. Since these two names are different this would mean that a user might use float4 in the C++ generator code and then tgFloat4 would appear in the SUIF generated code or vice versa. This is obviously not desirable. So a very minor change was made to the structures code which made it give SUIF the real name, so that float4 would be the name used in both the generated code and the C++ generator code. This change does in fact affect all TaskGraph structures, not just Cuda structures, however, it seems perfectly sensible as it does not affect whether other structures function properly or not.

4.3.5 Cuda Built-in Variable Support

Cuda built-in variable support is done by using the built-in structure support covered in the previous section (as Cuda built-in variables are all based on Cuda built-in structures) along with additional code to ensure that the built-in variables are pre-declared, so that the TaskGraph user need not declare them (and also cannot declare them incorrectly) and that SUIF does not explicitly declare the built-in variables in its generated output code.

Our implementation works by adding some extra defines in the cudakerneltaskgraph macro (which were discussed in section 4.3.2). Note, however, that the defines are not put in the cudahosttaskgraph macro since only Cuda kernels can use the built-in variables, thus, in the interests of type safety it is important *not* to add the defines to the cudahosttaskgraph macro. As was discussed in section 3.3.4, there are four Cuda built-in variables. It was decided to use a modified version of TaskGraphs standard macro (tVar) for declaring these built-in variables. The modified version of the macro is then called tVarUndeclared and the only difference between this and the standard TaskGraph macro is that it creates the variable but it will not declare it in the generated sourcecode, thus allowing the variable to be used but not declared which is exactly what is required.

This is done by ensuring that tVarUndeclared passes through an extra "declared" parameter through the various functions it calls and this ultimately gets passed to TaskIR's addVariableDefinition function which adds the variable definition to SUIF. This function then registers the variable with SUIF, but SUIF is not told to declare it in the sourcecode, thus making SUIF behave in the desired way. Fortunately this meant that on this occasion there was no need to modify the SUIF backend as SUIF appeared to be quite happy to behave in the desired way.

So all that was now needed was to declare the four Cuda built-in variables in our cudakerneltaskgraph macro using tVarUndeclared, with the relevant structures, and then these were ready for any TaskGraph user to use in their Cuda kernels.

4.3.6 Cuda Variable Type Qualifiers __shared__ and __constant__

As discussed in section 3.3.5, __shared__ and __constant__ variables are to be implemented by creating four new TaskGraph macros tVarShared and tVarConstant based on tVar along with tSharedArrayFromList and tConstantArrayFromList based on tArrayFromList (with the latter two allowing the flexibility of being able to declare arrays of a variable size).

The implementation for all four involves passing some special flags around which eventually end up being passed to addVariableDefinition in TaskIR. From this, a set_cuda_qualifiers function is called in the SUIF class var_sym and the CUDA_SHARED or CUDA_CONSTANT flags are passed through to it. var_sym is

then modified to store these qualifiers and the SUIF code that actually prints the variable declaration to disk prints either __shared__ or __constant__ before the rest of the declaration, depending on which, if either, flag is set.

In the case of tVarConstant and tConstantArrayFromList it is necessary to do a little more work than this. Recall (see section 3.3.5) that constant variables must be global variables rather than local variables and it is necessary for multiple TaskGraphs to use the same variable. Fortunately, SUIF provides some support for doing what is required as it is quite possible to tell SUIF to make the variable global instead of local. It is also possible to check whether a variable of the same name already exists in its entire database (which consists of the variables for all TaskGraphs, not just the current one) which makes it possible to ensure that, although a constant variable may be declared in multiple TaskGraphs, it will only be declared once.

Hence, for the most part, it was only necessary to modify the way TaskIR called the SUIF functions in addVariableDefinition when the CUDA_CONSTANT flag was passed to get the desired behaviour. There was, however, one small issue in that SUIF would add an "extern " keyword in front of the constant variable declaration because it was being declared global. This caused problems, so in a similar way to above it was necessary to add in some extra code that checked if the var_sym's CUDA_CONSTANT flag was set then it would not print "extern " when generating the sourcecode for that variable declaration. This then allowed the generated code to be as desired.

4.3.7 Cuda Standards Compliance Issues

During our implementation attempts we found that there were several areas where Cuda was not wholly compliant to C standards or the documentation fairly ambiguous about certain issues, this meant that we had to make special provisions in how the code was generated for Cuda. The main problem areas were:

- Cuda does not seem to always implicitly typecast where C normally would. In particular, it was found that typecasting a pointer variable to another type of pointer variable threw up errors unless it was explicitly typecasted.

- Cuda functions do not by default export with C linkage and, infact, it was found that it is essential to explicitly force non-kernel functions to be exported with extern "C" for them to link correctly with C++ code.

- Some Cuda functions require special flag-like parameters to be passed, such as cudaMemcpyHostToDevice when calling the function cudaMemcpy. It seems as though these flag-like parameters are built-in as they do not appear to be defined in any header files.

In order to resolve the first two issues, it was necessary to modify SUIF to accept a special isCudaCode parameter when generating the code. This tells SUIF that it must generate code for Cuda.

The first issue was resolved by locating the part of the SUIF codebase which determines whether to add in a cast or not. This is done in the class ctree's fold_converts function. Then, it was necessary to identify which of the many checks in that function tells it to "fold" the typecast in the particular case that was causing the problem. It was important to ensure that only the right typecast was unfolded as unfolding some casts could themselves prevent the code compiling (for example, typecasting the lhs of an expression is a bad idea). Once the check had been identified, another simple check to see if it was generating Cuda code was added, if it was, then it would unfold, if it was not, then it would fold as normal. This is to ensure that TaskGraph's normal C code generation is unaffected.

This brings us onto the second issue, GCC would not happily link with a Cuda DLL without the Cuda host code being declared as extern "C". Hence, it was necessary to modify the way SUIF generated code for function declarations. Two checks therefore need to be added when generating the function declaration string. SUIF must check that isCudaCode is set and that the function's CUDA_GLOBAL flag is not set (recall from section 4.3.2 that we assign a CUDA_GLOBAL flag to proc_sym so as to indicate that a SUIF function is a kernel function) since we do not want kernel functions to be declared extern "C" (as they should never be called directly from C++ and our experiments indicate that giving them C style linkage causes problems). Provided these checks are true then SUIF will print extern "C" prior to the function declaration.

Finally, there was the issue of fairly obscure flag-like parameters which Cuda functions need to take as input but which are not actually declared anywhere. It was decided to enable TaskGraph to use these by using a fairly simple hack in which the tVarUndeclared macro was used (which was discussed in section 4.3.5) and it declared a variable of the flag-like parameters name with some arbitrary type - since the type information is unimportant in this instance mainly because no variable will actually be declared in the generated sourcecode. This was sufficient to allow us to pass these parameters through to Cuda functions. It was decided to add the declarations for these parameters into the cudahosttaskgraph and cudakerneltaskgraph macros so that users can use them without having to declare them themselves. Of course, due to the large number of Cuda functions it is inevitable that some of these may have been missed, however, they can easily be added to TaskGraph, either to its core codebase or to a program which uses the library. In our implementation we currently only have two of these implemented: cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost.

4.4 Conclusion

This section has provided an overview of all the challenges that were encountered in implementing the design discussed in section 3. Some issues were to be expected at the start while others proved more problematic (e.g. it was not expected that the C part of Cuda would fail to adhere to C standards). It is also worth briefly noting that while this section has implemented function calling and recursion, both of these things are *not* possible inside of Cuda kernels since Cuda kernels do not support recursion and function calling requires special care with using a __device__ attribute.

We have now fully discussed the features that the TaskGraph extension supports, its design and how it was implemented. In the next section the TaskGraph extension will be evaluated primarily through the use of some examples which have been written to use the new TaskGraph extension.

5 Evaluation

This section is about evaluating the TaskGraph extension. Our evaluation will take the form of developing four examples for TaskGraph which are intended to show off the various benefits provided by the extension, including three Cuda examples (matrix multiplication, separable convolution filter and a ray tracer) and one non-Cuda example (expressions interpreter).

5.1 Test Setup

All performance tests in this chapter were performed on a E6300 Core2Duo clocked at 2.8ghz with 2GB of DDR2 RAM running at 800mhz and an NVidia 8800GTS graphics card with 640MB of video memory. The operating system used was Open SuSE Linux 10.2 with NVidia graphics driver version 171.06. The Cuda Toolkit used (which includes the Cuda compiler, nvcc) was version 1.1 and the Cuda SDK was also version 1.1.

5.2 Generic Examples

This section discusses a generic example written for TaskGraph which does not use Cuda but illustrates how some of our generic TaskGraph extensions can help to improve the code of a non-Cuda TaskGraph example.

5.2.1 Expression Interpreter

This example is intended to demonstrate how our generic extensions can be beneficial to a non-Cuda example. More specifically, it will be shown how the new tCall method for calling a TaskGraph recursively can be used by discussing a slightly modified version of the expression interpreter which was included in the original TaskGraph distribution.

The original example illustrated how TaskGraph could be used to specialise an expression interpreter. Specifically, the example features a mini expression based language which is sufficient to allow a program which computes the Fibonacci numbers to be written. The example then illustrates that specialising the interpreter to that program yields a significant performance boost over a standard interpreter.

5.2.1.1 Implementation

Our modification removes a small amount of very ugly and confusing code and replaces it by one much nicer line of code. Consider the following code snippet from the original example:

```
typedef TaskGraph<int, int> codeIntToInt;

...

class SpecVisitor : public voidVisitor
{
public:

...

  virtual void visitApp (const App &exp)
  {
    // Evaluate the argument
    exp.arg_ -> accept(*this); // this visitor delivers result to dest_

    // Find the taskgraph corresponding to this function
    cerr << "Fun is " << exp.fun_ << "\n";
```

```
    cerr << "TG is " << fenv_[ exp.fun_ ] << "\n";
    codeIntToInt *body = fenv_[ exp.fun_ ];

    // Now call the taskgraph via an indirect call to a C function in the
    // host code

    dest_ = tExecuteIntToInt( reinterpret_cast<pointer_size_t>(body), dest_ );
  }

...

}
```

Figure 5.1

visitApp is a function taken from the SpecVisitor class in the original example. This is because the interpreter operates as a visitor which walks over the expression tree. However, the exact details of the operation of the interpreter are not important to us. The key point is that this function generates TaskGraph code which calls another function, which, in the case of Fibonacci, happens to be itself. However, the real operation of this is not quite as simple since TaskGraph previously had no direct support for calling other TaskGraphs so it was done via an indirect routine tExecuteIntToInt which is passed both a pointer to the TaskGraph generated function that is to be called and it is also passed the parameter which is to be passed to that TaskGraph generated function. It also includes some very ugly typecasting.

To better understand how the tExecuteIntToInt routine works, the code is shown below:

```
/* First a little trickery to call a taskgraph from taskgraph-generated code
 */

static inline int appCallBackCPPIntToInt(pointer_size_t t, int a) {
  codeIntToInt *T = reinterpret_cast<codeIntToInt *>(t);
  //  cerr << "Callback: ";
  //  T->print();
  int res = T->execute(a);
  return res;
}
extern "C" int appCallbackIntToInt ( pointer_size_t T, int a ) {
  //  fprintf(stderr, "T = %d\n", T);
  int res = appCallBackCPPIntToInt(T, a);
  return res;
}
TaskExpression tExecuteIntToInt ( const TaskExpression &expr1,
                                  const TaskExpression &expr2)
{
  static TaskFunction2<int, pointer_size_t, int> func ( "appCallbackIntToInt" );
  return func.call ( expr1, expr2 );
}
```

Figure 5.2

The tExecuteIntToInt function uses the older function calling syntax discussed in section 3.2.3. It generates TaskGraph code which calls the function appCallbackIntToInt to which it passes the two parameters (pointer to function to be called and that function's actual parameter). However, appCallbackIntToInt is infact a callback function to the main C++ generator code as opposed to the generated TaskGraph code. This callback function, as also shown above, then takes these two parameters and using some additional nasty typecasting executes the TaskGraph function which it has been passed as the first parameter using the second parameter as the only input parameter to that function. It then returns this to the TaskGraph code that called it.

Obviously, this code is a hideous way of attempting to call a simple TaskGraph function. Our modification replaces all of the code shown above with just:

```
virtual void visitApp (const App &exp)
{
  // Evaluate the argument
  exp.arg_ -> accept(*this); // this visitor delivers result to dest_

  // Find the taskgraph corresponding to this function
  cerr << "Fun is " << exp.fun_ << "\n";
  cerr << "TG is " << fenv_[ exp.fun_ ] << "\n";
  codeIntToInt *body = fenv_[ exp.fun_ ];

  // Now call the taskgraph via an indirect call to a C function in the
  // host code

  dest_ = tCall(*body, dest_);
}
```

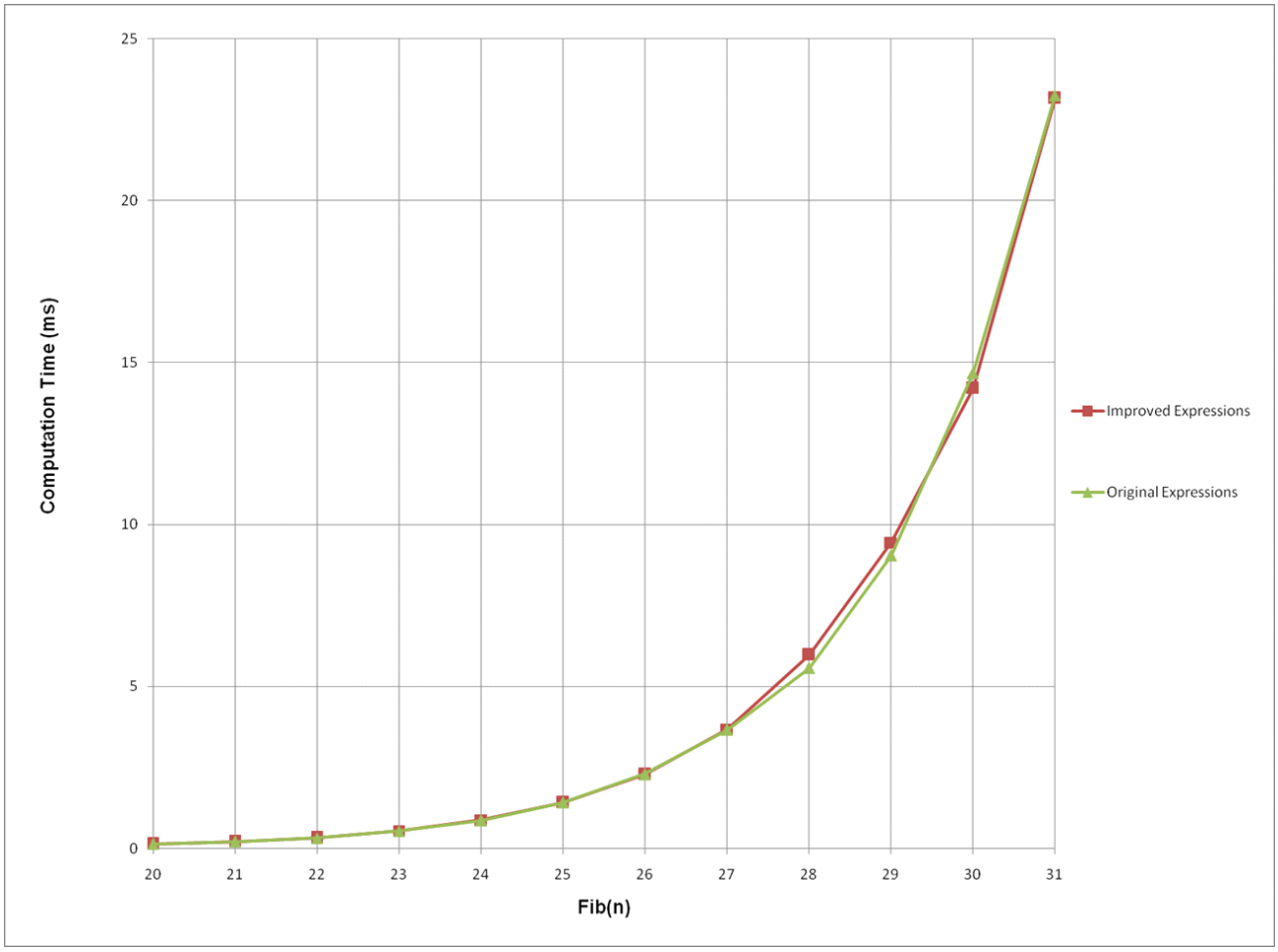
Figure 5.3

As can be seen, this is similar to Figure 5.1, except for the last line which is a simple tCall instruction telling TaskGraph to generate a function call to the TaskGraph "body" passing the parameter "dest_". This completely removes the need for the contents of Figure 5.2. It is much simpler, and much more elegant. It is even possible that it could, in theory, give performance improvements since it directly calls the TaskGraph function rather than calling a callback function which then calls the appropriate function.

Finally, note that as the complete sourcecode for this example is over 600 lines we have chosen not to include it in this report. The interested reader should refer to the submitted sourcecode for this project. However, rest assured that the only parts of the code in the original example which were changed are those that have been shown above.

5.2.1.2 Performance Tests

We now compare the performance of the original specialised expressions example and the new specialised expressions example. We compute the values of the nth Fibonacci number with n ranging from 20 to 31 in 1 step increments and then measure the time taken. For each test five runs are done and the result is averaged to try to reduce noise as much as possible. The final results are then shown in the following graph:



Expressions with Fibonacci performance times (lower is better)

As can be seen there appears to be no significant performance difference between the two versions. While there are some slight differences particularly between the two versions for higher values of n these can easily be put down to random variations particularly since an analysis of the different runs shows that there is a higher standard deviation between the results for higher values of n.

We can thus conclude that it appears that the performance overhead of using callbacks in the original example is very low and so while our improved example makes the code neater it, unfortunately, does not appear to improve the performance.

5.3 Cuda Examples

This section discusses the three TaskGraph Cuda examples that we have written including an overview of their algorithm design, relevant implementation details and an analyse of relevant performance results.

5.3.1 Matrix Multiplication

This is a simple example of matrix multiplication using Cuda and TaskGraph. The code for this is essentially a modified version of NVidia's matrix multiplication example which is discussed in the “NVidia Cuda Programming Guide” [8] that we have converted into TaskGraph Cuda code. This example is not intended to show any strong benefits that our TaskGraph extension provides but rather to be a relatively simple example which can be used both to test that the TaskGraph extension works and to demonstrate to potential users how to use the extension. It should also be noted that NVidia explicitly states that this particular example is not intended to be efficient but rather as simple as possible in order to teach the key concepts of Cuda.

5.3.1.1 Algorithm Design

This section will begin by briefly discussing the general algorithm used in this example, as described by NVidia [8]. The algorithm essentially takes two matrices, A and B, and produces a resulting matrix, C. It shares the workload between threads by splitting the matrix C up into equal sized square submatrices or "blocks" to be computed. Each thread block (see section 2.4.3) is then responsible for a specific block within the matrix and each thread is responsible for a specific element within each block. A and B are then also split up into blocks which are of the same size as those in C.

In order for each thread block to compute an element in C, Csub, it needs to access all the blocks along the corresponding rows in A and all the blocks along the corresponding columns in B. Thus, we can load the first two corresponding blocks from A and B, compute the product, Csub1, of these two blocks and then load the second two corresponding blocks and add each element of the matrix Csub2 to Csub1. This process is repeated until all blocks along the appropriate rows in A and columns in B have been computed, our resulting submatrix will then be Csub as required.

By choosing the number of threads within each thread block to be equal to the number of elements within each matrix block, each thread needs to load only one element of A and one element of B into shared memory and then it needs to compute only one element of Csub.

The reason for using the above loading scheme is that one can then reduce the number of accesses to global memory since each element within a block will only need to be loaded once, whereas when computing each element of C multiple accesses are required. Using NVidia's scheme these additional accesses will only be to shared memory, which is much faster.

5.3.1.2 Implementation

Our implementation is essentially just a case of converting NVidia's original Cuda code to TaskGraph Cuda code. The code in this particular example is for the most part a fairly straightforward syntactical conversion, however, there was one issue worth mentioning. NVidia's original code had a couple of statements of the form:

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
```

This is essentially calling a dim3 constructor and telling the compiler to create a dim3 structure (see section 2.4.3) with dim.x = BLOCK_SIZE and dim.y = BLOCK_SIZE. Unfortunately, TaskGraph does not support this syntax and it was decided that it was not worth implementing, mainly because the simple workaround for this is to instead write it out explicitly as:

```
tVar(dim3, dimBlock)
dimBlock.x = BLOCK_SIZE;
dimBlock.y = BLOCK_SIZE;
```

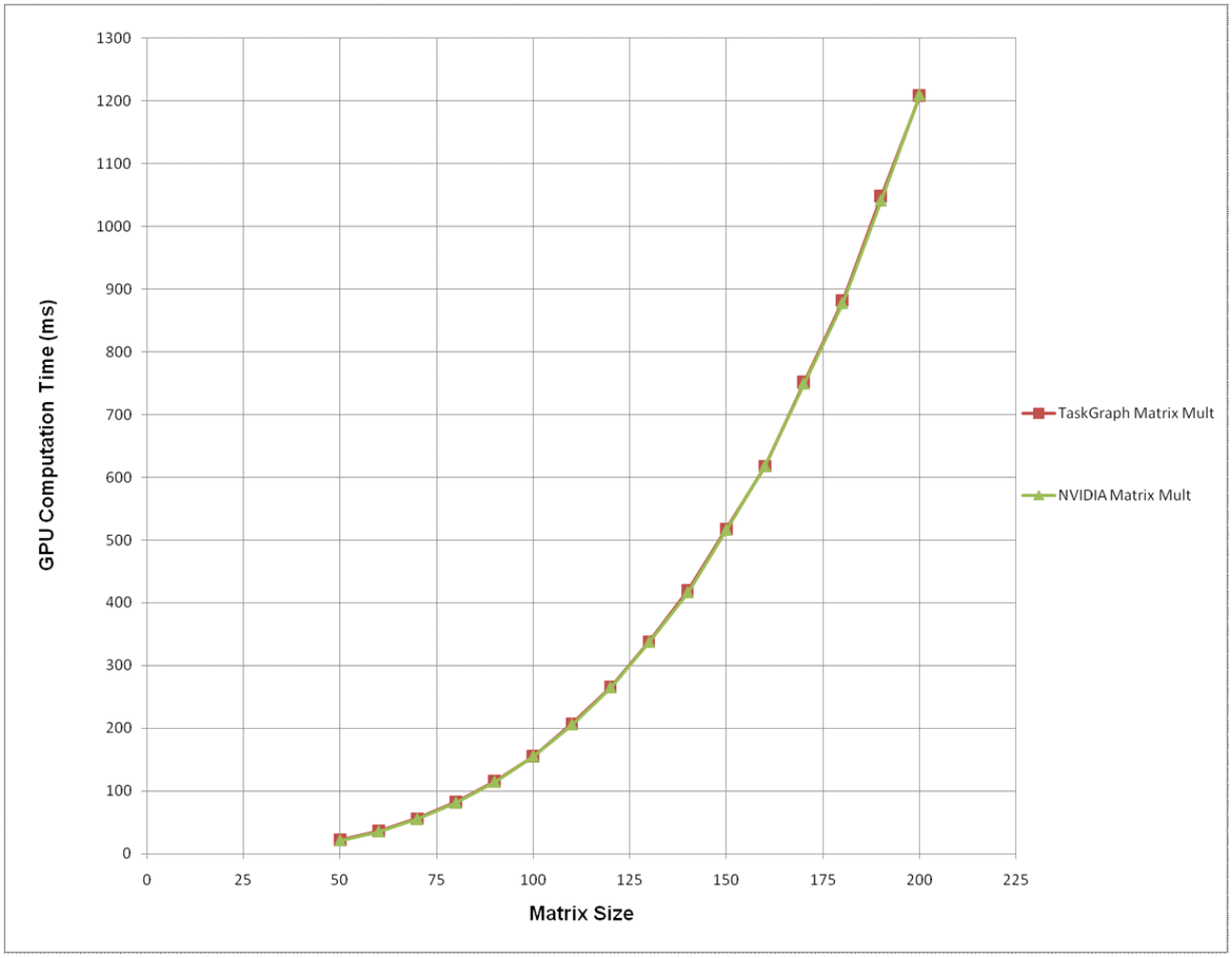
Complete sourcecode listings for this example are included in Appendix A.

5.3.1.3 Performance Tests

The performance of the matrix multiplication example for both the TaskGraph Cuda version and NVidia's original version is now to be compared. For the moment, only the times it takes to perform the actual computation are included, setup costs (i.e. compilation times in the case of the TaskGraph version, are excluded).

For the tests, the size of all dimensions of the matrices were set to be equal and they were gradually incremented by 10 each time, i.e. the first test was with A's width = 50, A's height = 50, B's width = 50 and B's height = 50, this was then increased to 60,60,60,60 respectively for the second test and so on up to a total size of 200 for each dimension. For each test three runs were performed, if one of the runs was substantially different from the other two runs then it was discarded and the run was repeated. The final time for each test was then the average of the three runs. This was done to ensure the accuracy of the results as unexpected events such as disk swapping, context switching etc could cause the results to fluctuate.

The results are then shown in the graph below:



Matrix Multiplication Computation Times (lower is better)

As can be seen, both versions appear to perform identically - there is no noticeable difference in performance between the two. This is what would be expected since once TaskGraph has generated the code it should be practically identical to NVidia's implementation. However, it was worth verifying this fact to ensure that there are no anomalies, for instance, issues such as the requirement for TaskGraph generated code to be dynamically linked rather than statically linked could affect performance.

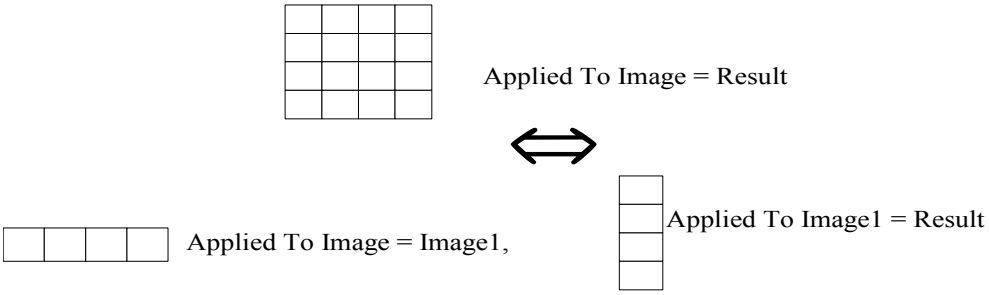
Of course, the real overhead for the TaskGraph version is the compilation time. It is worth noting that since the generated files are essentially the same for any matrix size asides from some minor changes to a few integer values, there is no reason to expect compilation times to vary for different matrix sizes. However, it was found that the compilation time can fluctuate somewhat between different runs of the same executable, especially compared to the GPU times where the fluctuations were generally quite small. So, if 10 runs were performed then the mean compilation time came to 877 ms with a standard deviation of 88 ms. This is in contrast to the GPU computation times which had a standard deviation of around only 1.5 ms.

The reason for these large fluctuations could be due to a variety of factors, the most likely of which is that compilation involves a certain amount of disk access which is likely to vary somewhat as the access times depend on where the head currently is on the drive and the current state of the filesystem (e.g. its fragmentation level). The other factor is that compilation is CPU dependent rather than GPU dependent and in general the CPU is likely to have a higher load with having to multi-task with multiple processes which may vary their usage of the CPU, whereas the GPU will have a more consistent load.

Clearly, the overhead of the compilation phase can be quite high and it can seen that even for the largest matrix size that was tested it increases the overall execution time by some 73%. Of course, given that no specialisation is used in this example, it would only need to be compiled once at the start of the application, thus, for applications which compute very very large matrices or which compute a large number of large matrices would not necessarily be heavily impacted by the compile time as it would then be nominal compared to the total computation time.

5.3.2 Separable Convolution Filter

This is a fairly complex example which has been taken from the NVidia Cuda SDK [30] and converted to TaskGraph Cuda. A convolution filter is a filter which turns each pixel into a weighted average of the surrounding pixels via means of a *kernel* (not to be confused with Cuda kernels) which specifies the weights of the surrounding pixels. This particular example applies a special type of convolution filter to an image, known as a separable convolution filter. A separable convolution filter is a filter that can be split into two separate passes such that each pass involves only a 1-dimensional convolution filter being applied to the image (usually both a row pass and a column pass). The reason why NVidia chose a separable convolution filter is that it is possible to optimise it to be particularly efficient with a Cuda implementation.



Example of a separable filter (i.e. a 2D convolution filter is equivalent to two passes with 1D filters)

As will be shown, this particular example can give a potential performance gain when written for TaskGraph Cuda as some of the loops may be unrolled in order to specialise for specific convolution kernel sizes. The loops may further be specialised to specific convolution filters providing particularly significant performance gains for filters which consist of a certain number of zeroes.

5.3.2.1 Algorithm Design

The main issues in the algorithm will be very briefly outlined here, however, the algorithm is rather complicated and the specific details are not directly relevant to us. We would therefore direct the interested reader to NVidia's own document explaining the algorithm [31].

The algorithm essentially works by splitting the image up into separate *blocks*. Each *thread block* is then responsible for a specific block within the image and each thread is then responsible for computing each element in the image.

The algorithm begins by loading the convolution kernel into constant memory since the same convolution kernel is used in all passes by all blocks, is never modified, and should be small enough to fit into the 64k of constant memory. This allows for very fast cached accesses to the kernel which is important as its data elements will be accessed very frequently.

However, there are issues with loading the necessary pixels from global memory into shared memory for each block in the actual image to which the kernel will be applied. Since elements towards the border of the block will require information from pixels outside the block (call them *apron* pixels) when applying the convolution kernel to that part of the image this needs to be taken into account when loading the pixels. If all the threads load a pixel each then all the threads which load the apron pixels will be idle during pixel computation, so a better method is to ensure that each thread loads multiple pixels.

Unfortunately ensuring that all the threads load the right pixels is not very easy and there will always be some idle threads. However, since the kernel is separable it means that one does not need to take into account the top and bottom apron pixels for the horizontal pass and similarly one does not need the left and right apron pixels for the vertical pass. As it turns out, this substantially improves the performance of the algorithm.

As one final point, an additional source of complexity is that for optimal performance the memory accesses must be coalesced. In other words, if all of the threads within a *warp* (i.e. the maximum number of threads

that can be simultaneously executed on the same multiprocessor) access memory elements consecutively and they fulfil certain half-warp alignment requirements then the GPU can coalesce these accesses and provide much better performance than if the memory accesses are random. This results in additional code which tries to ensure that these memory alignment requirements are met.

5.3.2.2 Implementation

The initial implementation was for the most part a fairly straightforward conversion aside from some minor quirks such as the dim3 constructor syntax discussing in section 5.1.2.

However, the significant part is that TaskGraph conveniently allows the code to be specialised to different kernel sizes by unrolling some of the for loops and it was found that this offers a noticeable performance gain. Of particular interest are the following two loops which appear in the original non-specialised TaskGraph code for this example:

From convolutionRowGPU_kernel:

```
tVar(int,k);
    tFor(k, -KERNEL_RADIUS, KERNEL_RADIUS)
      sum += data[smemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

From convolutionColumnGPU_kernel:

```
tVar(int,k);
    tFor(k, -KERNEL_RADIUS, KERNEL_RADIUS)
      sum += data[smemPos + IMUL(k, COLUMN_TILE_W)] * d_Kernel[KERNEL_RADIUS - k];
```

convolutionRowGPU_kernel is the kernel function that handles the row passes for the convolution filter, similarly convolutionColumnGPU_kernel is the kernel function which handles the column passes for the convolution filter.

The tFor line in both examples is a TaskGraph For loop and essentially tells TaskGraph to loop around the succeeding statement with k incrementing by 1 each time from the values -KERNEL_RADIUS to KERNEL_RADIUS where KERNEL_RADIUS is the radius of the kernel so that the width of the kernel must always be 2*KERNEL_RADIUS + 1 (this ensures that the kernel spans the same number of elements on both sides of a pixel, otherwise the algorithm could be even more complicated).

It is now possible to unroll these loops, this is done by simply changing the above code to the following:

From convolutionRowGPU_kernel:

```
    for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
      sum += data[smemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

From convolutionColumnGPU_kernel:

```
    for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
      sum += data[smemPos + k*COLUMN_TILE_W] * d_Kernel[KERNEL_RADIUS - k];
```

The TaskGraph tFor loops have been replaced with normal C++ for loops, this will mean that the succeeding line will be repeated in the TaskGraph sourcecode with k varying from -KERNEL_RADIUS to KERNEL_RADIUS. Furthermore, we replaced the IMUL instruction (which actually makes a tCall to __mul24 which is a Cuda 24-bit integer multiplication directive that is faster than 32-bit integer multiplication on Cuda hardware) with a compile time multiplication so as to ensure the multiplication is done at compile time as opposed to runtime.

NVidia themselves have a loop unrolling mechanism using templates in place for these two loops, if one sets the correct flag in the sourcecode. However, the advantage which we can offer over NVidia's mechanism is that in NVidia's case they would be stuck with only being able to use a specific kernel size, in our case we can generate the code depending on what is known about the convolution kernel at runtime. For example,

this could be used in say an image manipulation program where the user wishes to specify the size of their convolution kernel. The program can then runtime compile the convolution filtering code and execute it, this would be particularly useful for very large images/kernels or big batch processing jobs with the same convolution kernel as the performance gains from the loop unrolling would outweigh the losses from the initial compilation time.

As a further improvement, it is possible to specialise further to a specific filter as opposed to just a specific filter size, in which case our two loops above can become:

From convolutionRowGPU_kernel:

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
  sum += data[smemPos + k] * h_Kernel[KERNEL_RADIUS - k];
```

From convolutionColumnGPU_kernel:

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
  sum +=
    data[smemPos + k*COLUMN_TILE_W] *
    h_Kernel[KERNEL_RADIUS - k];
```

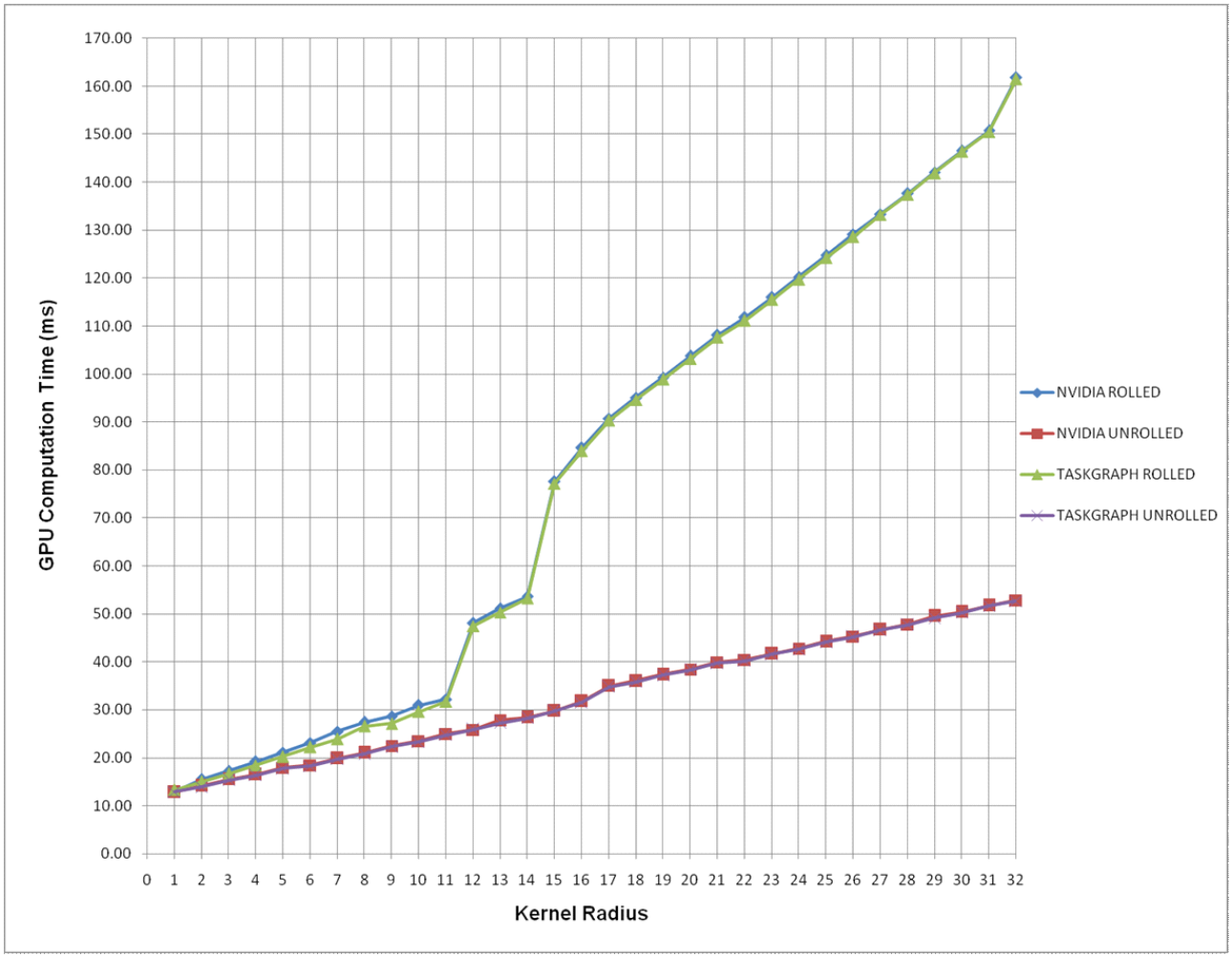
The variable d_Kernel has been changed to h_Kernel. d_Kernel is a TaskGraph reference to the kernel in constant memory, whereas h_Kernel is a normal C++ variable storing the kernel. Essentially, this code will insert the kernel values as compile time constants in the generated code, as opposed to loading them from constant memory while executing. This code could also offer a further advantage, if there are some zero entries in the kernel then in theory the compiler should just optimise these out, potentially resulting in a good additional speed boost.

Finally, note that as the sourcecode is over 500 lines we have chosen not to include it in this report. The interested reader should refer to the submitted sourcecode for this project.

5.3.2.3 Performance Tests

The performance is now compared between the original NVidia version of this example and our TaskGraph version. Results are compared for the relevant loops being both rolled, unrolled, specialised with zeroes and specialised without zeroes (as discussed above) in both versions. For the moment only the time it takes to perform the actual computation is included, setup costs (i.e. compilation times in the case of the TaskGraph version, are excluded).

For the first set of tests we tested four versions (NVidia rolled, NVidia unrolled, TaskGraph rolled, TaskGraph unrolled) as the kernel radius (recall, width of kernel = 2 * kernel_radius + 1) varied from 1 to 32 in increments of 1. In all cases a Gaussian kernel was applied to a 4096 x 4096 image. As before, for each test three runs were performed, if one of the runs was substantially different from the other two runs then it was discarded and the run was repeated. The final time for each test was then the average of the three runs. This was done to ensure the accuracy of the results as unexpected events such as disk swapping, context switching etc could cause the results to fluctuate. The resulting graph of our results is then shown below:



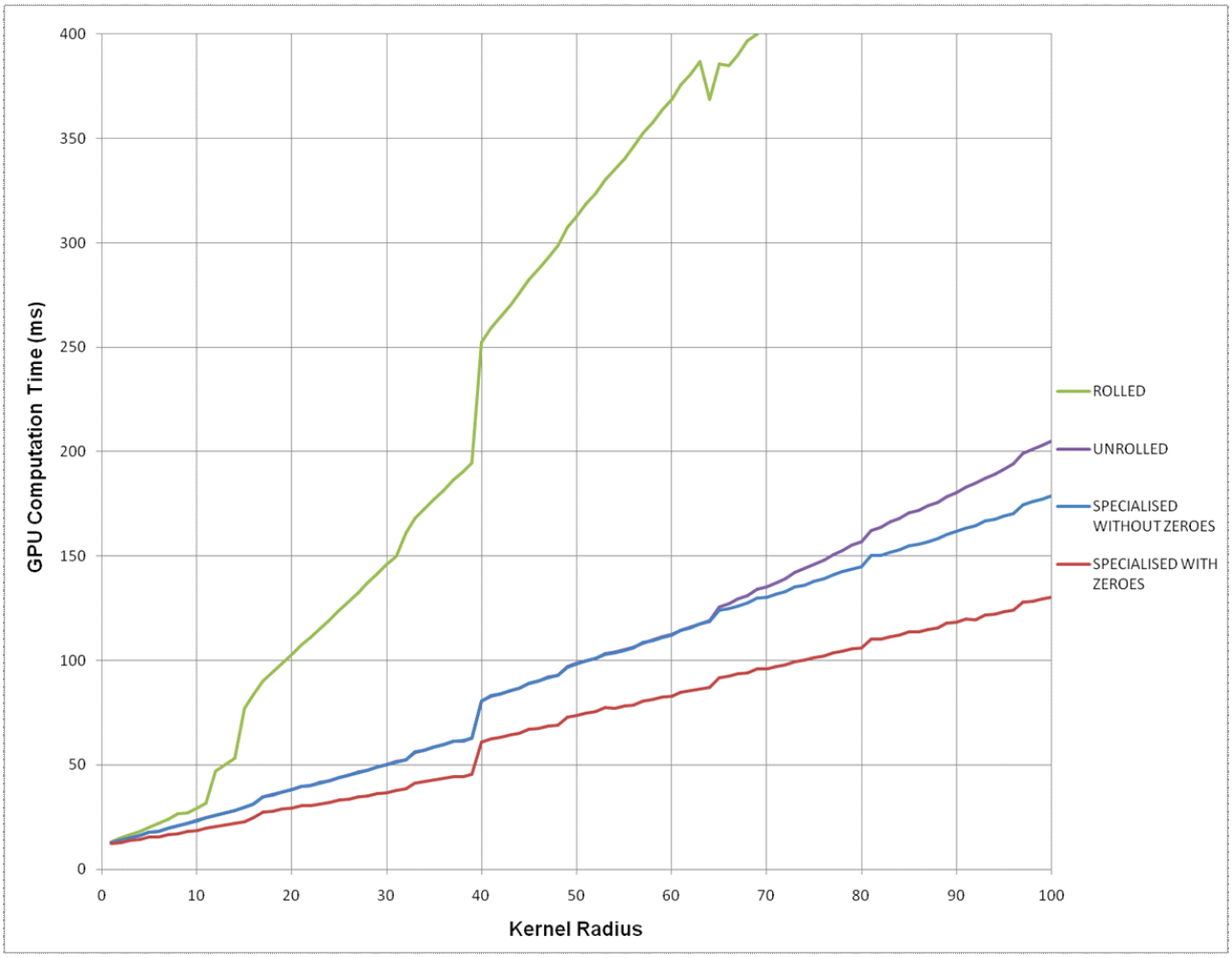
Separable Convolution Filter Performance Times (lower is better)

It can be seen that both TaskGraph unrolled and NVidia unrolled versions appear to have exactly the same performance, which would be expected. The NVidia rolled and TaskGraph rolled versions also have the same performance, which would also be expected.

Thus, having observed that there are no anomalies between the TaskGraph versions and the NVidia version it is now worth drawing attention to the fact that if the kernel size is not a constant then, as was mentioned earlier, in the NVidia case we must resort to rolled, however, in the TaskGraph case it is possible to still use unrolled. Comparing the performances of the two it can be seen that there is a fairly substantial performance improvement with the unrolled versions over the rolled versions which gets larger the greater the kernel radius.

Having determined that the rolled and unrolled TaskGraph version can match the performance of the NVidia version, the performance of the four different TaskGraph versions, rolled, unrolled, specialised to filter without zeroes and specialised to filter with zeroes are now compared. In this case the kernel radius is varied from 1 to 100 in increments of 1. Like above, in all cases the kernel was applied to a 4096 x 4096 image and for each test we did three runs, averaging the result. The kernel applied was a Gaussian kernel. However, in the case of specialising with zeroes, while this is the same algorithm as specialising without zeroes, a different kernel is used such that it is the Gaussian kernel but with all the even numbered elements being 0, i.e. 1st element is non-zero, 2nd element is zero, 3rd element non-zero, 4th element zero and so on. So approximately half the elements are 0.

The resulting graph is then shown below:

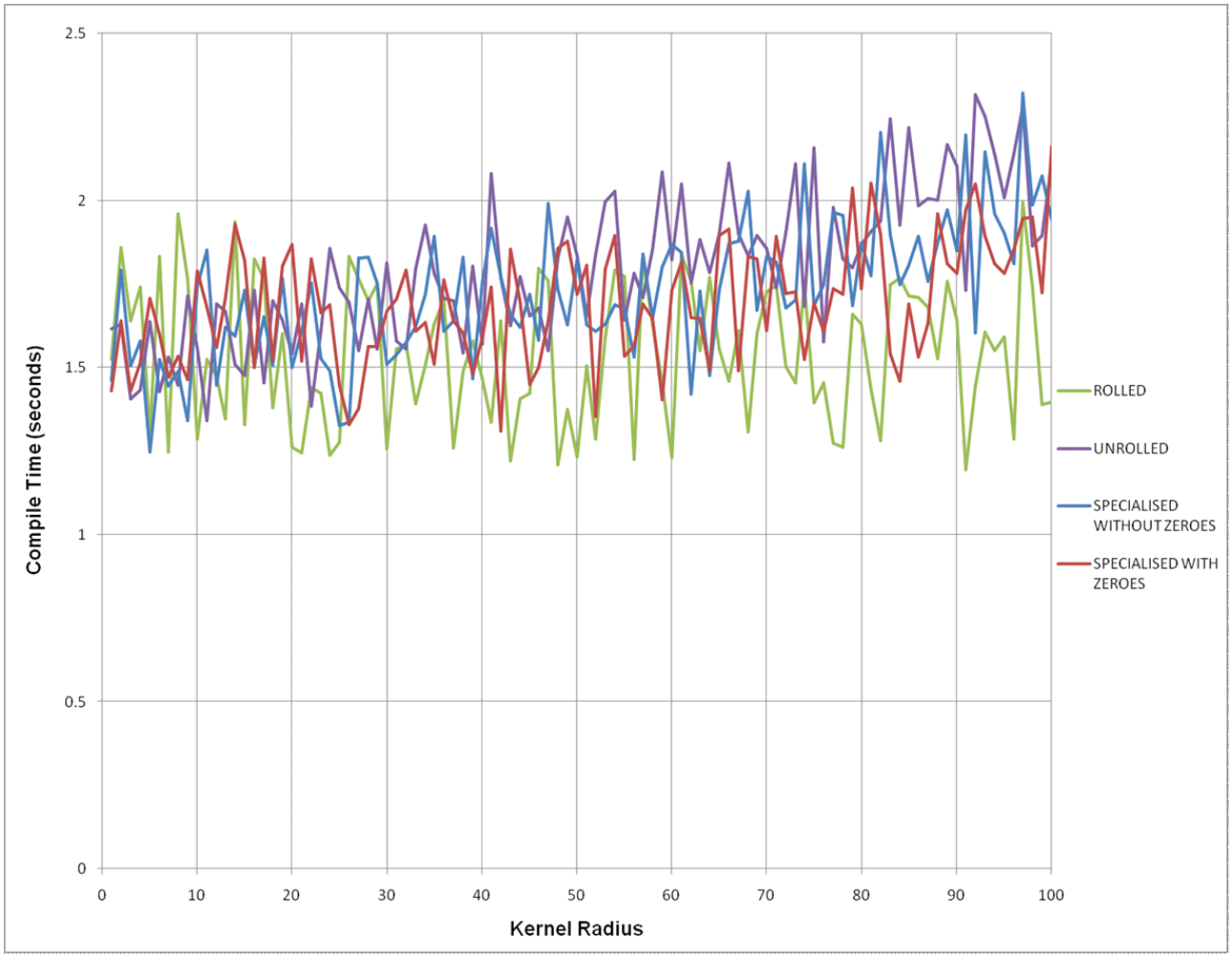


Separable Convolution Filter Performance Times (lower is better)

As can be seen, specialising without zeroes does not seem to provide a speed boost over unrolled up until a kernel radius of around 65, at which point it starts to perform marginally faster with the gap in performance growing with the radius size. Specialising with zeroes gives an additional speed boost over all the other versions for all kernel radius values. The gap in speed between this and specialised without zeroes also seems to grow as the kernel radius gets larger.

While it might seem odd to use a separable convolution filter with zeroes in it, it is possible that it could be used in cases where a very large kernel is to be applied but due to the high computational cost some zeroes are randomly placed in order to reduce the computation time at the expense of some accuracy.

Now it is worth examining the compilation costs, particularly since loop unrolling will increase the compilation times. The following graph illustrates these costs:

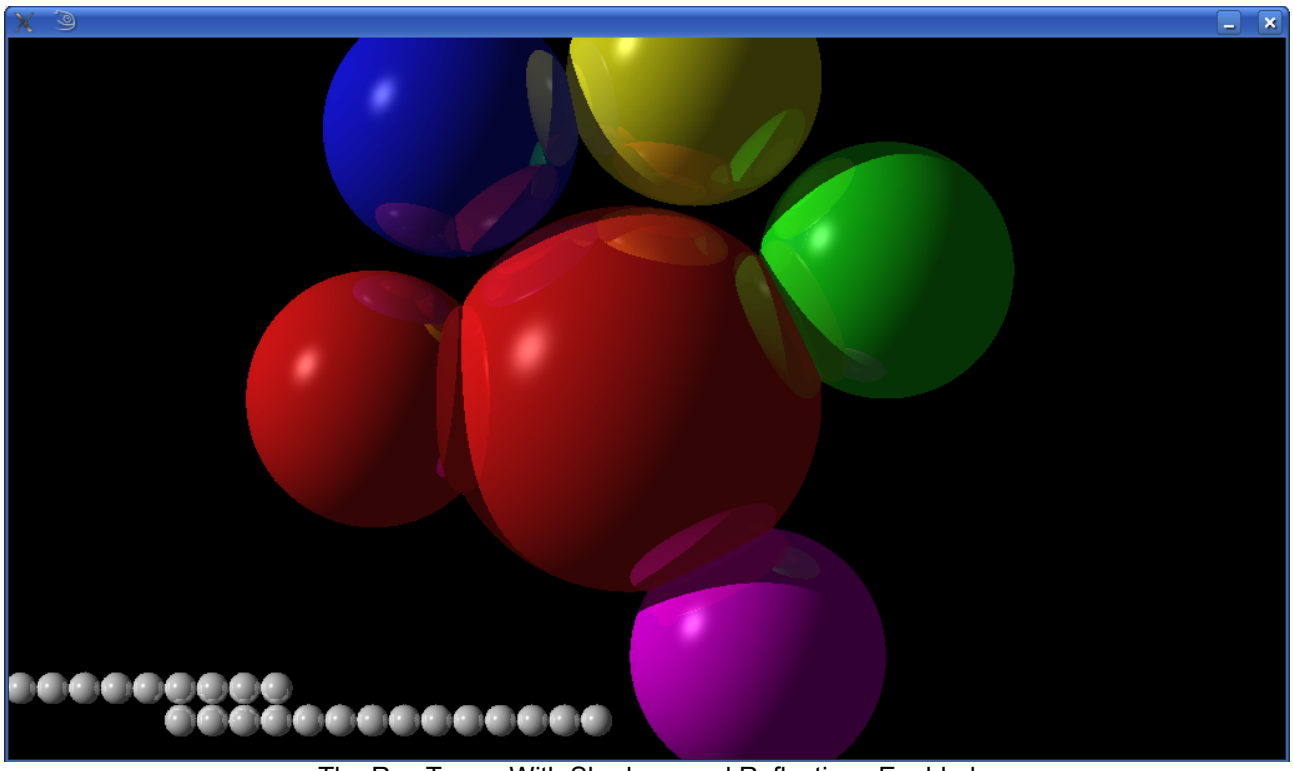


Separable Convolution Filter Compile Times (lower is better)

As can be seen and as also discussed in section 5.3.1.3 the compile times tend to have quite a high standard deviation. However, our key point of interest is whether the loop unrolling increases the compile times. It can be seen that for the three unrolled versions the compile times appear to rise to be slightly higher than the rolled version as the kernel radius gets very large. This increase is very slight however and random variations in compile time may well, in practice, have a greater influence than the loop unrolling.

5.3.3 Ray Tracing

With this example the construction of a real-time ray tracer is demonstrated. This is capable of specialising to the objects in the scene or both the objects and the eye position in the scene for additional performance benefits (the light positions may still move, in the latter case). The objects are currently only spheres, although in theory one could extend the code to handle polygons. It can handle both shadows and reflections, although all objects are assumed to be perfect mirrors and the reflectivity coefficient of each object is identical, for simplicity.



The Ray Tracer With Shadows and Reflections Enabled

5.3.3.1 Implementation

This example is loosely based on the source code to two different ray tracers:

- The TaskGraph ray tracer example which is included in the TaskGraph distribution and which specialises the scene to a specific set of objects and then renders it. It supports similar features to our implementation, it only handles spheres, supports shadows and perfect mirror reflectivity although it does allow different objects to have different reflectivity values. The example however was specifically written to run in a single thread on a CPU and thus needed to be modified to support Cuda. It was also coded to render one image and save it to disk since being CPU-based it was not written for real-time rendering.

- A Cuda ray tracer written by Eric Rollins [34]. This is a very basic ray tracer which only fires a primary ray and computes ambience, diffusion and specularity from a single light source. However, it provides a nice framework for the implementation of a Cuda ray tracer. Essentially it works by spawning one thread for each ray since each ray will operate independently of the other rays.

Essentially some code and ideas have been taken from both of these examples and then modified and added to in order to function as desired.

Due to some of the limitations both with Cuda and the TaskGraph Cuda extension whereby Cuda kernels do not support recursion and the TaskGraph Cuda extension does not support any function calling from kernels it was necessary to inline everything in the kernel code into the same function, which does cause a fair amount of code bloat, however it is really the only possible solution.

Essentially the work started off by using the Cuda ray tracer as a framework and then it was modified in several places to gain significant speed improvements over the original version. Two major modifications to the code which were carried out did not involve specialisation directly but affected memory transfers and improved the performance of the code :

- For the first, the original code used float4 yet its last component, w, was never used. For this reason we changed it to float3 and noticed a significant speed up, quite possibly because less data has to be transferred over the bus and more variables will be within a certain number of bytes of one another improving memory coalescence.

- The other modification was to use constant memory instead of shared memory for storing the object/eye information. The original Cuda ray tracer copied the object/eye information from the host to the device's global memory, the kernel would then load the information from the global memory to its shared memory. This double copying may incur some overhead, especially since only one thread can copy the data from global to shared memory while the other threads are idle. This is because the block sizes do not correspond to the number of objects, and sorting out which threads should load which objects (possibly several) is a complicated problem and its solution would in the very least involve modulo arithmetic which is extremely expensive on GPU hardware. The other issue is that due to the nature of a ray tracer the memory accesses tend to be fairly random and it is more difficult to avoid bank conflicts (see section 2.4.3) which impair performance with shared memory. However, constant memory does not suffer from bank conflict issues and the information only needs to be loaded from host to device once and then on a cache hit it can be as fast as reading from a register. This modification resulted in yet another fairly substantial performance gain.

The next modification was to replace some of the code for checking intersections between a ray and a sphere with the code from the TaskGraph ray tracer as the equations appeared to have been derived in a slightly different way. Swapping for the TaskGraph form of the equations did not really appear to affect the performance of the Cuda ray tracer by themselves, but their format did make specialisation easier to do and also made it easier to add things such as reflections. This was largely because the Cuda version had a PreTrace() function called on the CPU which pre-calculated some information on the CPU so the GPU didn't have to execute it, however, making the GPU execute it did not seem to affect the speed much. This did prove useful however since the pre-calculated information does actually specialise quite well and it is likely that by essentially putting some of the information in as constants rather than pre-computed variables the number of memory accesses are reduced. Of course, not all of the variables in those equations could be turned into constants but some of the values would have to be constantly recalculated anyway if reflections are to be implemented.

The chunk of code discussed just above was essentially the key code that could be specialised by simply unrolling the loop and then references to the object position, object radius and eye position could potentially be replaced by constant values. The following code snippet is provided to illustrate the final solution utilising specialisation:

```
 tVar(int,ob);
closestObj = -1;
tForMaybe(ob, 0, NUM_RENDERED_OBJECTS - 1, !isSpecialise) {
 for(int obj=0; obj < NUM_RENDERED_OBJECTS && (isSpecialise || obj == 0); obj++) {
  tIfMaybe(obj != reflected_from_object, (isSpecialise))
  {
   tIfMaybe(ob != reflected_from_object, !(isSpecialise))
   {
    if (SPECIALISE & EYE_OBJECT_SPECIALISE)  {
     V3Sub(g.eye, g2[obj].objectCenter, eye_object);
    }
    else
    {
     if (SPECIALISE & OBJECT_SPECIALISE) {
      V3Sub(rayPos, g2[obj].objectCenter, eye_object);
     } else {
      V3Sub(rayPos, sg2[ob].objectCenter, eye_object);
     }
    }

    firstPart = V3Dot(delta, eye_object);
    if (isSpecialise)
     plusMinusSquared = firstPart*firstPart - V3Dot(eye_object, eye_object) +
g2[obj].objectRadius*g2[obj].objectRadius;
    else
     plusMinusSquared = firstPart*firstPart - V3Dot(eye_object, eye_object) +
sg2[ob].objectRadius*sg2[ob].objectRadius;
    tIf (plusMinusSquared >= 0)
    {
```

```
            secondPart = tCall("sqrt",plusMinusSquared);
            s1 = -firstPart - secondPart;
            s2 = -firstPart + secondPart;
            tIf (s1 >= 0) {
            tIf (s1 < closestS) {closestS = s1; if (isSpecialise) closestObj = obj; else closestObj = ob; }
            } tElse tIf (s2 >= 0) {
            tIf (s2 < closestS) {closestS = s2; if (isSpecialise) closestObj = obj; else closestObj = ob; }
          }
        }
      }
    }
  }
```

This code may look fairly complicated at first, however, it is designed to allow one to specialise it in various different ways namely, to objects, to eye and objects or no specialisation. It does this by using our set of Maybe control flow instructions (see section 3.2.9). In some ways these make the code slightly harder to read than to just have a single if statement with three different cases for the different kinds of specialisation, but they make it easier to see what is different between the different specialisations and what is the same.

So, to walk through the code briefly, the top two for loops essentially decide whether to roll or unroll the loop based on the value of the isSpecialise boolean variable. Then inside the loop the code has to check against the two possible loop variables (ob or obj depending on which loop variable is active) to ensure that this is not a reflected ray from the object currently being checked (see our discussion on reflections below for why this is necessary).

The next chunk of code attempts to solve equations for the intersection of a ray with a sphere. There are essentially two sets of variables, g, g2 and sg,sg2. g,g2 are global C++ generator variables storing the eye/object information (thus its appearance in the code will force the generated code to include it as a constant value, set at generation time) while sg,sg2 are TaskGraph Cuda variables storing the eye/object information in constant memory. Thus, there are several checks to determine whether to use g2 or sg2 in the generated code depending on whether it is specialising or not and if so, what type of specialisation is being carried out.

If the code is observed closely, it should become clear how the specialisation is working. A brief point, however, is that one could be misled into thinking that the sg2[ob].objectRadius*sg2[ob].objectRadius is unnecessary as the squared radius of the sphere could have been pre-computed and simply left as a square. This is true, however, it seems that the real overhead is the memory access latency to the sg2[ob].objectRadius variable, thus, replacing it by a constant value provides a much greater performance gain than the gain given by simply removing a multiplication instruction.

This then brings us onto the implementation of shadows, reflections and multiple light sources:

Shadows were implemented by adding a second for loop which would check for intersections between the ray's intersection point and the light source in a similar way to the intersection loop discussed above. If there is an intersection then no light comes from that light source to that point.

For this loop, however, it was decided not to specialise it since specialising it seemed to have less affect on the overall performance and in some cases even decreased the performance. This is possibly because unrolling would overload the cache and also this loop tends to be executed less often probably because some primary rays may not intersect any object.

As a final point regarding shadows, it was necessary to add a special check to ensure that the current object being looped over is not the same object that was intersected by the primary ray. This is because otherwise it was found that the ray from the intersection point to the light source tends to intersect the object it is being emitted from! Another approach which is often used is to move the ray forward slightly. However, it was found that the ray needed to be moved forward quite a lot on Cuda hardware for it to function properly. This is possibly because it is not entirely floating point standard compliant. Either way, it was found that an if test is the most robust solution.

Reflections were implemented by simply looping over the rest of the code in the kernel but at the end of each iteration the reflected ray is computed and the original ray is replaced by this new reflected ray. There

were, however, similar issues to shadows with rays intersecting the object that they were emitted from so it was necessary to add another if test into the first intersection test for loop (i.e. the specialised loop originally designed to test for primary ray intersections, but which now also detects intersections for reflected rays).

Normally one would make recursive calls in order to compute reflected rays, however, this was not possible as kernels do not support recursion. Of course, any recursive function can be transformed to an iterative function which is roughly what has been done here. To simplify the algorithm slightly it is assumed that all objects have the same amount of reflectivity. Thus, using the original TaskGraph ray tracer code as an example a normal recursive ray tracing call for reflection would look like:

```
intensity += shootRay(&reflectedRay, depth + 1) * closestObject->finish->reflectionColour;
```

However, by expanding this recursion it can be seen that if all objects have the same reflectivity then we end up with a formula that looks like:

```
final_intensity = reflectless_intensity[maxdepth]*reflectivity^maxdepth + reflectless_intensity[maxdepth -
    1]*reflectivity^(maxdepth - 1) + ... + reflectless_intensity[1]
```

So this allows us to just maintain a variable final_intensity and keep adding the corresponding intensity values at each iteration depth.

Multiple light sources were implemented by simply adding a loop to loop over the various different light sources when shadows and light intensities are computed.

Strangely, it was found that when eye and object specialisation were enabled and there were two light sources the program would crash. However, if the light source loop was unrolled or it was not specialised to the eye position then it would function perfectly fine. We suspect that this could be a compiler bug in NVidia Cuda as we had already discovered a bug earlier on in development when using the 1.0 compiler which had a similar glitch that causes it to work fine when a loop was rolled but unrolling it would cause the program to crash. In that case, switching to version 1.1 fixed the problem, however, in this case, despite using version 1.1 there still seems to be an issue.

5.3.3.2 Performance Tests

The performance testing involved testing some five different variations of the ray tracing algorithm using 1 to 100 objects (in increments of 1), a resolution of 1024x576, one light source and shadows enabled. The five variations were: No specialisation (with no reflections), Object specialisation (without reflections), Eye and Object Specialisation (without reflections), Reflections without specialisation, Reflections with object specialisation. In the cases with reflections the maximum ray tracing depth was set to 5. For each run the time for it to render 510 frames was measured and then the average FPS (frames per second) was computed over that time period. For every test two runs were performed and the average results computed to try to minimise noise further (e.g. disk swap could cause a whole 510 frames to be inaccurate). Ideally it would have been preferable to do more than 2 runs for each test, however, since producing and compiling the results takes in excess of 3 hours it was decided not to do this due to time constraints.

The results are then shown in the following graph:

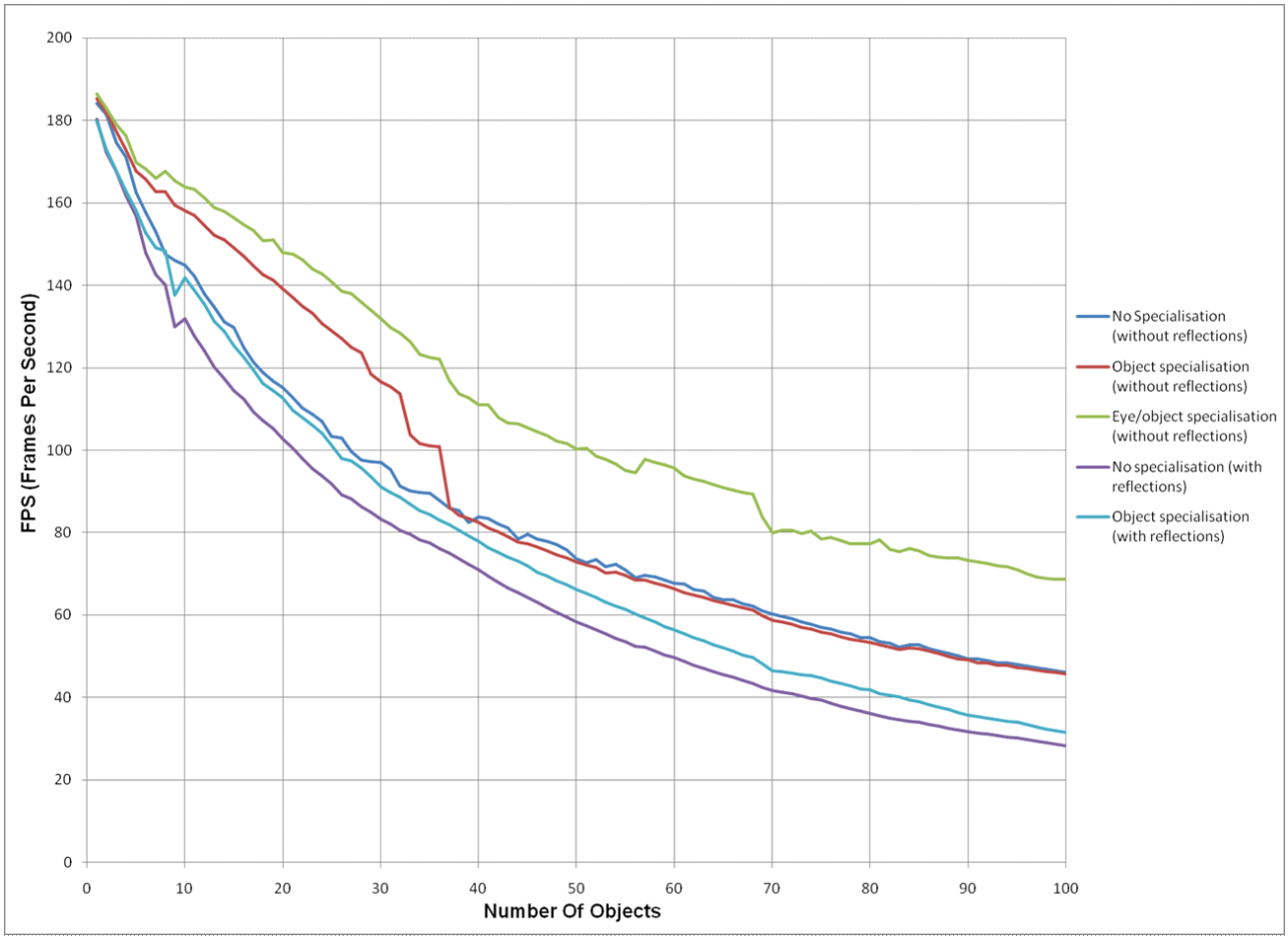


Figure 5.4: Ray Tracer Frame Rates (higher is better)

By far the greatest performance gains can be seen with eye/object specialisation compared to no specialisation without reflections. Object specialisation without reflections, however, gives some very nice performance gains for lower numbers of objects but at around 37 objects the performance gain suddenly drops down to nothing. Oddly, when comparing the cases with reflections enabled, object specialisation gives a fairly consistent small performance gain over no specialisation, even when there are more than 37 objects.

The results for object specialisation without reflections are rather puzzling. However, if the algorithm is modified slightly so that it only unrolls the loop for the first 30 objects and then proceeds with a normal rolled loop for the remaining objects then, by rerunning all the tests, the graph is as follows:

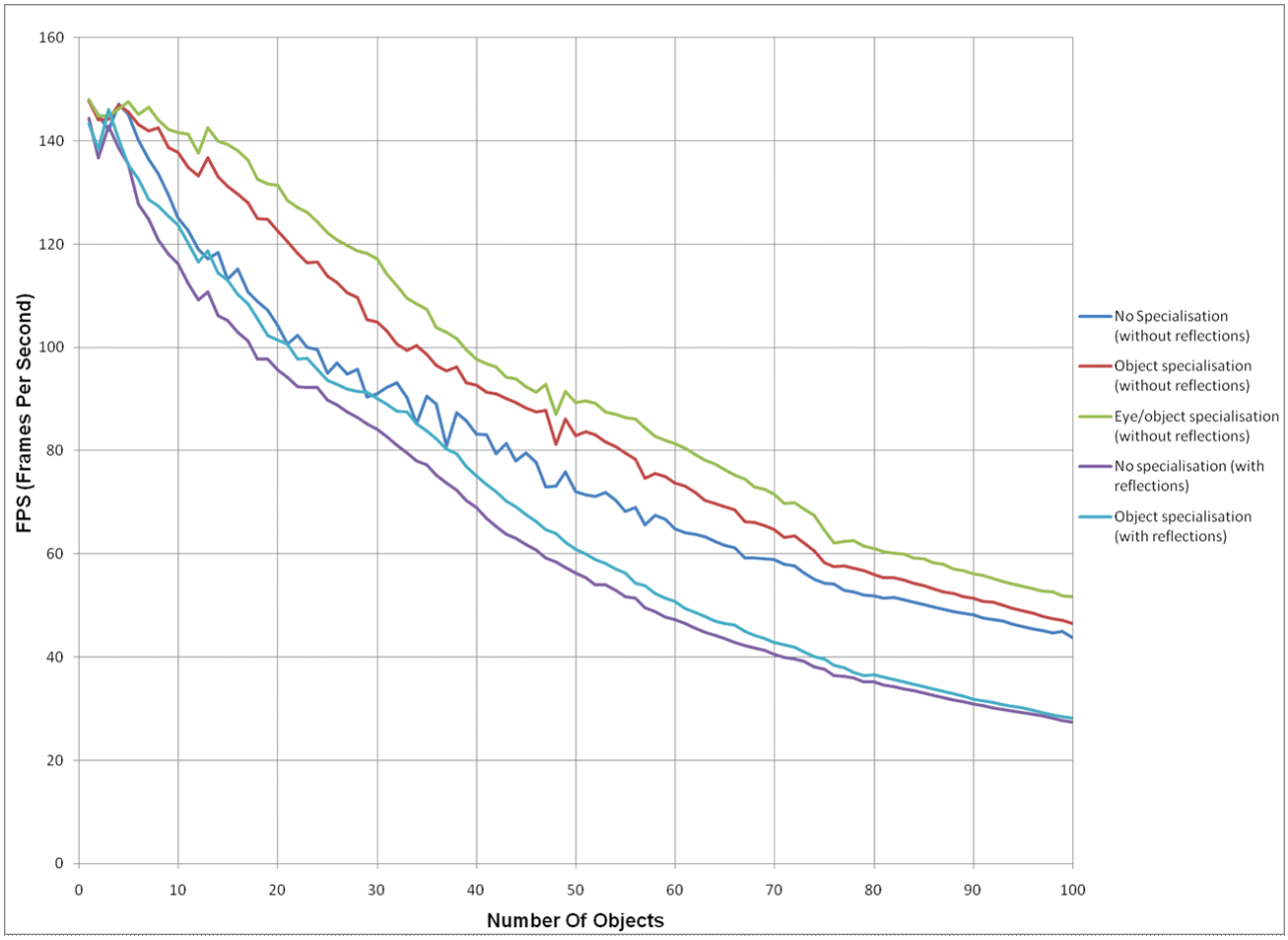


Figure 5.5: Ray Tracer Frame Rates (higher is better)

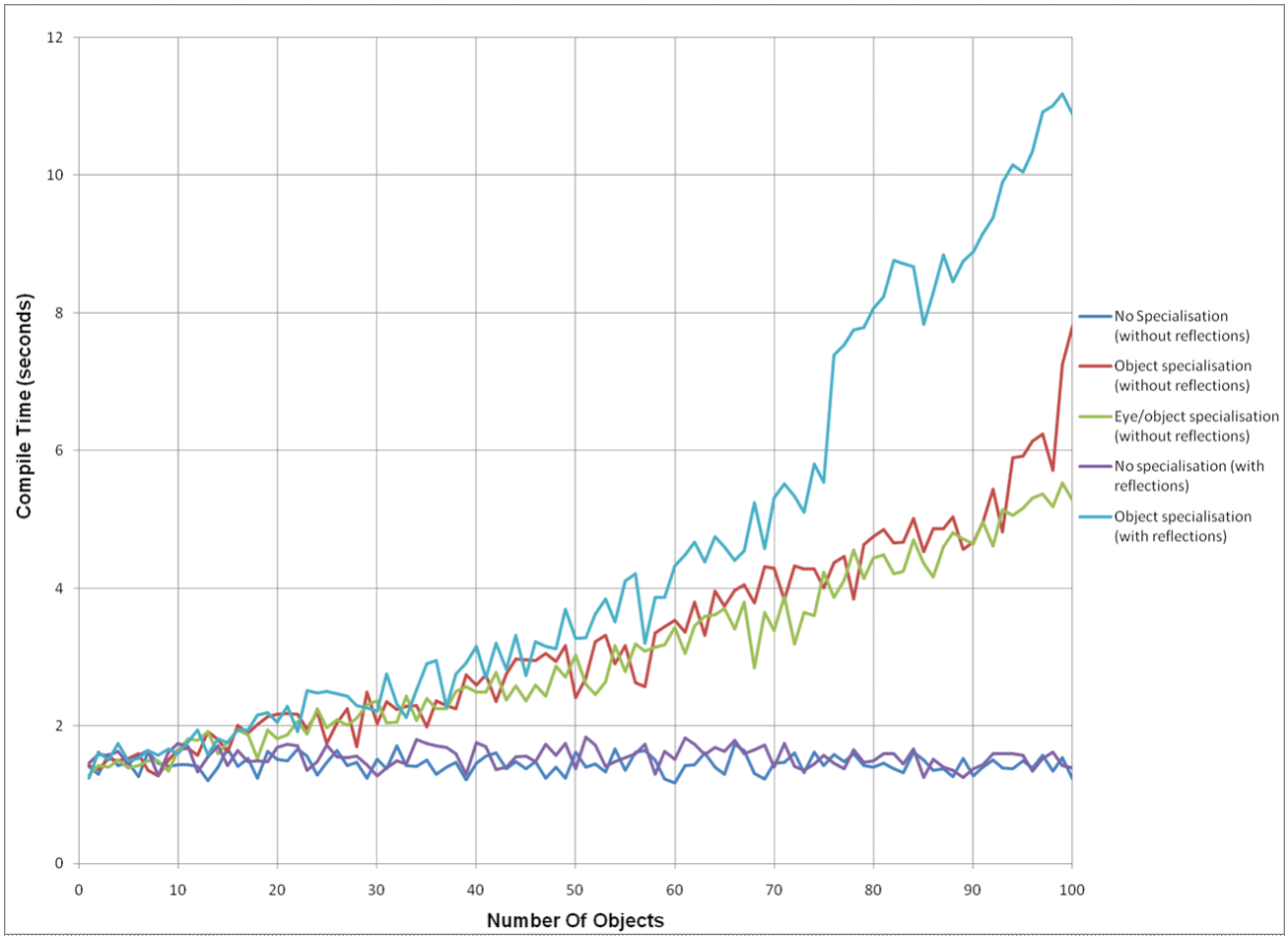
Now the object specialisation without reflections continues to provide a performance gain for much longer, however, this gain gradually drops off as more and more objects are added. This is unsurprising since the performance gain offered by the first 30 unrolled objects will get less significant the more total objects that are added. A similar drop off also occurs with eye/object specialisation and the two cases of reflections most likely for the same reasons.

By comparing Figure 5.5 with Figure 5.4 it can be seen that on the whole there are better performance gains with Figure 5.4 when specialising for more than 30 objects in all cases except for objection specialisation without reflections.

This still however, begs the question as to why there are such puzzling results for object specialisation without reflections. One possibility that immediately comes to mind given the above is that it could be an issue with the GPUs kernel code cache, which NVidia's documentation does not seem to discuss. However, one must assume that, especially considering that global memory access latencies can take 400-600 clock cycles [8], that there must be some sort of caching mechanism for the executable kernel code. Clearly, a large amount of loop unrolling could cause this to overflow. However, the difficulty with this theory is that the problem does not seem to exist when reflections are enabled and object specialisation is used, nor does it exist for eye/object specialisation, which should both use roughly the same amount of code memory as object specialisation without reflections.

The only final possibility would be to conclude that NVidia's compiler does appear to have several bugs, as was mentioned earlier in section 5.3.3.1, and it is possible that it simply is not very good at optimising code since sometimes certain combinations of code could cause it to make poor use of the available registers. Beyond this, we have been unable to come to a clear conclusion on this issue.

The compilation times are now briefly examined which were collected from our tests above. These do have a fair amount of noise for reasons discussed in section 5.3.1.3, however they still give a reasonable indicator of compilation performance. The following graph details these results:



Ray Tracer Compilation Times (lower is better)

There is a definite difference in compilation times. The specialised versions all have higher compilation times than the non-specialised versions. Oddly, we find that while object specialisation and eye/object specialisation without reflections have roughly the same compilation times, object specialisation with reflections has a noticeably higher compilation time than those two. This is surprising because object specialisation with reflections adds very little additional generated code over object specialisation without reflections. The only explanation would again appear to be that when reflections are enabled the Cuda compiler is doing something rather strange which is causing it to take longer.

5.4 Stability

During our development of the examples there were several bugs uncovered in TaskGraph for each of the examples tried. From duplicate tCall statements appearing to problems with the structures code causing segmentation faults when accessing a variable declared as a TaskGraph structure in normal C++ code. The main reason why so many bugs keep appearing is that TaskGraph is a large and complicated piece of software and, in our case, it was necessary to modify SUIF as well, which is an even bigger piece of software. This is furthered still by the fact that Cuda is a fairly complicated and not very thoroughly documented language, thus it is often necessary to assume certain things which are not necessarily true, for instance, assuming that typecasting in Cuda and C operate in the same way (see section 4.3.7).

It is for the above reasons that while we would like to say that TaskGraph is a thoroughly tested piece of software it is possible that there are still bugs. There are also a few known issues. One known issue is that function calling from Cuda kernels has not been implemented and attempting to do so is likely to result in undefined behaviour. The other issue is that Cell support has been broken (see section 4.2.2) since it would have been too time consuming to test TaskGraph on the Cell and learn how to program with the Cell architecture.

Furthermore, there are a couple of examples that were included in the TaskGraph distribution which no longer work due to architectural changes within TaskGraph. These are namely, mathintprt, which no longer compiles and it is believed that more than a minor fix may be required to get it to function properly as it is dependent on using the & operator on certain TaskGraph classes but we have overloaded it in order to add pointer support, which is causing problems. Secondly, the interpreter example (Note: This is NOT the same example as the expressions interpreter) hangs, it may well be that only a minor fix in the way parameters are passed is required to correct this. There were also several other examples which had been broken, however all of these examples were successfully given minor patches to make them work again with the new architecture.

5.5 Debugging

As has just been discussed in the previous section, TaskGraph has had and may continue to have some stability issues. Furthermore, when a user writes an example it can be hard to ascertain whether the problem is in the example code or the code generated by TaskGraph. Hence, it will be briefly discussed how the TaskGraph examples were debugged both to check that the example code is correct and that there are no bugs in TaskGraph causing it to generate incorrect code.

TaskGraph outputs the sourcecode that it generates to a file, usually in /tmp, this was helpful as one can then check to see if the generated sourcecode is correct. When converting from a non-TaskGraph example then it can also be helpful to copy and paste parts of the sourcecode back into the original non-TaskGraph example, if the mistakes cannot be spotted, in order to isolate which part of the code is wrong. Once the incorrect statement is found it is usually trivial to check whether the generator sourcecode is corresponding correctly with the generated sourcecode, if it is not, then there is likely a bug in the TaskGraph library.

5.6 Completeness of the Extension

We have managed to implement a large subset of the language features available in Cuda. However, there are a few small areas where such features were not implemented. The features which are still not supported by the TaskGraph extension are highlighted below:

- TaskGraph has no support for texture memory mainly because there do not seem to be many instances in NVidia's examples where this was actually used and actually likely to yield interesting results when combined with metaprogramming. Thus, it was decided not to implement it.
- Function calling inside kernels is not presently implemented. Implementation of such a feature would require special care as the __device__ attribute (which specifies that it is a kernel function rather than a host function) would have to be set correctly. Furthermore, Cuda kernels do not support recursion, meaning that it should ideally be ensured that they are not allowed to do this. Of course, the simple work around is to simply inline any external functions, possibly using macros (which is what was done for the ray tracing example).
- Constructors for the Cuda built-in data types is not currently supported (see section 5.3.1.2). While there is obviously a simple workaround for this it might be nice to add support for consistency with the Cuda language and it also allows for slightly neater code at the same time.
- Pointer arithmetic is not supported, thus one cannot directly add values to a pointer, although it is possible to dereference pointers like an array. This is caused mainly by a limitation in SUIF which does not seem to allow pointer arithmetic. We believe that it is likely that adding pointer arithmetic support to SUIF would be highly non-trivial.

6 Conclusion

The TaskGraph extension has been successfully completed and most of our original goals have been met. In particular:

- **TaskGraph Cuda extension** - The TaskGraph Cuda extension has been written and provides support for the vast majority of the Cuda language. Support has been added for things such as host and kernel TaskGraphs, calling Cuda functions, Cuda built-in structures and variables, Cuda global calls, Cuda variable type qualifiers, invoking the Cuda compiler and obtaining/manipulating pointers from Cuda memory allocation routines. This has also left intact TaskGraph's existing type safety features, for example, it is not possible to assign a dim3 structure to a float4 structure.

- **Generic TaskGraph extensions -** A variety of generic TaskGraph extensions have been added, including pointer support, neater methods of function calling including allowing TaskGraphs to call one another and partly as a side effect of this even recursion.

- **Single source model** - It is now possible for both the C++ code and the Cuda code to coexist in the same source file

One of the main objectives before starting this project was to see if once having written the TaskGraph Cuda extension we could find some Cuda examples which could really benefit from metaprogramming. This was a risky proposition not least because Cuda is a difficult language that is relatively low-level and thus it is necessary to be constantly conscious of the various different types of GPU memory (constant, shared, global, texture etc) along with issues to do with memory alignment, SIMD and multi-threading but also because these high risk elements could only come last in the project after the TaskGraph extension was built.

However, despite the risk, we managed to develop two good examples (the separable convolution filter and ray tracer) which really demonstrate how specialisation can give a performance boost to Cuda code along with one additional more introductory example (matrix multiplication) and a neat generic example illustrating how our TaskGraph extensions can improve the code of a non-Cuda example (the expressions interpreter). In particular, the achievements obtained through these examples are:

- The **separable convolution filter example** demonstrates how applications which wish to do a large amount of batch processing with different filters could obtain a substantial performance gain of potentially over 300% by using TaskGraph. It also allows one to optionally specialise to either just the filter size, or the actual values in the filter, thus allowing one to decide which type of specialisation is preferable for the given situation.

- The **ray tracing example** shows how a performance gain can be obtained by specialising to the objects or both the objects and the eye positions in the scene. It can offer performance gains of up to 30% for basic ray tracing with shadows, or a smaller performance gain of up to around 10% with reflections and shadows enabled.

- The **matrix multiplication example** is a less interesting, but worthwhile example which does matrix multiplication using TaskGraph Cuda and was designed to be a simple test bed to show the TaskGraph Cuda extension working and to make it easier for users to learn the language.

- The **expressions interpreter example** in the original TaskGraph distribution was modified to demonstrate how our generic TaskGraph extensions can be used to make non-Cuda examples easier to understand and code.

So, on the whole the project has been largely successful and our original goals have been largely met, however, there are some areas which are not ideal:

- **Completeness** - While most of the Cuda syntax is supported by our TaskGraph extension, there are a few areas which have been left out, most significantly, texture memory support and function calling from inside kernels (see section 5.6).

- **Type Safety** - While we have largely managed to keep TaskGraph's type and syntax safety capabilities, there are areas where type safety is not fully kept. In particular, function calls are not type safe, however, the situation has not really been made any worse in this regard since the old

method of function calling that was originally implemented in TaskGraph was not type safe either.

- **Bugs** - As was discussed in section 5.4, while TaskGraph has received a fair amount of testing, there may still be bugs present which could cause strange results when trying to code new applications which use the library.
- **Ray Tracer Specialisation Issues** - The ray tracer produced some strange results for object specialisation without reflections, preventing one from obtaining a performance boost with a large number of objects. The reason for this is unclear, however, it could be either a code caching issue and/or that the Cuda compiler is not terribly smart and is doing something slightly strange when compiling certain variations of the code.

6.1 Further work

Demonstrate specialisation with a normal 2D Convolution filter - We have shown performance advantages that can be offered with a separable convolution filter, however, a more flexible example would be to implement an all-purpose 2D convolution filter using TaskGraph Cuda that is not restricted only to separable filters. It is likely to be the case that one can then specialise this to different filter sizes and even different filters, giving a potential speed boost.

Implement the remainder of the Cuda language - Both texture memory and function calling from kernels are not currently supported (see section 5.6). For completeness, it would be worthwhile adding support for these. It may also be worthwhile adding support for the Cuda built-in data type constructors and pointer arithmetic support (as was also discussed in section 5.6).

Add Cuda kernel recursion support – Once function calling from kernels is supported this could perhaps be further extended to allow recursion to be supported in kernels even though Cuda does not natively support it. This could be done by essentially pushing and popping data onto a stack behind the scenes, in the same way that functions are usually handled.

Perform more extensive testing of the TaskGraph extension - While the extension has received a fair amount of testing, as the Cuda language is quite vast it is inevitable that there may still be some bugs remaining. In particular, some issues were highlighted which are believed to exist that have broken Cell support (see section 4.2.2) and it is always possible that there may still exist some other unknown bugs in the extension.

Type-safe Function Calls - Our implementation does not provide very good type-safety for function calls. One of the major problems is that it does not appear to be very feasible to ensure type-safety for function calls if the TaskGraph library has not been told the specific details of the function which the user wishes to call, since it will not know the function type. It may, however, be worth exploring this further since if it is to be assumed that the TaskGraph library is aware of the function to be called then, for example, it would be possible to create a series of wrapper functions which would be used to check that the parameter types are correct. The wrapper function would then simply call the tCall function with the appropriate function name. This is similar to some code which was already in the TaskGraph library, however, it does pose problems in that the function specification must be pre-programmed into TaskGraph.

However, it may be an idea to consider using a combination of the above method and standard tCalls. For instance, all the Cuda functions could be pre-programmed but with a tCall function on standby if the user needs to call a function which has not been pre-programmed. The other possibility would be whether some sort of TaskGraph preprocessor could be used, the preprocessor could then be invoked on the source file for the C++ program, it could then establish what include files it uses and search for the appropriate header information from those include files and then generate the appropriate function wrapper definitions so when the C++ compiler is invoked it will be able to determine whether or not the calls are type safe. It could even be possible to make this as a program which searches the hard disk for all include files and generates TaskGraph wrappers for them and then simply puts them in a special include file which all TaskGraph programs must include. The user then only needs to run this each time they update the include files on their system.

Memory allocator for constant memory - One of the issues which was encountered is that Cuda does not allow dynamic allocation of constant memory and the amount of constant memory that may be allocated is limited to 64k. Therefore a nice idea would be to write a constant memory allocator, which could simply allocate all 64k (or perhaps some lower maximum that could be defined by the user) at start to itself, when constant memory is needed one could simply request it by making a call to the memory allocator which would simply return a pointer to a free region of the required size. It would of course have to carefully handle issues such as memory fragmentation and also there is the question of whether its memory management code should run on the GPU or the CPU or a combination of the two which would also be likely to influence whether the user can call the allocation function from host or kernel code or both.

Improve Ray Tracer – While fairly substantial performance improvements have been demonstrated through specialisation with a relatively simple ray tracer, it would be a good idea to code a more sophisticated real-world ray tracer using TaskGraph Cuda and see if this will also yield performance benefits through specialisation. Features which could be added would be polygon-based ray tracing as opposed to sphere based and also more sophisticated reflectivity (e.g. Firing reflected rays at random within the bounds of a cone so as to allow for non-mirror reflections and also support for different objects with different reflectivity values).

Implement Radiosity with Specialising - An interesting idea would be to see if it might be possible to implement a radiosity algorithm using TaskGraph Cuda and specialise this, most likely in a similar way to the ray tracer (i.e. specialise to object positions). Due to the speed of modern GPUs it is also likely that this could be done in real-time.

Improvements to the Indeterministic Control Flow Instructions – In section 3.2.9 we highlighted our implementation of these additional “Maybe” control flow instructions and some of their weaknesses was also highlighted. It would be useful if a way could be found of being able to use the same loop variable for both the rolled and unrolled versions of a loop, one possibility would be to not use the Maybe instruction for the actual loops (although they might still be used, say, for some if statements inside the loop when, for instance, different variables are to be used depending on the specialisation) but instead to be able to explicitly tell TaskGraph to unroll that particular loop. This solution might also do away with the slightly nasty syntax required in the standard C++ loop that ensures it only does 1 iteration if specialisation is not enabled. It would furthermore be worthwhile exploring what kind of programs could benefit from these additional instructions since in this report we have only provided one example of them in practical use (see section 5.3.3.1 on our ray tracer).

tFor improvements - TaskGraphs tFor loop construct is defined in a fairly unusual way and lacks the power of proper C for loops. It requires one to specify the loop variable, the lower bound, upper bound and, in the case of tForStep, the increment. However, normal for loops take the first parameter as being a statement to be executed at the start, the second parameter as being a boolean expression to be evaluated each iteration and the third parameter as being a statement to be executed at the end of each iteration. Clearly standard for loops provide for much more flexibility and ease of use. Thus, a useful feature would be to explore whether it is possible to improve the tFor loops to function in a way similar to standard C for loops.

Architecture Independent Coding Without Performance Loss - In the background (section 2), the RapidMind project was mentioned which is in some ways similar to this project, except their approach is to allow coding in an architecturally independent way, whereas in our case we have decided to allow the programmer to code in an architecturally dependent way with a view to allowing the programmer to obtain a maximal performance gain. However, a third alternative approach would be to allow the programmer to code mostly in an architecturally independent way, but then allow some small isolated additional code which could be added to optimise the architecturally independent code for specific architectures. This would potentially allow the performance gains of architecturally dependent code to be kept, whilst striving to keep the simplicity and portability of architecturally independent code.

7 Bibliography

[1] Olav Beckmann, Alastair Houghton, Paul H J Kelly. Run-time Code Generation in C++ as a foundation for domain-specific optimisation. In Domain-Specific Program Generation, 2003 International Seminar, Germany. March 2003. URL: http://www.doc.ic.ac.uk/~phjk/phjk-Publications.html

[2] Web homepage of MetaOCaml. Accessed March 2008. URL: http://www.metaocaml.org/

[3] Walid Taha. A Gentle Introduction to Multi-stage Programming. In DSPG, 2004. URL: http://www.cs.rice.edu/~taha/publications/journal/dspg04a.pdf

[4] OpenMP ARB. Homepage of OpenMP. Accessed March 2008. URL: http://www.openmp.org

[5] Wikipedia. OpenMP page. Accessed March 2008. URL: http://en.wikipedia.org/wiki/OpenMP

[6] Berkeley UPC Compiler Group. Homepage of UPC. Accessed March 2008. URL: http://upc.lbl.gov/

[7] Konstantine Berlin. UPC vs. MPI and OpenMP: Analysis of a Hybrid Approach to Parallel Programming. November 12, 2002. URL: http://www.cs.umd.edu/Honors/reports/berlin.pdf

[8] NVIDIA Corporation. NVIDIA Cuda Programming Guide Version 1.1. URL http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[9] Anton Lokhmotov, Benedict R. Gaster, Alan Mycroft, Neil Hickey, David Stuttard. Revisiting SIMD programming. In Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07). Urbana, Illinois, USA. URL: http://www.cl.cam.ac.uk/~al407/research/papers/lcpc07.pdf

[10] Web homepage of Clearspeed Technology. Accessed March 2008. URL: http://www.clearspeed.com

[11] Andrew Richards, Chief Technology Architect at Codeplay Technology Limited. The Codeplay Sieve C++ Parallel Programming System. Whitepaper, URL: http://www.codeplay.com/downloads_public/sievepaper-2columns-normal.pdf

[12] Wikipedia. GLSL page. Accessed March 2008. URL: http://en.wikipedia.org/wiki/GLSL

[13] António Ramires Fernandes. OpenGL Shading Language Tutorial. Lighthouse 3D web site. Accessed March 2008. URL: http://www.lighthouse3d.com/opengl/glsl/index.php?vertexp

[14] OpenGL ARB. Fragment Shader Specification. Accessed March 2008. URL: http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_shader.txt

[15] Wikipedia. High Level Shader Language page. Accessed March 2008. URL: http://en.wikipedia.org/wiki/High_Level_Shader_Language

[16] Wikipedia. Geometry Shader page. Accessed March 2008. URL: http://en.wikipedia.org/wiki/Geometry_shader

[17] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. In ACM SIGPLAN Notices, volume 29, issue 12, pp 31-37. 1994. URL: ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/94/620/CSL-TR-94-620.pdf

[18] NVIDIA Corporation. NVIDIA Tesla GPU web site. Accessed May 2008. URL: http://www.nvidia.com/object/tesla_c870.html

[19] Wikipedia. Cg Programming Language page. Accessed May 2008. URL: http://en.wikipedia.org/wiki/Cg_%28programming_language%29

[20] Wikipedia. Stream Processing page. Accessed May 2008. URL: http://en.wikipedia.org/wiki/Stream_processing

[21] Stanford Imagine Project homepage. Accessed May 2008. URL: http://cva.stanford.edu/projects/imagine/

[22] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine Stream Processor. In the Proceedings of the International Conference on Computer Design. 2002. URL: http://cva.stanford.edu/publications/2002/kapasi_iccd2002_arch.pdf

[23] Stanford Merrimac homepage. Accessed May 2008. URL: http://merrimac.stanford.edu/

[24] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, François Labonté, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, Ian Buck. Merrimac: Supercomputing with Streams. In the Proceedings of the SC'03 Conference, Phoenix, Arizona. November 2003. URL: http://merrimac.stanford.edu/publications/sc03_merrimac.pdf

[25] Brook Language homepage. Accessed May 2008. URL: http://graphics.stanford.edu/projects/brookgpu/lang.html

[26] LibSh homepage. Accessed May 2008. URL: http://libsh.org/

[27] ATI FireStream homepage. Accessed March 2008. URL: http://ati.amd.com/technology/streamcomputing/index.html

[28] The OpenVIDIA project. Accessed March 2008. URL: http://openvidia.sourceforge.net

[29] Lutz Latta. Building a Million Particle System. In the Game Developers Conference. 2004. URL: http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf

[30] NVidia Cuda SDK download page. Accessed May 2008. URL: http://www.nvidia.com/object/cuda_get.html

[31] Victor Podlozhnyuk. Image Convolution with Cuda. June 2007. URL: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/convolutionSeparable/doc/convolutionSeparable.pdf

[32] Stanford Shading Language homepage. Accessed June 2008. URL: http://graphics.stanford.edu/projects/shading/

[33] TaskGraph website. Accessed January 2008. URL: http://www.doc.ic.ac.uk/~phjk/Software/TGL/

[34] Cuda ray tracer example web site. Eric Rollins. Accessed June 2008. http://eric_rollins.home.mindspring.com/ray/cuda.html

[35] ZDNet news article on 80 core Intel processors. Accessed June 2008. URL: http://news.zdnet.com/2100-9584_22-6158181.html

Appendix A (Cuda Matrix Multiplication Sourcecode)

```
/*
 * Copyright 1993-2007 NVIDIA Corporation.  All rights reserved.
 *
 * NOTICE TO USER:
 *
 * This source code is subject to NVIDIA ownership rights under U.S. and
 * international Copyright laws.  Users and possessors of this source code
 * are hereby granted a nonexclusive, royalty-free license to use this code
 * in individual and commercial software.
 *
 * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
 * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
 * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
 * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
 * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
 * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
 * OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
 * OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE USE
 * OR PERFORMANCE OF THIS SOURCE CODE.
 *
 * U.S. Government End Users.   This source code is a "commercial item" as
 * that term is defined at  48 C.F.R. 2.101 (OCT 1995), consisting  of
 * "commercial computer  software"  and "commercial computer software
 * documentation" as such terms are  used in 48 C.F.R. 12.212 (SEPT 1995)
 * and is provided to the U.S. Government only as a commercial end item.
 * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
 * 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
 * source code with only those rights set forth herein.
 *
 * Any use of this source code in individual and commercial software must
 * include, in the user documentation and internal comments to the code,
 * the above Disclaimer and U.S. Government End Users Notice.
 */

/* Matrix multiplication: C = A * B.
 *
 * This example is based on NVIDIAs sample code as described below but modified for taskgraph as a simple cuda
example.
 * This sample implements matrix multiplication and is exactly the same as
 * Chapter 7 of the programming guide.
 * It has been written for clarity of exposition to illustrate various CUDA
 * programming principles, not with the goal of providing the most
 * performant generic kernel for matrix multiplication.
 *
 * CUBLAS provides high-performance matrix multiplication.
 */

#include <stdio.h>
#include <stdlib.h>
#include <TaskGraph>
#include "TaskUserFunctions.h"
#include <stdarg.h>
#include <boost/preprocessor.hpp>
#include <sys/time.h>
```

```
using namespace tg;

typedef TaskGraph<void> cuda_host;
typedef TaskGraph<void, float*, float*, float*, int, int, int> cuda_kernel;

#include <TaskCudaTypes.h>

/* DEFINES TAKEN FROM matrixMul.h */

// Thread block size
#define BLOCK_SIZE 16

// Matrix dimensions
// (chosen as multiples of the thread block size for simplicity)
//#define WA (40 * BLOCK_SIZE) // Matrix A width
//#define HA (60 * BLOCK_SIZE) // Matrix A height
//#define WB (18 * BLOCK_SIZE) // Matrix B width
#define WA (8 * BLOCK_SIZE) // Matrix A width
#define HA (10 * BLOCK_SIZE) // Matrix A height
#define WB (10 * BLOCK_SIZE) // Matrix B width
#define HB WA  // Matrix B height
#define WC WB  // Matrix C width
#define HC HA  // Matrix C height

/* END matrixMul.h */

double utime () {
  struct timeval tv;

  gettimeofday (&tv, NULL);

  return (tv.tv_sec + double (tv.tv_usec) * 1e-6);
}

int main( int argc, char *argv[] ) {
  cuda_host T;
  cuda_kernel T2;
  cudakerneltaskgraph(cuda_kernel, T2, tuple5(C, A, B, wA, wB))
  {
    tVar(int, bx);
    tVar(int, by);
    tVar(int, tx);
    tVar(int, ty);

    // Block index
    bx = blockIdx.x;
    by = blockIdx.y;

    // Thread index
    tx = threadIdx.x;
    ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    tVar(int, aBegin);
    aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    tVar(int, aEnd);
    aEnd = aBegin + wA - 1;
```

```
    // Step size used to iterate through the sub-matrices of A
    tVar(int, aStep);
    aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    tVar(int, bBegin);
    bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    tVar(int, bStep);
    bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    tVar(float, Csub);
    Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    tVar(int,a);
    tVar(int,b);
    b = bBegin;
    tForStep(a,aBegin,aEnd,aStep)
    {
        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        tVarShared(float[BLOCK_SIZE][BLOCK_SIZE], As);

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        //shared_float1 Bs[BLOCK_SIZE][BLOCK_SIZE];
        tVarShared(float[BLOCK_SIZE][BLOCK_SIZE], Bs);

        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        tCall("__syncthreads");

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        tVar(int,k);
        tFor(k,0,BLOCK_SIZE-1)
//          for(int k = 0; k < BLOCK_SIZE; k++)
        {
          Csub += As[ty][k] * Bs[k][tx];
        }

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        tCall("__syncthreads");
        b += bStep;
    }

    // Write the block sub-matrix to device memory;
```

```
  // each thread writes one element
  tVar(int,c);
  c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;

}

cudahosttaskgraph( addc_TaskGraph, T, tuple0() ) {

  tInclude("stdlib.h");
  tInclude("stdio.h");
  tInclude("string.h");
  tInclude("math.h");
  tInclude("cutil.h");

  tCall("CUT_DEVICE_INIT");

  // set seed for rand()
  tCall("srand", 2006);

  // allocate host memory for matrices A and B
  unsigned int size_A = WA * HA;
  unsigned int mem_size_A = sizeof(float) * size_A;
  tVar(float*,h_A);
  h_A = tCall("malloc", mem_size_A);
  unsigned int size_B = WB * HB;
  unsigned int mem_size_B = sizeof(float) * size_B;
  tVar(float*,h_B);
  h_B = tCall("malloc", mem_size_B);

  // Initialize host memory by inserting random values into the matrices
  tVar(int, i);
  tFor(i,0, size_A - 1)
  {
    h_A[i] = tCall("rand") / (float)RAND_MAX;
  }
  tFor(i,0, size_B - 1)
  {
    h_B[i] = tCall("rand") / (float)RAND_MAX;
  }

  // allocate device memory
  tVar(float*, d_A);
  tCall("CUDA_SAFE_CALL", tCall("cudaMalloc", cast<void**>(&d_A), mem_size_A));
  tVar(float*, d_B);
  tCall("CUDA_SAFE_CALL", tCall("cudaMalloc", cast<void**>(&d_B), mem_size_B));

     // copy host memory to device
  tCall("CUDA_SAFE_CALL", tCall("cudaMemcpy", d_A, h_A, mem_size_A, cudaMemcpyHostToDevice));
  tCall("CUDA_SAFE_CALL", tCall("cudaMemcpy", d_B, h_B, mem_size_B, cudaMemcpyHostToDevice));

  // allocate device memory for result
  unsigned int size_C = WC * HC;
  unsigned int mem_size_C = sizeof(float) * size_C;
  tVar(float*, d_C);
  tCall("CUDA_SAFE_CALL", tCall("cudaMalloc", cast<void**>(&d_C), mem_size_C));

  // create and start timer
  tVar(unsigned int, timer);
  timer = 0;
```

```
tCall("CUT_SAFE_CALL", tCall("cutCreateTimer", &timer));
tCall("CUT_SAFE_CALL", tCall("cutStartTimer", timer));


// setup execution parameters
tVar(dim3, threads);
threads.x = BLOCK_SIZE;
threads.y = BLOCK_SIZE;

tVar(dim3, grid);
grid.x = WC / threads.x;
grid.y = HC / threads.y;

// execute the kernel
tCudaGlobalCall(T2, grid,threads,0,d_C,d_A,d_B,WA,WB);

// check if kernel execution generated and error
tCall("CUT_CHECK_ERROR", "Kernel execution failed");

// allocate mem for the result on host side
tVar(float*, h_C);
h_C = cast<float*>(tCall("malloc", mem_size_C));

// copy result from device to host
tCall("CUDA_SAFE_CALL", tCall("cudaMemcpy", h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost));

// stop and destroy timer
tCall("CUT_SAFE_CALL", tCall("cutStopTimer", timer));
tCall("printf", "GPU Processing time: %f (ms) n\n", tCall("cutGetTimerValue", timer));
tCall("CUT_SAFE_CALL", tCall("cutDeleteTimer", timer));

timer = 0;
tCall("CUT_SAFE_CALL", tCall("cutCreateTimer", &timer));
tCall("CUT_SAFE_CALL", tCall("cutStartTimer", timer));

// compute reference solution
tVar(float*, reference);
reference = cast<float*>(tCall("malloc",mem_size_C));
tFor (i, 0, HA - 1)
{
 tVar(int,j);
 tFor (j, 0, WB - 1) {
  tVar(double, sum);
  sum = 0;
  tVar(int,k);
  tFor (k, 0, WA - 1) {
   tVar(double, a);
   tVar(double, b);
   a = h_A[i * WA + k];
   b = h_B[k * WB + j];
   sum += a * b;
  }
  reference[i * WB + j] = sum;
 }
}

// stop and destroy timer
tCall("CUT_SAFE_CALL", tCall("cutStopTimer", timer));
tCall("printf", "CPU Processing time: %f (ms) n\n", tCall("cutGetTimerValue", timer));
tCall("CUT_SAFE_CALL", tCall("cutDeleteTimer", timer));
```

```
    // check result
    tVar(int, res);
    res = tCall("cutCompareL2fe", reference, h_C, size_C, 1e-6f);
    tCall("printf", "Test ");
    tIf (1 == res)
    {
      tCall("printf", "PASSED\n");
    }
    tIf (1 != res)
    {
      tCall("printf", "FAILED\n");
    }

    // clean up memory
    tCall("free", h_A);
    tCall("free", h_B);
    tCall("free", h_C);
    tCall("CUDA_SAFE_CALL", tCall("cudaFree", d_A));
    tCall("CUDA_SAFE_CALL", tCall("cudaFree", d_B));
    tCall("CUDA_SAFE_CALL", tCall("cudaFree", d_C));
  }

  const char* args[] = {"-Xcompiler", "-fPIC", "-I", "/usr/local/cuda/include", "-I",
"/home/hectonix/NVIDIA_CUDA_SDK/common/inc/", "-L", "/home/hectonix/individual_project/", "-lGL", "-lGLU",
"-lcutil", NULL};

  double t = -utime ();
  T.compile(tg::NVCC, true, args);
  t += utime ();
  printf("Compilation time took %f seconds\n", t);
  T();
  T();

}
```