# Towards an Implementation of OpenMP on the NVIDIA G80 Series Architecture

Keir, Paul

Published: 23/08/2007


# Towards an Implementation of OpenMP on the NVIDIA G80 Series Architecture

Paul Keir

August 23, 2007

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2007

**Abstract**

The following dissertation describes the design and creation of an OpenMP C compiler. The compiler targets the parallel capability of a modern series of NVIDIA graphics cards, and the generated output source code is written in the NVIDIA CUDA C language. The project also proposes and implements an extension to the OpenMP language: a single new directive and supporting runtime routines. By this route, problems with pointer indirection in the presence of a dual address space memory architecture are relieved.

# Contents


**1 Introduction** **1**

**2 Background** **2**
2.1 OpenMP . . . . . 2
2.2 Source to Source Compilers . . . . . 3
2.3 GPGPU . . . . . 3
2.3.1 Shading Languages . . . . . 4
2.3.2 Third Party GPGPU Solutions . . . . . 4
2.3.3 NVIDIA CUDA . . . . . 5

**3 ANTLR** **7**
3.1 Backus-Naur Form . . . . . 8
3.1.1 Extended Backus-Naur Form . . . . . 8
3.1.2 Context-Free Grammars . . . . . 9
3.2 Top-Down Recursive-Descent Parsing . . . . . 9
3.3 ANTLR Actions . . . . . 11
3.4 ANTLRWorks . . . . . 11
3.5 ANTLR TokenRewriteStream . . . . . 12

**4 The NVIDIA CUDA C Programming Language** **14**
4.1 The Driver and Runtime API . . . . . 14
4.2 Built-in Variables . . . . . 14
4.3 Syntax Overview . . . . . 15
4.4 Shared Memory and Registers . . . . . 16
4.5 Native Synchronisation . . . . . 17
4.6 CUDA Language Restrictions . . . . . 18

**5 Memory Management** **19**
5.1 Placement of Parallel Regions . . . . . 19
5.2 Two Memory Subsystems . . . . . 20
5.3 Pointers . . . . . 21
5.4 Further Indirection . . . . . 22
5.5 Stack Allocated Variables . . . . . 23
5.6 Runtime Stack Access . . . . . 23
5.7 Dynamic Memory Allocation . . . . . 24

**6 Language Design** **26**
6.1 Accessible Memory . . . 26
6.2 Customising Malloc . . . 28
6.3 Function Decoration . . . 28
6.4 Limitations . . . 29

**7 Front End - Parsing** **31**
7.1 BNF Language Grammars . . . 31
7.1.1 Base Language Considerations . . . 31
7.1.2 The C Grammar . . . 32
7.1.3 OpenMP Grammar . . . 33
7.1.4 Extensions to the OpenMP Grammar . . . 34
7.2 Tree Representations . . . 35
7.3 Intermediate Data Structures . . . 36

**8 Back End - Code Generation** **37**
8.1 Approaching Code Translation . . . 37
8.1.1 Registers versus Shared Memory . . . 38
8.1.2 1D Blocks and Grids . . . 38
8.2 Directives . . . 39
8.2.1 Parallel Construct . . . 39
8.2.2 Work-sharing Constructs . . . 41
8.2.3 Combined Parallel Work-sharing Constructs . . . 45
8.2.4 Master and Synchronisation Constructs . . . 45
8.2.5 Data Environment . . . 48
8.3 Runtime Library Routines . . . 54
8.3.1 Execution Environment Routines . . . 54
8.3.2 Lock Routines . . . 56
8.3.3 Timing Routines . . . 56
8.4 Host Runtime Library . . . 57
8.4.1 `gpump_copy_to_device` . . . 57
8.4.2 `gpump_alloc_list_add` . . . 57
8.4.3 `gpump_malloc` . . . 57
8.5 Implementing the Accessible Extensions . . . 58
8.5.1 `accessible` data . . . 58
8.5.2 `accessible` functions . . . 58

**9 Performance Analysis** **59**
9.1 The Watchdog Timer . . . 59
9.2 Host to Device Transfer . . . 60
9.3 Speedup . . . 61
9.4 Comparison to Traditional OpenMP . . . 62
9.5 Analysis . . . 62
9.6 Compiler Execution Runtime . . . 63

**10 Conclusions** **64**
10.1 Development Summary . . . . . 64
10.2 Future Work . . . . . 65
10.3 Conclusion . . . . . 66

**A ANTLR OpenMP Grammar** **67**

**B Entry-Point Grammar Rules** **73**

**C OpenMP Loop Construct Example** **75**

**D Translated OpenMP Loop Construct Example** **76**

**E System Specifications** **79**

# List of Figures

8.1 Minimal Loop Concept . . . . . 42
8.2 Translated Minimal Loop Concept . . . . . 43

9.1 Array Size on Performance with 256 Threads on 4 Blocks . . . . . 60
9.2 Contrast of Block and Thread Scaling . . . . . 61

# Acknowledgements

I would first of all like to thank my supervisor, Mark Bull, for his generous assistance.

To my wife Steffi, sincere thanks for your loving support.

Many thanks also to the fine lecturers, and supporting staff of the MSc. in HPC, and also to my fellow students who have been valued companions, and a highly motivational influence.

# Chapter 1

# Introduction

The dual influence of a strong commercial market, with the highly parallel nature of three-dimensional interactive rendering, has allowed modern graphics cards to offer an extremely high price to performance ratio. The following MSc. dissertation proposes that the OpenMP parallel programming API could usefully be applied to describe programs compiled for the contemporary G80 series of NVIDIA graphics cards.

Cards from this range are composed of up to 128 serial processors, and can be programmed using a new parallel language system based on C; also from NVIDIA, and known as the Compute Unified Device Architecture, or CUDA. NVIDIA provide a compiler for CUDA, and the proposal primarily involves the development of a translating compiler which can accept a C OpenMP program as input, before automatically creating compilable CUDA C as output.

Following this introduction, Chapter 2 provides the broader context in which the work is set. Chapters 3 and 4 then provide further background information, presenting the two vital technical components, ANTLR and CUDA. Chapter 5 discusses memory management, while Chapter 6 proposes minor related modifications to OpenMP. Chapter 7 demonstrates the role of parsing, and ANTLR in analysing code, in preparation for the back-end code generation, discussed in the subsequent Chapter 8. Finally Chapter 9 looks briefly at performance benchmarking before Chapter 10 concludes the dissertation.

Appendix A and B include excerps from the grammar. Appendix C and D then provide a before and after example of the compiler's code translation. Finally, Appendix E lists the specifications of the development and testing system.

# Chapter 2

# Background

There is presently much interest surrounding the mainstream adoption of parallel processing. The term *multi-core* is highly marketed, and core counts have replaced the clock counts as a consumer metric of quality; commodity processors everywhere incorporate two or four processor cores. Furthermore, such incremental improvements are likely to continue: IBM's recent demonstration of an 80-core processor as part of their *TeraScale*[14] project inspired many, but leave many pondering how best to extract such computational power.

The IBM/Toshiba/Sony-developed Cell Broadband Engine (CBE) processor[13] too has multiple heterogenous cores. While also marketed as a HPC component, the CBE sits in the living room of millions worldwide as the main CPU of the Playstation3.

The videogame market plays a role in one more parallel system. Approaching the processor market from a different avenue, graphics cards and their floating-point capacity have fostered a new community developing their use on non-graphical, *general purpose* applications. NVIDIA has even recently begun marketing to the HPC sector directly, with the 128-processor variant cards comprising the basic element in their *Tesla*[8] product range.

Although the focus of this dissertation directly involves graphics cards, it is clear that chip parallelism will soon become commonplace. The challenge remaining is for software developers to embrace the change. One established technology offering a standardised method of accessing shared memory parallelism is OpenMP.

## 2.1 OpenMP

Most popular computer languages in use today were designed principally for serial proram development. Imperative languages such as C, C++ and Fortran require some augmentation to allow developers to write parallel programs. OpenMP is a publicly

available standard defining a suite of compiler directives, library routines, and environment variables for use in shared-memory parallelism by these three languages.

In addition to the benefit of standardised portability, OpenMP's method of controlling parallelism using **`#pragma`** compiler directives interspersed with existing code allows for both serial and parallel code to develop together, co-existing lexically within the same file. A command line compiler option, perhaps with some conditional compilation, is then usually sufficient to switch between the parallel, or more easily debugged serial version. Considered together with the manageably small collection of parallel language tokens specified by OpenMP, as distinct from one built upon message passing library calls, is often believed to be more intuitive, and to reduce code development times.

Finally, although message passing libraries can operate on shared memory architectures, it is widely reported that OpenMP can provide better performance.

## 2.2 Source to Source Compilers

The output of OpenMP compilers is often not machine code. The granularity of parallelism exposed through OpenMP is equivalent to that of common threading libraries. It follows that many OpenMP compilers will transform code into a functionally equivalent program; with method calls to threading libraries such as POSIX Threads (Pthreads) replacing any OpenMP instructions. Performing such code transformations *outside* of the main compiler though may have negative performance implications. However, the portability benefits can often be sufficient to offset such qualms.

The C programming language is also a popular compilation target. Many computer languages, such as the functional language Haskell, or the parallel language Unified Parallel C (UPC), target it with their compilers. IBM's experimental new language, X10 compiles to Java source code. Within such fields of language research, source to source compilation is often the most practical solution. Low-level, assembly-level features, such as register allocation, are the primary sacrifice.

## 2.3 GPGPU

Although from a historical perspective, the use of graphics hardware for general purpose computing can be traced to the formation in 1978 of Ikonas Graphics Inc.[9], much of the current interest in this area stems from the emergence of auxiliary real-time graphics cards in the late 1990s.

At upwards of sixty times per-second these graphics cards transform multitudes of three-dimensional coordinates, in parallel, to produce a final, two-dimensional image. This restricted and explicitly parallel mode of computation has allowed the floating

point processing capability of such devices to dramatically exceed that of all current multi-core processors.

More recently, the use of such commodity units for general purpose computation, dubbed GPGPU, has blossomed into both a vibrant research field, and a strong differentiating factor between competing graphics hardware vendors. GPGPU has proven highly capable of data parallel tasks as varied as partial differential equations[10], biological sequence matching[11], and quantum chromodynamics[12] to name but a few. NVIDIA's Tesla hardware range, supported by the CUDA language, is now the first graphic card technology to directly target the HPC market.

Nevertheless, the requirement for a programmer to use a traditional graphics API and shading language when developing for such devices, together with associated restrictive hardware access patterns, had left many unconvinced.

### 2.3.1 Shading Languages

The oft-repeated transformation of graphics primitives, such as points, triangles and texture maps into a final raster screen image, is known as the graphics pipeline. As sections of this pipeline ultimately became programmable, a handful of new programming languages emerged; languages such as the OpenGL Shading Language, or NVIDIA's Cg. Programs written in these languages are known as 'shaders', and in their compiled form, their execution is managed by a host application. Typically shaders are used to enable compelling or artistic variation in the aesthetic representation of interactive three-dimensional virtual environments.

Appropriately for languages concerned with realtime graphics, shading languages are often highly reminiscent of the C programming language. Distinguishing features include new built-in variables and variable qualifiers, a specialised and minimal runtime API, and vector and matrix data types. Execution of the resulting shader programs is then controlled through graphics APIs running on the host. Programs may have been compiled already, or may be compiled at runtime.

For those less concerned with image rendering, such languages would quickly reveal how thin the veneer of C was. Adherence to a traditional graphics pipeline was still required by the underlying hardware, meaning that certain variable types could be manipulated only at certain locations upon it. From an HPC perspective this meant that such systems supported a generalised gather but not scatter to DRAM locations. The rapidly changing nature of the hardware industry could also mean that something like branching may or may not be supported by any given card.

### 2.3.2 Third Party GPGPU Solutions

Recent research projects and commercial endeavours have emerged to present usable, general purpose computing APIs targeting the GPU. These GPGPU platforms, APIs

and languages are often implemented as software wrappers around host graphics APIs like OpenGL or Microsoft's DirectX. A user typically then has the common pairing of host graphics API and shading language entirely subsumed within a single concise interface.

In common with many abstraction methodologies, this approach offers advantages over traditional methods based on graphics application development:

- Reduced Learning Curve
- Increased Device Independence
- Configurable Targeting of either CPU or GPU
- No Shader File Management
- Simpler Portability of General Codes.

Notable projects taking this approach include Stanford University's BrookGPU; Microsoft's C# Accelerator; Waterloo University's LibSh, and Rapidmind's commercial successor to it.

Each of these projects though all share a common weakness: a new interface must be learned by the novice. Furthermore, none are yet firmly established and so in the long term, code maintainability is an issue. Without the backing of an open industry standard, many HPC developers and researchers are no doubt cautious of investing in languages and APIs which may be unsupported in five years. Confirming such doubts, both Accelerator and the Sh project increasingly appear dormant. Another related technology vendor, PeakStream Inc. was recently acquired by Google.

### 2.3.3 NVIDIA CUDA

NVIDIA's CUDA is a new software interface for G80 series developers interested in GPGPU. CUDA extends the 'C' programming language to allow the management of both host and device operations, even within a single file or function, and is compatible with Linux and Windows operating systems. The CUDA compiler driver accepts an extended 'C' language file, optionally given a .cu extension, and makes use of both its own, and the host system's native compiler. The resulting host executable may then also contain any compiled CUDA binary objects.

NVIDIA intends CUDA to be used on future GPUs, and this will likely mean that thread counts for the G80 series, already in the thousands, will increase still further. To help manage the addressing of these threads, CUDA incorporates a hierarchical system where up to 512 threads exist within a CUDA software entity known as a block. Both the amount of threads per block and the three-dimensional block shape are user defined, with all block threads given a unique identifier specified in Cartesian coordinates. All threads *within a block* are guaranteed to run within the same multiprocessor, and so can synchronise and update shared memory together.

Each block is itself also uniquely identified by a coordinate in a larger CUDA entity, a grid. As before, the number of blocks per grid, and two-dimensional grid shape are user defined. Threads running in different blocks though cannot communicate through shared memory.

A restricted beta release of the CUDA SDK was made available by NVIDIA in late 2006. This is prior to the availability of compatible NVIDIA G80 series graphics cards. Early CUDA releases included a compiler driver based on Open64; a hardware emulator; a BLAS and FFT host library; documentation; and example code. At this stage, details of the CUDA system were covered by a confidentiality agreement, and so legally the only possible interchange was through direct email communication with NVIDIA.

Since then, a range of hardware has been released, and CUDA has moved from beta to public release; currently version 1.0. CUDA documentation now also includes sizable documentation on the PTX virtual instruction set architecture, and new features have been added to the runtime library, such as atomic instructions. For those concerned at the short-term view of GPGPU solutions, NVIDIA have indicated they intend to support CUDA [2] and the PTX ISA across future hardware development iterations. Perhaps the best indication that NVIDIA is truly serious about GPGPU is its announcement of a third product range, *Tesla*, aimed directly at the HPC market. CUDA is the primary programming system for Tesla.

# Chapter 3

# ANTLR

ANTLR, or as the project is also known, ANother Tool for Language Recognition, is the successor to PCCTS, the Purdue Compiler Construction Tool Set. ANTLR is open source, written in Java, and is primarily the work of Terence Parr.

As hinted at by the ironic resignation in its title, ANTLR is only one implementation of a software mould used in many parser generation tools; from the venerable Lex and YACC pairing, to the contemporary Boost Spirit, or JavaCC. The principle of such *compiler compilers* is that the sourcecode of a parser can be generated automatically from a language's Backus-Naur Form (BNF) grammar description.

The fundamental element upon which a parser operates is a token. Tokens are akin to the words of a natural language sentence, and are output by the lexer, which is also specified using BNF. ANTLR abandons the approach of separating the lexer and parser seen in the Lex/YACC and Flex/Bison project pairings. Instead, an ANTLR grammar file describes both, with lexer rules by convention written in uppercase, and appearing after the parser rules in the file.

One of ANTLR's distinguishing features is that the language of the generated parser code is chosen by the user. However, although the *options* header segment of a grammar allows for the straightforward specification of say C# as a target language, with `language=CSharp;`, the crucial non-BNF embedded code must also be written in the target language. It follows that such a choice will preferably be made by a project once, and early. Java was selected as the target language for the project simply for the rapid prototyping features available when code generated in Java is used together with the ANTLRWorks tool. Java was also perceived as something of a *lead* language for the rapidly evolving ANTLR.

## 3.1 Backus-Naur Form

Backus-Naur Form (BNF) refers to a text-based notation system used for the formal specification of *context-free* grammars describing *predominantly* computer programming languages. Formal representation of a system's expected behaviour is of course vital for safety-critical software, but BNF is also highly convenient as a natural-language agnostic, and parsemonious means of expressing new language concepts, and ultimately, applications.

BNF was devised by Fortran's creator John Backus in the 1950s as a formal method of describing the ALGOL programming language [1]. The initialism em BNF was originally understood as Backus-*Normal* Form, though at the suggestion of Donald Knuth, the middle letter's meaning was subsequently understood to refer instead to the ALGOL report's editor, Peter *N*aur. Peter Naur was also responsible for a significant simplification of BNF's notation.

The following simple BNF example demonstrates one of the dozens of production rules from a C programming language grammar:

```
<expression-statement> ::= ; | <expression> ;
```

Here, as in all BNF rules, the symbol on the left-hand side of the `::=` may be replaced validly by either of the two *alternatives* from the right-hand side. As seen here, BNF symbols are often identifiably surrounded by angle brackets, while alternatives are separated by a `|`. Indicated by its lack of brackets, `;` is a *terminal*, signifying that the character will itself form a piece of the final parsed string. Character strings are also legal terminals. The production rules for symbols listed on the right-hand side, such as **`<expression>`**, would normally be specified elsewhere in a BNF grammar.

In practise, the **`<expression-statement>`** rule will distinguish any valid C expression statement, for example `x ^= y;`, from an *empty* statement comprised solely of a semicolon (as often seen in a for loop's first statement). Therefore `x ^= y;` would be matched to the rightmost of the two alternatives, with the `x ^= y` part, free of the semicolon, analysed further by the **`<expression>`** rule.

### 3.1.1 Extended Backus-Naur Form

Syntactic variations of BNF exist. For example, `::=` may be found written `:=`, or simply `:`. A popular flavour of BNF is Extended BNF. Originally developed by Niklaus Wirth, EBNF incorporates the syntax of regular expressions to more elegantly represent options and repetitions. For example, a C programming language grammar rule to

[1] ALGOL was initially known as the International Algorithmic Language (IAL)

differentiate a variable declared *with* an initialiser, from one declared *without*, could be described in BNF by:

```
<init-declarator> ::= <declarator> = <initialiser> |
                      <declarator>
```

While in EBNF, using square *option* brackets, the need for a formal alternative is removed, to become:

```
<init-declarator> ::= <declarator> ['=' <initialiser>] ;
```

EBNF does not extend the fundamental capabilities of BNF; *any* grammar expressed in EBNF, can also be expressed in BNF. The primary benefit of EBNF is to allow for a more concise and elegant representation of a grammar. EBNF is used by this project to define the grammar of the extended OpenMP-based language discussed previously. It is also employed by the ANTLR parser generator; discussed in the following section.

### 3.1.2 Context-Free Grammars

Extended BNF is only capable of describing *context-free* grammars. Many programming languages however do require some contextual information to describe their correct operational behaviour. In practise this discrepancy is not a problem, as most parser generation programs facilitate the embedding of custom non-EBNF code, placed alongside the BNF grammar syntax.

## 3.2 Top-Down Recursive-Descent Parsing

Although the variations in the BNF *syntax* utilised by an individual parser generator are comparatively unimportant, the algorithm embodied by the resultant parser executable, or *recognizer*[2], is. ANTLR is an *LL* parser generator. The *LL* initialism derives from the fact that the generated recognizer both parses input from *L*eft to right, and constructs a *L*eftmost derivation of that input /footnoteit is equivalent to say either that a recognizer is top-down or that it builds a leftmost derivation. The top-down parsing approach of ANTLR means that the generated code has a closer conceptual mapping to the original BNF grammar; increasing readability, though perhaps with a corresponding performance cost.

[2] ANTLR can produce three types of recognizer; a parser, lexer or tree parser.

The other significant characteristic of a parser is the degree of *lookahead* it employs. The lookahead of a parser specifies the maximum number of tokens which can be examined while attempting to resolve BNF rule alternatives. In the following excerpt from the ANTLR Java grammar,

```
relationalOp
  : ('<' '=' | '>' '=' | '<' | '>')
  ;
```

the parser is required to lookahead by two tokens to distinguish the *Less than* from the *Less than or equal to* operator; or the *Greater than* from the *Greater than or equal to* operator. For efficiency, parsers have traditional been equipped to handle only a small number of *lookahead* tokens; often only one. For example, an *LL* parser with a lookahead of two could be denoted in brief as *LL(2)*[3], and can then be classified as belonging to the larger class of *LL(k)* parsers. While the second version of ANTLR was such an *LL(k)* class parser, the third, *current* version innovates through its implementation of a novel *LL(*)* parsing algorithm. The following C-style variable declaration rule excerpt /footnoteThis example is very simple. For one thing, variable names are restricted to capital letters only. may help illustrate how grammars produced with the new *LL(*)* system can be expressed readably:

```
var_declaration
  : type pointer* ID ';'
  | type pointer* ID '=' NUM ';'
  ;
type    : 'int' | 'char' ;
pointer : '*' ;
ID      : 'A'..'Z'* ;
NUM     : '0'..'9'* ;
```

Here, the location and capitalisation of the **ID** and **NUM** rules signify they belong to the lexer. The kleene star, *****, specifies that a preceding rule or terminal may be matched zero or more times. The critical observation here then is that a production rule such as **var_declaration** could not be used by an *LL(k)* parser generator for any fixed value of *k*, as there is no limit to the number of pointer stars produced by **var_declaration**'s **pointer** rule invocation. This *does not* however imply that any such rule or grammar cannot be refactored into an *LL(k)* form, but clearly outlines ANTLR's approach to making grammars more comprehensible, and their creation straightforward.

Incidentally, whereas YACC and many other parser generators use a *bottom-up*, or *LR*[4]

[3] Parsers with a lookahead of one, often omit the bracketed value, i.e. LR, instead of LR(1)

[4] YACC is more precisely described as an *LALR* parser generator.

approach, parsing from *R*ight to left is far less common; though more so in natural language parsing.

## 3.3 ANTLR Actions

*Actions* is the term used by ANTLR for the code snippets which may be embedded into a grammar's BNF rules. Most fundamentally, ANTLR actions can be used to overcome otherwise intractable BNF ambiguity, or problematic lack of contextual information. The following rule from the ANTLR C Grammar provides a useful example:

```
type_id
  : {isTypeName(input.LT(1).getText())}? IDENTIFIER
  {System.out.println($IDENTIFIER.text+" is a type");}
  ;
```

The actions in this **`type_id`** rule are composed using the Java language and are, as always, enclosed in curly brackets. The first action is a *syntactic predicate*, and allows *contextual* information to help determine a match from an alternative. In this case, the text input stream's next token is provided to a user-defined Java method with boolean return type. A positive response from the **`isTypeName`** method will thus allow ANTLR to proceed to formally match **`type_id`** with the input.

Although the second action merely prints a small message, it demonstrates how elementary it is to access the full text for a matched rule [5]; in this case the *lexer* rule, **`IDENTIFIER`**. Should the built-in **`<rule>.text`** field of a matched *parser* rule be accessed, the *entire text* matched by it, and all its subrules is provided. Such data will be absolutely essential for the project's parse tree modification, to be discussed.

## 3.4 ANTLRWorks

ANTLRWorks is a useful application providing a complementary GUI to ANTLR's command-line interface. From within the ANTLRWorks Integrated Development Environment, a grammar may be edited, debugged, and visualised. The debugger in particular offers an impressive range of features, such as the ability to step-debug, both forward and in reverse, through an ANTLR EBNF *grammar*; as opposed to through the generated recognizer sourcecode. Although ANTLRWorks is immature, and has many bugs and restrictions, it is nevertheless a fantastic and progressive tool; well used during the project.

[5] Note that the println action will only execute upon a successful match of **`type_id`**.

## 3.5 ANTLR TokenRewriteStream

Upon the command line invocation, **`java org.antlr.Tool omp.g`**, a number of java files are created, the most prominent being the parser and lexer; in this case ompParser.java and ompLexer.java. In addition to the *generated* files, parsing of an input file also requires a hand-built control harness, or *main* file. A minimal **`main`** method from such a file is shown here:

```
public static void main(String args[]) throws Exception {
  ompLexer lex = new ompLexer(
                 new ANTLRFileStream(args[0]));
  TokenStream tokens = new CommonTokenStream(lex);
  ompParser parser = new ompParser(tokens);

  try {
    parser.translation_unit();
  } catch (RecognitionException e) {
    e.printStackTrace();
  }

  System.out.println(tokens);
}
```

This simple parser program demonstrates the use of the generated ompParser and ompLexer Java classes, and also of the **`CommonTokenStream`** class; the only class which was not generated. The parsed and unmodified input is then returned as output to the **`System.out.println`** method. The **`CommonTokenStream`** class, part of the **`org.antlr.runtime`** package, represents the most common stream of lexer output tokens which feed an ANTLR parser. The practicable **`TokenRewriteStream`** class inherits from the **`CommonTokenStream`** and is intended for use when manipulation of a parsed input stream is to be performed, before returning the altered stream to the caller.

The utility of **`TokenRewriteStream`** forms a cornerstone in the projects's transformation of OpenMP C into CUDA C. To illustrate this, the **`openmp_construct`** rule from the combined grammar is modified below by the inclusion of a small piece of ANTLR action code:

```
openmp_construct
@after{
  ((TokenRewriteStream)input).replace(
    $openmp_construct.start,
    $openmp_construct.stop,
    "// This text replaces an OpenMP Construct.");
}
  : parallel_construct
  | for_construct
  | sections_construct
  | single_construct
  | parallel_for_construct
  | parallel_sections_construct
  | master_construct
  | critical_construct
  | atomic_construct
  | ordered_construct
  ;
```

An **@after** action such as that used above, is executed just before a rule returns [6]. The result of running this code on an input is that all OpenMP constructs found there, are replaced by the quoted text.

The code from the **main** method must also be lightly modified to support such techniques; thus **TokenRewriteStream** is used instead of **CommonTokenStream**.

In practise, a new **TokenRewriteStream** member variable is assigned to the **input** object in the grammar's initialisation stage. This avoids always casting the **input** object at its numerous references.

Unfortunately the interactive debugger of ANTLRWorks cannot handle grammars making such use of **TokenRewriteStream**. Debugging was from this point accomplished using the NetBeans IDE, operating both on custom-built, project support classes, and the ANTLR-generated java files.

[6] The **@init** action similarly executes *before* all other rule actions.

# Chapter 4

# The NVIDIA CUDA C Programming Language

As CUDA C is the back-end target for the compiler, the next few sections will briefly present some of the syntax and semantics of the CUDA programming style.

## 4.1 The Driver and Runtime API

CUDA provides both a *driver* and a *runtime* API. The runtime API is built upon the lower-level driver API. The benefits of the driver API over the runtime API include a marginally quicker initialisation phase per GPU kernel, and early developer access to additional features, such as 16-bit floating point values. The crucial distinction for the project at hand though was the driver API's lack of support in the emulator. Although not a simulator, step debugging and **`printf`** calls are often extrememly useful. Only one API can be used at once, and so the runtime API was selected.

## 4.2 Built-in Variables

Like the OpenMP function **`omp_get_thread_num()`**, CUDA provides a built-in *variable*, **`threadIdx`**, which can be used to obtain an unsigned integer value identifying the curious thread. A single GPU has up to 128 processors, however NVIDIA strongly recommend the use of multiple threads per processor. The thread management required for this is provided by an NVIDIA technology known as *GigaThreads*, and is intended to facilitate mitigation of the proportionately high main memory access time. With one eye also on future developments, this explains why **`threadIdx`** is actually a 3-tuple, with **`threadIdx.x`**, **`threadIdx.y`** and **`threadIdx.z`** together identifying a thread uniquely, but *only within its own block*.

Each CUDA thread is part of a new construct known as a *block*. Typically, many blocks will be launched by a CUDA kernel. Though a block may be swapped in and out of active execution by the scheduler, each block will run on only one of the multiprocessors. This allows the threads of each block to efficiently manipulate their own portion of fast, *shared* memory; optionally in a synchronised fashion.

Another built-in variable, **`blockIdx`**, identifies each block. Also a 3-tuple, **`blockIdx`** can be used together with **`threadIdx`** to provide a *globally unique* thread identifier.

Two further built-in 3-tuple variables, **`blockDim`** and **`gridDim`** are analagous to the OpenMP **`omp_get_num_threads`**, and complete the set. The former gives the extent of threads in each dimension of a block, while the latter gives the extent of blocks in another new construct, the *grid*. Only one grid is in execution at once. The product of the number of threads in a block, by the number of blocks in a grid is therefore the total number of threads in execution at one time.

## 4.3 Syntax Overview

Excepting the limitations discussed later, a C function intended for execution on the GPU need only be preceded by the CUDA function qualifier, **`__device__`**. If the function has been designed to execute on the host system *and* the GPU, then **`__host__`** must also be added[7]. The following function declaration illustrates the use of both qualifiers together:

```
__host__ __device__
unsigned int LCG(unsigned int a);
```

The **`__global__`** qualifier distinguishes a function which is called from the host, but executes on the device. Only one such call can be active simultaneously, and so represents the individual entry point of a GPU kernel. **`__global__`** function calls are asynchronous; so permitting concurrent processing on the CPU and GPU.

Any call to a **`__global__`** function must also specify an *execution configuration*. The following **`__global__`** function,

```
__global__
void Kernel_1(float *pGlobalData);
```

has its execution configuration specified by the following call syntax:

[7] **`__host__`** is optional on a function intended *solely* for the host system.

```
Kernel_1<<<Dg, Db, Ns>>>(pGlobalData);
```

The first two parameters, **Dg** and **Db**, specify the grid and block dimensions, and so determine the number of GPU threads to launch. The third parameter is optional and specifies in bytes the amount of shared memory allocated per block. This is the only way to dynamically allocate shared memory. Requesting too much shared memory will result in a kernel launch failure.

## 4.4 Shared Memory and Registers

There are 8192 32-bit registers, and 16KB of shared memory on each multiprocessor; shared memory is accessible to all threads in a block. Variables created in shared memory are declared with the **__shared__** qualifier preceding the type. Variables declared with automatic scope, e.g. `int i;` are allocated to registers. Alternatively, if no registers are available, automatic variables are instead allocated in a region of main memory known as *local* memory.

The following example code updates data stored in main memory, referenced by formal parameter, **pGlobalData**. The data is a factor of **NUM_BLOCKS** larger than the **__shared__** array, **shrd**. Each thread updates the **threadIdx.x** element of **shrd** before updating the **pGlobalData** element corresponding to its **tid** value. The block and grid dimensions are assumed to be one dimensional; which is to say, their **y** and **z** fields are 1.

```
__global__
void WriteID(int *pGlobalData)
{
  __shared__ int shrd[NUM_THREADS_PER_BLOCK];
  unsigned int tid = threadIdx.x +
                     (blockIdx.x * blockDim.x);

  shrd[threadIdx.x] = tid;
  pGlobalData[tid] = shrd[threadIdx.x];
}
```

## 4.5 Native Synchronisation

Threads within a block have the opportunity to synchronise their execution using the CUDA barrier, texttt__syncthreads[8]. As well as providing a means to avoid race conditions, **__syncthreads** is also used by the compiler to coallesce reads and writes to main memory into a single bulk transfer. The developer incorporating **__syncthreads** in this way can use the call to signify that a lengthy read or write request has been issued; allowing the scheduler to accept the call as a *yield* instruction. The following code illustrates a trivial update of the previous code which, by careful placement of **__syncthreads**, should run more efficiently.

```
__global__
void WriteID(int *pGlobalData)
{
  __shared__ int shrd[NUM_THREADS_PER_BLOCK];
  unsigned int tid = threadIdx.x +
                     (blockIdx.x * blockDim.x);

  shrd[threadIdx.x] = tid;
  __syncthreads();
  pGlobalData[tid] = shrd[threadIdx.x];
}
```

A variation on this example is seen in many CUDA example codes. This can briefly be outlined as as:

1. Issue a read request from main memory to shared memory
2. Insert a call to **__syncthreads()**
3. Perform calculations
4. Issue a write request from shared memory to main memory
5. Insert a call to **__syncthreads()**

[8] **__syncthreads** is a subset barrier, released when all threads of a block arrive.

## 4.6 CUDA Language Restrictions

On cursory examination, CUDA does appear similar to C, or C++. However, there are significant differences which limits the choice of back-end implementation strategies. The following lists the most significant CUDA restrictions; all apply to device, rather than host code:

- No function pointers
- No double-precision floating point values
- No **`static`** storage class
- All functions are inlined
- No function recursion

It was also significant for the project that built-in CUDA variables can only be used within functions targeted solely at the GPU. For example, a function qualified with both **`__device__`** and **`__host__`** could not reference **`threadIdx`**. Unfortunately, conditional preprocessing, with macros such as **`__CUDACC__`** or **`__CUDABE__`** cannot overcome this limitation.

# Chapter 5

# Memory Management

The decision to execute only the OpenMP **`parallel`** constructs on the device, and to update the device memory exclusively outside of those constructs is due to the physical layout, and price to performance ratio of memory. This chapter considers how the presence of a dual address space memory hierarchy affects the compiler development. Such a memory architecture also has significant effect on the use of C pointers when provided as list items to OpenMP constructs.

## 5.1 Placement of Parallel Regions

The latency and bandwidth of data storage is directly proportional to its cost. This is reflected in the omnipresent storage hierarchy beginning with a processor's registers and caches, on to system memory, harddisks, network storage, and finally remote archival facilities. It is also true that historically the price/performance ratio of memory modules has not matched that of processors. Performance aware programs targeting the GPU will reflect this in their data access patterns.

The GPU has access to its registers, caches, and the card's system memory. Should a developer require further GPU storage capacity, the host system, with attendant persistent storage must be used. Transfer of data between the host system and graphics card uses the PCI Express x16 interface on the motherboard; between the two system memories. This is currently performed by the host using a blocking function such as **`cudaMemcpy`**, which occurs *outside* of the execution of a GPU kernel program. The GPU's main memory can therefore be seen as a bottleneck, not so much for its capacity, which can now reach 1.5GB, but for the state restriction on when additional persistent storage serialisation can be performed by the attendant host.

With this in mind, implementation of an OpenMP parallel region on the GPU begins with the assumption that the graphics card's system memory will be sufficient to accommodate its execution. An alternative approach for a compiler would involve *splitting* a data-hungry kernel to permit the update of device memory buffers by the host. The

negative performance implications of that approach make it advisable to recommend refactoring the code to utilise more than one parallel region. Engineering such a compilation result would also be cumbersome.

Asynchronous data transfer between host and device during CUDA kernel execution is not currently permitted. This is anticipated to be included in future CUDA release. Such a development would open the possibility of implementing currently unavailable systems routines which rely on IO. It is encouraging to observe within the last few months that the call to the single active CUDA device kernel has become asynchronous; allowing the CUDA host thread to perform other tasks in parallel. For the moment though a kernel has a limit of 1.5GB of device memory.

## 5.2 Two Memory Subsystems

To ensure the correct operation of an OpenMP **`parallel`** region, each variable and function referenced within it must have at least one twin existing on the graphics card. Depending on the sharing attributes specified, variables may also require initialisation and multiple instantiation. The following code excerpt will help illustrate the steps required to handle some common data clauses.

```
int a = 1, b = 2, c = 3, d = 4, e;
#pragma omp parallel for shared(b) firstprivate(c,d)\
                         lastprivate(d) private(e)
{
  d *= omp_get_thread_num();
  e = c + d;
  #pragma omp critical
  {
    a += e;
    b += a;
  }
}
```

The **`shared`** data clause requires that a variable should maintain its contents across both entry and exit of the parallel construct. In this example, the **`shared`** attribute is assigned to two variables, **`a`** and **`b`**; with **`a`** assigned as **`shared`** by default. At the start of the **`parallel`** construct, the two variables must firstly have memory allocated within the graphics card's memory; so creating their twins. Their values are then copied into the corresponding twinned integers on the card's memory; perhaps using **`cudaMemcpy`**. At this stage, **`a`** and **`b`** are suitably initialised. Upon completion of the GPU kernel, and thus also on exit from the OpenMP **`parallel`** region, their final values must once again be copied across the PCI Express bus, this time from the device to the host.

The remaining variables, **c**, **d** and **e** all require multiple twins on the card, as their association with the **private** data clause requires a separate instance for each thread. Of these, the values of **firstprivate** variables **c** and **d** are used to initialise each thread's copy at the start of the **parallel** region. Only **lastprivate** variable **d** is required to update the original host variable upon completion of the kernel.

Note that both the **for** and **sections** constructs can be used to alter the sharing attribute of a list item already assigned as **shared**, to **private**. As there are no dynamic memory allocation facilities available on the device, this implies that items reassigned in such a fashion must have their memory pre-allocated at the start of each region. It is also worth noting that as NVIDIA recommends the use of considerably more than one thread per processor, the **private** clause will have a disproportionately large impact on memory resources.

Choosing the **parallel** region boundaries as a trigger for the transfer of data required by each OpenMP region is a practical choice. Alternatively, it might also be possible to update the twinned variables on the graphics card's on an ongoing basis throughout the serial phase of each program. However the update of larger datasets may produce suprising and offputing performance spikes for the user, and would require significant additional runtime bookkeeping. There is already an assumed startup cost upon entry to an OpenMP **parallel** region. It is advisable that any costs related to device preparation remain conceptualised by a user in this location.

## 5.3 Pointers

A pointer variable included as a **shared** or **firstprivate** list item to an OpenMP **parallel** region running on a conventional SMP can initialise each thread's private pointer variable by duplicating the memory address content of the original list item. However, as memory addresses on the graphics card's memory have no relationship to those of the host, pointer variables, and their target datasets, require special handling.

An OpenMP **parallel** construct with an integer pointer list item having the **shared** data attribute, is a simple example worth contemplating. If we could assume that the pointer will dereference only a scalar value such as an integer, there would be less difficulty: device memory to contain the integer target could be allocated, the host integer's value transferred to the device, and the new device integer's address used to initialise the corresponding device pointer under consideration. There is though considerable flexibility in the use of pointers in C. It is equally likely that our pointer contains the address of a single integer element from an array of integers[9]. No longer then is it possible to transfer a single integer to the device; and yet the pointer alone cannot be used to obtain the size of the array for transfer. It becomes necessary to examine the size and address of all known variables.

[9]The use of array pointers is underused in C. e.g. **double (*p)[10];**

If a list containing all variable addresses and size pairs were available, it is certain that a pointer's target dataset could ultimately be identified and with due diligence, copied across to device memory. Indeed, bearing in mind that the translation unit will have been parsed, it is fully reasonable to assume that a list of all global, local, and function formal parameter variable addresses, and their sizes, which are in scope at the **`parallel`** construct, can readily be created. A compiler-generated address comparator, placed immediately prior to the GPU kernel, could then facilitate the necessary range checks resulting in the identification of the target memory region. Functions such as **`gpump_alloc_list_add`** for accessing this list are discussed in chapter 5. However, there are other memory-resident objects which would escape such a register.

## 5.4 Further Indirection

Arbitrary levels of pointer indirection have been implemented. The following simple example illustrates a scenario involving two levels of indirection; a pointer to a pointer.

```
void retarget()
{
  float **pp;
  float *p;
  float old = 0.0f, new = 1.0f;

  p = &old;
  pp = &p;

  #pragma omp parallel shared(pp, new)
  {
    #pragma omp critical
    *pp = &new;
  }
}
```

Although within the above code the pointer variable **`p`** is never referred to by name within the **`parallel`** region, the pointer value it *contains*, is updated there. This unnamed variable must be created on the device. Upon exit from the **`parallel`** region, it must also update the original copy of **`p`**; through **`pp`**. Stated simply, *no more than* one variable must exist on the device for every level of indirection present in any pointer variables listed in a **`parallel`** region's data clauses. Optimally, less may be required. For example if only **`pp`** itself had been updated in the **`retarget`**'s **`parallel`** region. It is also observed that such variables may in fact themselves be arrays; for example **`p`** could be an array.

## 5.5 Stack Allocated Variables

In the following code example, an integer variable, **d**, with automatic storage, is declared in a function, **prepare**, before its address is used to initialise the integer pointer variable, **pd**; itself then passed to a second function, **execute**.

```
void prepare()
{
  int d[2] = {-1, -1};
  int *pd = &d;

  execute(pd);
}

void execute(int *p)
{
  #pragma omp parallel num_threads(2) shared(p)
  {
    int tid = omp_get_thread_num();
    p[tid] = tid; /* host must prepare p's data */
  }
}
```

Notwithstanding the possibility that **execute** is itself free to declare another integer variable named **d** prior to the **parallel** construct, the **prepare** function's local array **d** is, at the **parallel** construct, simply out of scope, and therefore not amenable to the discussed pointer tracking registry strategy.

## 5.6 Runtime Stack Access

The GCC compiler provides a number of extensions to the C language. Two such functions, **__builtin_return_address** and **__builtin_frame_address** provide the return or frame address of a function. Both accept a single parameter; a constant unsigned integer specifying the frame of the call stack relative to the callee. It is thus possible to traverse the call stack's frames, including the out-of-scope, stack-allocated variables under investigation. Though the documentation for these functions warns that restrictions apply on an unspecified class of machines, both functions are actually straightforward to use, and did work as described on the project machine; with the exception that only string-literals were accepted as function arguments.

In practise however, the stack data retrieved by such methods is absent of crucial semantic information. The most significant unknown is the size of each stack variable.

For example, assuming that a pointer supplied to an OpenMP data clause has been identified as a stack variable, and that the name of the function declaring said variable has also been obtained[10], then knowledge gained from the parser could inform us of the types and sizes of that function's local variables. However it cannot tell us about the order the compiler has stored them in. So, an integer pointer may be pointing *either* to an scalar integer variable, or any element of a neighbouring integer array.

Further attempts to eliminate uncertainty via this route of investigation run the risk of losing portability; as we obtain further information about only those compilers providing such details, and those we have access to. The limitations of this inquiry, which can perhaps be seen as an *inside* approach, are due to the distance maintained from the compiler source code. The GNU debugger could also be employed, yet through language extension there are still more acquiescent and portable solutions which remain *outside* of the compiler's aegis.

## 5.7 Dynamic Memory Allocation

A second class of pointer challenges emerge from the use of dynamic memory allocation routines. Memory allocated dynamically remains valid regardless of whether or not the function requesting the allocation has returned; though this is not the problematic aspect. Static code analysis, built upon the project's C parser, cannot determine the quantity of memory requested or received. An acute illustration of this would be to imagine the second parameter of the standard **`malloc`** function, accepting the output of a random number generation routine.

The GNU version of the standard C library has many novel and interesting features. One such feature set is referred to as 'Memory Allocation Hooks'. By assigning special predefined function pointer variables, such as **`__malloc_hook`**, to custom memory allocation routines, it is relatively simple to track all uses of **`malloc`**. For example, **`printf`** could be used to print the size and address of every memory block obtained through a call to **`malloc`**.

Complementing this, many UNIX installations also support the use of an environment variable, **`LD_PRELOAD`** which informs the linker to use the libraries listed there before all others; for programs using shared libraries. It is therefore possible, using the memory allocation hooks, to supply a rudimentary **`malloc`** wrapper library, that merely registers the parameters of successful calls to the **`malloc`** family of functions for subsequent use, as before, tracking pointer targets. By using **`LD_PRELOAD`**, this system can usefully be applied to routines called by both user *and* third-party code.

A project trial of such a scheme revealed a suprising quantity of memory allocation calls from within the standard C libraries. Upon further analysis, many such allocations might optimally be classified as irrelevant in this context, however it soon became clear that it is actually in principle rather than practise that this approach has limitations.

[10]The GNU C library function **`backtrace_symbols`** can be used.

First of all, overriding the **`malloc`** family is only useful if this is actually the method of memory allocation employed. It is entirely possible that a developer will use a custom memory allocation routine. Secondly, as mentioned earlier, the system fails to penetrate library functions which have been linked statically; unless of course the sourcecode is available.

On reflection it becomes apparent that a more tractable approach refrains from the methodology of a detective; whereby an understanding is sought by examining allocation clues left behind by a user. Rather, a more collaborative approach is proffered, wherein a specified programming style, adhered to by a developer is honoured by the safe execution of the resulting program. This approach will incorporate new memory allocation functions, and also extension of the set of OpenMP constructs. This is discussed fully in the following chapter on language design.

# Chapter 6

# Language Design

The dual address space memory architecture targeted by the project's OpenMP implementation is a characteristic fundamentally different to that found in traditional Symmetric Multi-Processing (SMP) OpenMP configurations. Another important distinction is the lack of many standard C library functions on the device; primarily those relating to I/O, but also of memory allocation. In this chapter, a handful of extensions to the set of OpenMP directives and runtime library routines are described. Such extensions aim to minimise user effort in porting code, allowing cross-platform transformation of OpenMP C code to NVIDIA CUDA C, while operating outside of the internal mechanisms of both the CUDA and host system compiler.

## 6.1 Accessible Memory

Variables declared within the sequential part of a code, which are subsequently to be read from or written to within an OpenMP **`parallel`** region, must be identified by the use of a new declarative directive, **`accessible`**. The syntax of the **`accessible`** directive [11] is as follows:

---

**`#pragma gpump accessible(`** *list***`)`** *new-line*

---

The **`accessible`** directive should be placed lexically after the relevant variable declarations, but prior to the **`parallel`** region using it. As *list* can contain more than one variable, it is possible to specify a number of **`accessible`** variables within a single **`accessible`** directive. Only variables which are accessed using pointer indirection need be declared **`accessible`**.

[11] The **`accessible`** directive uses the **`gpump`** string instead of **`omp`** as an unknown OpenMP pragma causes an error on a conventional OpenMP compiler.

The following sample code listing illustrates how the **a1** function, from the current OpenMP specification's *Simple Parallel Loop* example[1], can be called from another function **call_a1**. The new directive allows the runtime library to identify that the **a** and **b** pointer variables, which default to having the **shared** attribute, are pointing to the start of two separate blocks of data, each of size: **1024 * sizeof(float)**.

```
void a1(int n, float *a, float *b)
{
  int i;

#pragma omp parallel for
  for (i=1; i<n; i++) /* i is private by default */
    b[i] = (a[i] + a[i-1]) / 2.0;
}

int call_a1()
{
  float d1[1024], d2[1024];
#pragma gpump accessible(d1,d2)

  a1(1024, d1, d2);
}
```

During code translation, the **accessible** directive is used to inform the compiler of the parameters and insertion site for a **gpump_alloc_list_add**[12] function call. At the end of every bracket-delimited scope, the list items should automatically be removed from the memory object list by the compiler; by insertion of matching calls to **gpump_alloc_list_remove**.

When the **accessible** directive is used with variable declarations, the semantics are similar to the **sharable** directive from Intel's Cluster OpenMP[5]. With Cluster OpenMP, data accessed by more than one thread must be declared **sharable**. The **accessible** directive distinguishes itself from **sharable** through its secondary use with function types, described below.

The **accessible** directive directly addresses the absence of runtime information on stack-declared variable allocations. For consistency it may be recommended that **accessible** be used to identify global variables also; although this is currently lacking in implementation. Such a technical necessity for global variables would likely disappear as the system matures, or evolve solely into a runtime optimisation.

[12] See chapter 7 for more discussion of the **accessible** implementation.

## 6.2 Customising Malloc

Memory allocated dynamically that is required within a **`parallel`** region must also be tallied to enable runtime pointer resolution, address translations, and host to device data transfer. Although dynamic memory allocations are unaffected by scope, the value of parameters such as **`malloc`**'s **`size`** parameter often cannot be determined through static code analysis. Additionally, although more relevant for error detection, dynamic memory may also have been freed before a subsequent **`parallel`** region is encountered.

Each of the four most common dynamic memory allocation routines, **`malloc`**, **`calloc`**, **`realloc`**, and **`free`**, have therefore each been wrapped using one of four new host functions; **`gpump_malloc`**, **`gpump_calloc`**, **`gpump_realloc`** and **`gpump_free`**. The new routines accept the same formal parameters, and return the same values as those over whom they wrap. Should a region of dynamically allocated memory be accessed within a **`parallel`** region, these functions should be used in preference to other methods for dynamic allocations in the preceding sequential part of the code.

Internally, each of the four new functions themselves call the corresponding C library allocation function. Should that call be successful, the runtime library's internal list of valid and monitored memory regions is updated; whether by addition, removal or modification. By leaving the **`malloc`** family untouched internally, the user remains free to provide custom memory allocation through memory allocation hooks such as **`__malloc_hook`**.

## 6.3 Function Decoration

In common with the present project, cluster implementations of OpenMP have an obligation of care concerning the movement of data required on both sides of the conceptual **`parallel`** region boundary. Cluster versions of OpenMP though do not encounter an architectural distinction between host and device hardware. Consequently, all standard C library functions remain available in both sequential and parallel sections. CUDA programs targeting the graphics card do not have comparable provisions.

As a consequence, it is impractical to approach a translation unit's functions with the assumption that every one should compile for both the host and the device. The presence of an I/O function call such as **`printf`**; a common routine such as **`strlen`**; or any function available in binary form only, will not compile for the device. Although it would be a technical solution to request that only functions targeting both architectures be included in a translation unit, this places an unwelcome burden upon the user, and existing codes may require to be substantially refactored.

Functions which are required within a **`parallel`** region must therefore first be identified in the code by the user. The **`accessible`** directive is utilised here once again, with the following syntax:

**#pragma gpump accessible** *new-line*
*function-definition*

CUDA itself provides function type qualifiers, **__device__**, **__global__** and **__host__**, which may be used singly, or in combination to specify where a function will execute, and from where it is called. For the project's purposes, the **#pragma** syntax is preferred over such qualifiers, not only for consistency, but also for the more portable nature of the **#pragma** preprocessor directives. Compilers unware of the meaning of a particular directive can safely ignore it. To put it another way, the proposed language design will profitably remain legal C.

CUDA function qualifiers are though used in the compiled output. Any routine identified as **accessible** is transformed into two separate functions. The first of these is identical to the original; except preceded with an optional CUDA **__host__** qualifier. The second new function, targeting the device, has its name appended with **_d**, as do all function call references made within its body. The CUDA qualifier **__device__** must precede this generated function.

## 6.4 Limitations

The language extensions described up to now are intended to be a minimal intrusion on the OpenMP semantics. One goal has been to ensure that a user familiar with the use of OpenMP and the C language is presented with a reasonably familiar environment. Regarding indirection, a pointer targeting an array is honoured with the full array copied to the device prior to kernel execution; and not merely a scalar value. This is performed without explicit transfer and variable management by the user.

Without further information from the host compiler, assumptions about the device memory's intended state will not always be correct. The current system does not track the *types* of the memory blocks identified to **accessible**. This behaviour compares with the fact that **malloc** also has no understanding of types. This can though present a problem when an OpenMP pointer list item references a user-defined structure which itself contains pointers.

To illustrate how this problem manifests in the current implementation, consider the following code:

```
typedef struct _new_type {
  float *p;
  int secs;
} new_type;

void caller()
{
  new_type x;

  x.p = 0;
  x.secs = 31556926;

  callee(&x.p);
}

void callee(float **pp)
{
  #pragma omp parallel shared(pp)
  {
    #pragma omp master
    *((int *)pp+1) += 1;
  }
}
```

Having a pointer type as a list item, the translated source code will at runtime consult the known allocations list; updated by the **`accessible`** directive. The *pointer* pointer **`pp`** should then be found to target the first of the two 32-bit values comprising the **`new_type`** type.From the context, our compiler then makes the incorrect heuristic decision that both of the 32-bit **`accessible`** values are pointers. Thus, on transfer to the device, the value of **`31556926`** has a possibility, when perceived as pointer, of inadvertently targeting a valid allocation, and being converted to the device address space; so destroying the original value. Although the example code's pointer manipulation makes presumptions about data sizes and orders, such free use of pointers is not uncommon within C codes.

A proposed solution would involve identifying both the size, type, and the byte offset of each user-defined structure member, relative to the structure's base address. This can be referred to as *identifying* the type. As the project compiler has no knowledge of the size of basic C types, nor of the arrangement in memory of each structure member, this identification could only be completed at runtime.

# Chapter 7

# Front End - Parsing

Before new code can be generated, the *source* code must first be analysed and in some sense, *understood*. Parsing is a crucial process here, wherein expectations about the arrangement of source code *sentences* are compared to a formal list of rules; a grammar. Parsing alone though forms only the skeleton upon which the ANTLR actions can operate. Together they can allow the effective aggregation of type information, essential to the subsequent back-end code generation phase. This chapter demostrates how such elements were used to prepare OpenMP code for translation.

## 7.1 BNF Language Grammars

The provenance and modifications applied to the different BNF grammars used to construct the OpenMP parser are now described.

### 7.1.1 Base Language Considerations

C and C++ both support OpenMP, though the project only requires one base language. There follows a brief outline of the considerations leading to the choice of one.

Although the *C++* programming language offers many valuable features over its predecessor, C remains popular, particularly within the HPC community. C++ is also regarded as a challenging language to parse[13]. As the compiler's back-end, CUDA is neither a complete implementation of C nor C++, and so currently pulls in neither direction. It is believed that proving the concept using C++ would not be a significantly more potent outcome than with C. Bearing in mind the short duration of the project, the extra effort required to handle C++ could not then be justified. C was therefore chosen as the language upon which to build the project's OpenMP compiler.

[13]Some have even suggested that an alternative syntax, more amenable to parsing [6] be considered for C++.

### 7.1.2 The C Grammar

The current distribution of ANTLR includes a version of Jutta Degener's ANSI C Yacc grammar, modified for ANTLR by Terence Parr. A minimal symbol table is included which, uses ANTLR *actions* to provide contextual information to the syntactic predicates querying for new variable types; possibly defined by prior invocations of **`typedef`**.

For the majority of the grammar, a lookahead of two is required, and is specified in the *options* grammar's header section as **`k=2;`**. Setting **`k`** explicitly like this is an optimisation, allowing for the generation of a faster parser, with less generated code. Individial rules can themselves also assign a temporary lookahead value. In this case the lookahead is increased to three for the **`enum_specifier`** and **`struct_or_union_specifier`** rules to allow disambiguation.

The lookahead is also decreased on two occasions. The first prevents problematic recursion in the **`external_declaration`** rule. The second overcomes an inherent ambiguity in the **`if-else`** construct. For example:

```
if (A) if (B) do_this(); else do_that();
```

could be interpreted as either of the following options:

```
/* Alternative 1 */
if (A) {
  if (B) do_this(); else do_that();
}

/* Alternative 2 */
if (A) {
  if (B) do_this();
}
else do_that();
```

Modifying the lookahead leads to the essentially arbitrary choice taken by most compilers: the first of the two alternatives is taken.

For the project's purposes the grammar was an adequate platform upon which to build the OpenMP parser. Only one *bug*, subsequently confirmed, was identified and corrected: An erroneous rule alternative, **`'*' IDENTIFIER`** of **`postfix_expression`**, caught with an input such as **`a[2*b];`**, was unrequired, and so removed; presumably a typographical error.

### 7.1.3 OpenMP Grammar

The OpenMP Application Program Interface Version 2.5 specification document has a grammar appendix which lists the extensions to the C and C++ language grammars[14] required to parse the directives of OpenMP. The notation employed[15] to encode this grammar is similar in appearance to a collection of BNF rules. To allow this collection to join and function together with the C grammar described previously, the OpenMP and C grammar rules must be included in the same file. This required that the OpenMP rules were first converted to a form compatible with ANTLR. The process was a straightforward procedure, and the steps involved are itemised below:

- Single quotes were inserted around terminal strings such as **`num_threads`**
- Hyphens in rule names were replaced with underscores
- References to **`new-line`** were removed
- Instead of placing rule alternatives on separate lines, the **`|`** symbol was used
- A terminating semicolon was added to each rule
- A new rule, **`openmp_pragma`** was created to match: **`# pragma omp`**
- The existing **`identifier_list_rule`** from the C grammar was used instead of **`variable-list`**
- Rule recursion was replaced by ANTLR's unary operators; **`*`**, **`+`** and **`?`**
- The **`?`** operator replaces the subscript of every optional <rule>$_{opt}$ reference
- For every optional rule *sequence* identified as <rule>$_{optseq}$, a new rule was created using the original rule name, appended with _optseq

#### OpenMP Entry Points

The remaining task was to determine where in the C grammar, the newly included OpenMP rules could be matched. Using the OpenMP specification grammar as reference, those rules which are declared there, but never referenced were identified. Three *transitional* C grammar rules, or OpenMP entry points, were thus found. The following table pairs them with the OpenMP grammar rule names which appear in their definition:

| C Rule | OpenMP Rule |
|---|---|
| `declaration` | `threadprivate_directive` |
| `statement` | `openmp_construct` |
| `statement_list` | `openmp_directive` |

[14]The C90 version of the OpenMP grammar rules were used.
[15]The notation is identified as that described in Section 6.1 of the C standard.

The existing C grammar **declaration** rule had two alternatives: a typical variable declaration, and one instead preceded by a **typedef**. The **threadprivate_directive** is supported by simply including it as a third alternative. The **openmp_construct** rule can similarly be matched alongside the six existent alternative statement types within the C **statement** rule.

The OpenMP specification cautions that as the **flush** and **barrier** constructs do "not have a C language statement as part of their syntax"[1], care should be taken with their placement. Example A.20 from the specification demonstrates how an OpenMP **barrier** or **flush** directive cannot be used as the immediate substatement of an **if** statement. **openmp_directive** is therefore not added, as **openmp_construct** was, to the list of C **statement** rule alternatives. It is instead included within the **statement_list** rule. While this would appear to classify it as a statement, examination of the C grammar reveals that **statement_list** is crucially then matched only within the **compound_statement** rule. As a compound statement is always bounded by a pair of braces, and can minimally exist as that alone, it follows that this is a safe location to match these two *standalone* directives. The **statement_list** rule was therefore modified in accordance to this; matching one or more of, **statement** *or* **openmp_directive**, as shown:

```
statement_list
  : ( openmp_directive // OpenMP entrypoint 3 of 3
  | statement
  )+
  ;
```

### 7.1.4 Extensions to the OpenMP Grammar

The two **accessible** extentions to OpenMP are facilitated by the addition of two *new* rules, and the modification of three existent in the ANSI C grammar rule set. The first of the new rules recognises the new **#pragma gpump** token pair.

Any compiler should be able to ignore the new **#pragma omp accessible** extensions. However, an OpenMP compiler will not accept the **accessible** keyword as it is not a part of the OpenMP standard. Consequently the **gpump** string is used instead, allowing OpenMP compilers to safely ignore the new directives, while remaining in the C language. The following ANTLR rule was created to match the new pragma,

```
gpump_pragma
  : ( '#pragma' | '#' 'pragma' ) 'gpump'
  ;
```

The use of **accessible** as a data declarative directive is constructed similarly to the rule for OpenMP's only declarative directive, **threadprivate**. This second new rule, **accessible_directive**, is also matched alongside the **threadprivate_directive**, as a fourth alternative within the C grammar **statement** rule. The **accessible_directive** is shown below:

```
accessible_directive
  : gpump_pragma 'accessible' '(' identifier_list ')'
  ;
```

When the **accessible** declarative directive is used in its second incarnation, as a qualifier for functions targeting the device, the C grammar rule, **function_definition** can be modified to identify this new matching. The ANTLR **?** symbol identifies elements which either occur once, or not at all. When used to precede a function definition, the **#pragma gpump** terminal set, are just such an collection. The **function_definition** rule is thus modified to the form shown in Appendix B. **external_declaration** must likewise be modified in support of **function_definition**.

## 7.2 Tree Representations

Abstract syntax trees (AST) are a common program representation scheme. An AST is created from a parse tree, and can eliminate unnecessary parse information; for example the semicolons at the end of statements. When represented visually, an AST is clearly less dense than the equivalent parse tree, and is more amenable to extensive, multiple pass transformations; such as within a low-level compiler.

An alternative syntax representation is the parse tree; also known as a *concrete* syntax tree. The parse tree is an acyclic graph representing the entire set of rules encountered during a parse; with leaves of the tree representing the terminals. To illustrate why a parse tree can obtain quite so many nodes, an example is proposed. To match the integer literal, **9** in the following file-scope expression,

```
int x = 9;
```

the C parser, having *already* identified that this is likely a match for the **initializer** rule, traverses a further *sixteen* rules before reaching the terminal; so adding a similar number of nodes to the parse tree in the process.

The parse tree was chosen as the alteration medium for the project. Although operating directly upon the parse tree limits the project's translation to the C source language, such manipulation is straightforward, and sufficiently powerful for the code translation

patterns needed. Within ANTLR, this parse tree is created implicitly as the OpenMP parser traverses the program code, and its *verbose* nature is not an issue. Parse tree manipulation is accomplished using ANTLR actions from within the OpenMP grammar, and is discussed in the next chapter. Due to the CUDA target's close relationship to C, it is even possible to complete the desired translation in one pass.

## 7.3 Intermediate Data Structures

The most valuable information collected by the compiler frontend, is the type information of the variables. Above all, this data is used to handle the transfer of data to and from the device and host. Any variable referenced within, say a **`parallel`** region, whether a formal list item or not, must have space allocated for it on the device. For this to happen, its size, type, pointer level, and more are required. The **`DeclInfo`** class has seven member variables to hold the type values:

| DeclInfo Member | Description |
|---|---|
| `String name` | The name of the variable |
| `String type_specifier` | The variable type e.g. **`"unsigned char"`** |
| `ArrayList<String> extents` | One entry for each dimension |
| `long array_size` | Can use **`sizeof`** at runtime on unsized arrays |
| `int pointer_level` | 0 is a variable, 1 is a pointer etc. |
| `eTYPE decl_type` | Function pointer, declaration, or definition |
| `DataClause data_clause` | OpenMP sharing attribute e.g. **`shared`** |

The **`DeclInfo`** class is used in the parser by the ANTLR *symbol stack*. A *stack* is required to store the symbol data due to the scoping rules of C. The symbol data of a variable reference is resolved only by traversing the symbol stack *backwards*, in the direction of the global file scope. The first symbol encountered with the same name as the target, contains its type information.

The fields of a temporary **`DeclInfo`** object are updated while parsing, with separate grammar rules identifying different aspects of the variable's declaration; until finally reaching the **`direct_declarator`** rule. At this point a new name-value association is added to the symbol stack.

# Chapter 8

# Back End - Code Generation

The following chapter addresses the specific methods used to translate an OpenMP program's directives, API calls, and associated sequential code into a form capable of compiling for, and exploiting the parallel compute capability of NVIDIA's G80 series cards. Where an OpenMP item has not been implemented due to lack of time, a possible implementation strategy is described. If an OpenMP item has proven intractable, an explanation of this is also provided. The arrangement of sections within this chapter is inspired by that of the OpenMP specification.

## 8.1 Approaching Code Translation

Even by hand, the refactoring required to transform OpenMP C code into CUDA C code, though mildly intricate, is thankfully not structural. The principal sites of alteration are the locations of the OpenMP pragmas, with subsequent modifications essentially local. In general, the sequential code can remain largely untouched. This is largely by virtue of OpenMP's specification of sequential and parallel symbiosis. The construct's region code is then itself amenable to transplant into a new **__global__** function. The transformation of the fundamental **parallel** construct, can assist by leading towards an initial guiding framework. First of all, an OpenMP program with only one **parallel** construct, itself having no function calls, is considered. The following transformation stages are required:

1. Replace the **parallel** construct with a call to a new **__global__** function, **gpump_parallel**
2. Insert a runtime call *before* the new **__global__** call, to transfer and initialise, from host to device, variables *to be used* within the parallel region
3. Insert a runtime call *after* the new **__global__** call, to transfer and initialise variables, from device to host, used within the parallel region

4. Create the **gpump_parallel** function by transplant of the **parallel** region's structured block

Further detail will be added in the sections to follow, though at the high level described, the framework remains true to this throughout. One notable exception concerns functions called from within a **parallel** region. As mentioned in Chapter 6, those functions must be qualified with the **accessible** directive, and through transformation, may possibly become two separate functions; targeting both host and device.

### 8.1.1 Registers versus Shared Memory

Although optimisation was not a priority, it was incumbent upon the project to make *some* use of the fast thread-local device memory available through each multiprocessor's registers or shared memory. Consequently, variables provided to an OpenMP construct either as **private**, **firstprivate**, or **lastprivate** have been implemented using straightforward automatic variable declarations. Space-permitting, these will be allocated in registers.

Alternatively, a shared memory realisation of **private** variables could be considered, and may on occasion be advantageous. For example, a hierarchical **reduction** clause implementation could use **__syncthreads** to first perform subcalculations for each block of a grid.

As the number of threads is resolved only at runtime, **__shared__** variables acting for **private** OpenMP list items would require dynamic memory allocation. However, as no memory allocation functions, callable from the device are available, such memory must be requested as part of the kernel's execution configuration. This is though a precarious manouever, as requesting too much shared memory in the kernel configuration can easily result in a launch failure. Targeting registers instead, through *automatic* declarations has the advantage of a smoother failsafe option: slow, capacious *local* memory is used should register space be inadequate.

### 8.1.2 1D Blocks and Grids

Although each of the first two kernel execution configuration parameters, **Dg** and **Db** are of CUDA *dimension* type **dim3** [16], only the **x** field need be set for use with the current generation of graphics hardware; the **y** and **z** fields are then set to 1.

Such an approach allows for a simpler *1D* addressing scheme, where only the **x** fields of **threadIdx** and **blockIdx** are used for the thread and block identifiers. This is possible due to CUDA's specified limitations. The maximum number of threads in a block is 512, and the maximum size of a grid dimension is 65535. The **x** integer field can reference 512 unique threads. The number of blocks can also be specified by a

[16] **dim3** is an unsigned integer 3-tuple

single integer due to the fact that a third grid dimension is not yet supported by CUDA [17]. Providing scalar integer variable types for either **Dg** or **Db** is supported too; integers are promoted to **dim3**s.

Such simplification was particularly useful when considering OpenMP's approach to specifying thread numbers. Rather than using six integers to specify the number of CUDA threads, the proposed scheme uses only two. Nevertheless, a prime number of threads remains highly impractical. The OpenMP interface, without nested parallelism, permits only a scalar integer value to request a thread count; for subsequent parallel constructs. Consequently, although specifying a prime number of threads less than 512, say using the **OMP_NUM_THREADS** environment variable, is *possible*, any larger prime number is not.

To overcome this issue, the number of threads requested must be either a multiple of the number of processors, or less than that number of processors. For example, specifying **num_threads(251)** on a 128-processor 8800 GTX card, results in the creation of 256 threads; 16 blocks, each with 16 threads. Requesting 255 threads produces the same result, despite that number being composite. This is primarily a practical solution, aiming for a balanced and transparent work distribution; while avoiding primality tests of up-to 41-bit values.

## 8.2 Directives

The following section describes the stages required to implement the OpenMP directives, which begin with the string **#pragma omp**, into the NVIDIA CUDA language.

### 8.2.1 Parallel Construct

The combined grammar's **parallel_construct** rule incorporates an ultimately well-used pattern of embedded action code placement, with entries placed both before and after a rule's other operations. The code within the **@after** action in this rule, shown below, executes subsequent to all other rule operations. Although the equivalent built-in ANTLR action for rule initialisation, **@init**, is also available, it was found to produce unpredictable results due to multiple invocations in the presence of ANTLR's backtracking; though a similar effect can safely be achieved by placing the actions directly prior to a rule's first alternative.

[17] $65536^2$ is equal to $2^{32}$

```
parallel_construct
@after  {
  g_bInParallel = false;
  g_pt.create_parallel(g_VarNamesAccessed, $Symbols, $sb.text);
  tokens.replace( $parallel_construct.start,
                  $parallel_construct.stop, g_pt.getPar());
}
  :
  {
    g_bInParallel = true;
    g_iStackIndexAtParallel = $Symbols.size();
    g_VarNamesAccessed.clear();
    g_ParFor.parallel_for = false;
  }
  parallel_directive sb=structured_block
  ;
```

The lower of the two action code blocks shown above is executed first, and ensures that variables accessible by rules matched elsewhere are properly initialised. The upper, **after** block, with parsing of the construct now completed, is free to translate the token stream using the **replace** method. The token stream from the **structured_block** rule will be used to form most of the body of a new kernel function. Consider the following conceptual example,

```
int minimal_par()
{
  int        p;
  #pragma omp parallel private(p)
  {
    int      x;
    /* comments remain */
    p = 0;
    x = p + 10;
  }
}
```

The structure block following the **parallel** directive is transplanted verbatim into the code below[18]; even the braces remain. As **p** was identified as a **private** sharing attribute, it must be redeclared. By using automatic storage, a separate copy of **p** will exist for each thread; and should hopefully be allocated space in a register. Note that variable **p**'s new declaration syntax no longer has extra spaces between the type and the

[18]The function body *outside* of the structured block, has though been slightly tidied; to aid readability

name. The original declaration of **p** in the sequential part remains untouched; while the declaration syntax of **p** in the kernel has been newly generated:

```
__global__ void gpump_parallel0(gpump_params0 *gpump_li0)
{
  int p;

  {
    int      x;
    /* comments remain */
    p = 0;
    x = p + 10;
  }
}
```

In this example, the text of the structured block was not modified between the execution of the two ANTLR code action blocks. This will often not be true. Were **p** instead identified as having the **shared** attribute instead, modifications would be required; as discussed later.

Using **TokenRewriteStream.replace**, the text of the original OpenMP **parallel** construct is exchanged for the call and execution configuration of the device kernel; i.e. **gpump_parallel0** shown earlier. Additional calls to host runtime support routines responsible for the transfer of data to and from the device are also inserted, and can be inspected in Appendix D. For brevity, only the kernel call and subsequent CUDA call is shown here:

```
gpump_parallel0<<<gpump_utils_get_num_blocks(),
                  gpump_utils_get_threads_per_block()>>>(
                  gpump_li0);
cudaThreadSynchronize();
```

The call to **cudaThreadSynchronize** ensures the translation conforms to the semantics of OpenMP. In OpenMP, the execution of a **parallel** region is a synchronous operation. However, the call to a device kernel in CUDA is *asynchronous*; and so returns prior to completion. By inserting a call to **cudaThreadSynchronize**, the program will not proceed from this point, until the preceding kernel call has completed.

### 8.2.2 Work-sharing Constructs

Much of the **parallel** construct's translation process can be applied again to the first of the work-sharing constructs, the loop construct. The text of the matched **for_-**

```
void minimal_parfor()
{
  int i, j;
  #pragma omp parallel
  {
    #pragma omp for private(j) schedule(static)
    for (i = 0; i < 10; i++)
    {
      j = i;
    }
  }
}
```

Figure 8.1: Minimal Loop Concept

**construct** now forms part of the structured block later processed by the **parallel_construct**, and discussed previously. The bounds of the loop, and its initial expression statements though must first undergo some modification. Were this text not altered, each of the CUDA threads might redundantly execute the entire workload of the original for loop; rather than dividing the labour as expected of OpenMP. For example, to parallelise a loop involving 80 iterations using 8 threads should, with a **static** workload distribution, provide each thread with 10 iterations. The following two code excerpts illustrate the transformation from OpenMP loop construct, to CUDA kernel,

The absence of the OpenMP **nowait** clause compels the compiler to place a barrier, **gpump_barrier_d**, after the *for-loop*; so halting each individual thread until all complete their own portion of the workload. The output below also demonstrates that the iteration variable, **i** has been correctly implemented according to the **private** sharing attribute. Note that in this example, **i** and **j** are operating as temporary variables. By here having **private** sharing attribute, OpenMP does not expect useful values from them after the loop construct.

The above form of the three, colon-separated, loop initialisation expressions is freshly created by the compiler, according to the parameters of the original loop. As described in the OpenMP specification, the **for** directive places some restrictions on the structure of the corresponding C language loop. For example, a loop's increment expression may only incorporate the **+** or **-** operators. It is this canonical OpenMP em for-loop which allows for the consistent use of the *not equal to* loop structure; shown above, and capable of representing all permissable *for-loops*.

Consequently, should the prior source code incorporate a *less than or equal to* operator instead of a less than, the variable initially holding the *bound* of the loop, **gpump_b**, would simply be set to 11[19] instead. Another loop using **i-=3;** as its increment expression would result in an initial assignment of **-3** for the *increment* variable, **gpump_-**

[19] In fact this becomes **10 + 1** due to the possibility that the **10** may be a variable expression.

```
__global__ void gpump_parallel0(gpump_params0 *gpump_li0)
{
  int i;
  int j;
  int gpump_lb, gpump_b, gpump_incr;
  {
    gpump_lb    = 0;
    gpump_b      = 10;
    gpump_incr  = 1;
    gpump_utils_calc_static_for_bounds(&gpump_lb, &gpump_b,
                                       gpump_incr);
    for (i = gpump_lb; i != gpump_b; i += gpump_incr)
    {
      j = i;
    }
    gpump_barrier_d(); // implicit loop barrier
  }
}
```

Figure 8.2: Translated Minimal Loop Concept

**incr**. The runtime function used to calculate each thread's *for-loop* parameters is as follows,

```
__device__
void gpump_utils_calc_static_for_bounds(int *plb, int *pb,
                                        int incr)
{
  int tid         = gpump_get_thread_num_d();
  int nthreads    = gpump_get_num_threads_d();

  int niters      = ceil((float)abs(*pb - *plb) / abs(incr));
  int num_longer_blocks = niters % nthreads;
  int chunk_size  = niters / nthreads +
                      (tid < num_longer_blocks);

  *plb += tid * (incr * chunk_size);
  *plb += (tid >= num_longer_blocks) *
            (num_longer_blocks * incr);
  *pb = *plb + (incr * chunk_size);
}
```

After obtaining the total number of threads, the current thread *id*, and the total number

of *for-loop* iterations, the *chunk* [20] size is then determined. Only if the number of iterations, divided by the number of threads leaves no remainder, will the chunk size be equal for *all* threads. In every other situation, the remainder can be redistributed, by adding *one* to the result of the division to a number of threads equal to the remainder value; so giving each thread its own own chunk size. The threads with lowest ids are first to be so lengthened.

Each thread then defines its own lower loop bound, scaled according to its id. As the *increment* value does not change, the loop's bound assignment, which uses a multiple of **`incr`** as an offset from the lower bound, guarantees the effective execution of the *not equal to* form used by the following loop. Incidentally, if there are more threads than loop iterations, a quantity of threads will safely be assigned a chunk of zero size, and so will not execute their *for-loop*'s work statement.

The implementation of a *non-default* OpenMP chunk size builds upon this technique. Each individual *chunk* is thus handled by again inserting code to call **`gpump_utils_calc_static_for_bounds`**, followed by the altered *for-loop*. Now though this is itself placed within a new loop which at each iteration provides a thread with a new chunk, until no further work remains.

The framework for the *non-default* chunk size is currently also used for the *default* situation; in this case, the outer loop performs only one iteration. A full example of the generated code for the *for-loop* is demonstrated in Appendix D.

Of the *for-loop* **`schedule`**s, only **`static`** has been realised. The **`static`** schedule was prioritised due to the lack of communication and synchronisation required between threads [21]. Nevertheless, although performance of a **`dynamic`** or **`guided`** schedule may be compromised by the tally of remaining chunks in main memory, their implementation should be possible in a lock-free fashion using CUDA atomic routines such as **`atomicSub`**.

Currently, the chunk size (and schedule) of a *for-loop* can only be set through the **`schedule`** clause.

### Other Work-sharing Constructs

The two remaining work-sharing constructs, the **`sections`** and **`single`** constructs, have not been implemented. Both should be possible, and their implementation is now briefly considered.

The basic framework of the **`sections`** construct can be implemented using a C **`switch`** statement. The initial **`switch`** "parameter" accepts the unique thread id, thereby selecting from the **`case`** alternatives. Each **`case`** statement would then be created from the transplanted structured blocks following the **`section`** directives from the original source code.

[20] The OpenMP concept of a chunk is adopted; a contiguous non-empty loop iteration subset.

[21] As will later be discussed, synchronisation between multiprocessors is problematic.

A **single** construct implementation can begin as the **master** construct; by executing the structured block only *if* the thread's id is zero. Alternatively, each thread encountering the construct could atomically increment an integer counter in main memory; zeroed on kernel initialisation. The first thread discovering the integer's *original* value to be zero, executes the structured block. The thread finding the counter having one less than the total number of threads, can reset the value to zero for future use.

### 8.2.3 Combined Parallel Work-sharing Constructs

There are two combined parallel work-sharing constructs in C: the parallel loop; and the **parallel sections** constructs. Both are functionally equivalent to a **parallel** construct enclosing only its relevant work-sharing construct.

The **parallel for** construct is similar to the **parallel** construct. Looking at the grammar rule for each of them, both are revealed simply to match two child rules; the first being the relevant directive, while the second is a statement. The **parallel for** construct's statement though is an iteration statement. This distinction is however only syntactical; compared to a **parallel for** construct, a **parallel** directive, followed only by an iteration statement, has a different *semantic* meaning within OpenMP.

The **parallel for** construct is implemented using a variation on the method used for the **parallel** construct. The main difference is that the **CUDAPrinter** method called by the rule's **@after** action block, is given a **ParFor** object as its formal parameter; instead of the text from a matched **statement** rule. This **ParFor** object is utilised as it was by the loop construct: to encapsulate the parameters of the *for-loop* as they are identified by the parser. It is worth noting that the **ParFor** object does *also* contain the **statement** rule text; i.e. the loop body. An example translation of a **parallel for** construct, applied to a modified version of the first example from the OpenMP specification is included in Appendices 3 and 4.

The **parallel sections** construct has not been realised. Like the **parallel for** construct, the previous outline for a **sections** construct implementation should combine with the **parallel** construct with minimal further alterations.

### 8.2.4 Master and Synchronisation Constructs

Preparatory work for the project had anticipated that implementing OpenMP operations relating to synchronisation would be challenging. The release by NVIDIA of new firmware, accompanied by *atomic* routines during the term of the project was greatly encouraging, and has enabled hitherto intractable translations. However, to realise the **critical**, **atomic**, and **ordered** constructs, some form of locking mechanism is required. As will be discussed later, even with the new routines, a lock applicable to all threads of a grid was found to be beyond reach. So it is that the **critical**, **atomic**, and **ordered** constructs have also not been implemented.

Nevertheless, the eleven CUDA atomic routines available *could* accommodate the five legal OpenMP **`atomic`** expression statements, but only when operating upon integers. All eleven CUDA atomic routines manipulate integer pointer targets only.

subsubsection**`master`** Construct The **`master`** construct was successfully implemented by adding a few ANTLR actions to the **`master_construct`** rule; as shown below,

```
master_construct
  :
  { g_bInMaster = true; }
  md=master_directive sb=structured_block
  {
    g_bInMaster = false;
    String fn = g_bInParallel ?
        "gpump_is_master_d()" : "gpump_is_master()";
    tokens.replace(  $md.start, $sb.stop,
                      "if ("+fn+")\n\t" + $sb.text);
  }
;
```

The **`gpump_is_master*`**[22] calls are implemented as separate host and device runtime library routines. While the device routine returns non-zero, *only* to the thread with zero id, the host routine will always return **`1`**. The **`boolean`** variable, **`g_bInMaster`**, may be used to detect errors conditions; for example, an OpenMP barrier cannot be placed within a **`master`** construct.

### `barrier` Construct

An OpenMP **`barrier`** operation is achieved by replacing the construct with a call to the CUDA runtime routine, **`gpump_barrier*`**. Being a *standalone*[1] directive, the action code within the ANTLR grammar rule, which enables the source translation, is completely straightforward. It involves no other parsing rules, and the text of the function call simply replaces the **`barrier`** construct directly. The runtime library routine, though, shown below for the device, is interesting.

[22] When used to follow a function name beginning with textttgpump_, the asterisk signifies that both a device and host version of the routine exists; with device function names ending in **`_d`**.

```
__device__ void gpump_barrier_d()
{
  if (threadIdx.x == 0) {
    atomicAdd(&gpump_atomic_counter, 1);  // first set...
    while (
      atomicCAS(&gpump_atomic_counter, gridDim.x, gridDim.x) !=
        gridDim.x);
    atomicSub(&gpump_atomic_counter, 1);  // ...then unset.
    while (atomicCAS(&gpump_atomic_counter, 0, 0) != 0);
  }
  __syncthreads();
}
```

There follows a description of the process implemented by the above code. A counter, pre-initialised to **`0`**, is incremented by "thread zero" of each block; all other threads call **`__syncthreads`**. When the counter becomes equal to the number of active CUDA blocks, **`gridDim.x`**, all "thread zeros", monitoring the counter from a while loop, now also call **`__syncthreads`**, so releasing every thread; and so too the barrier.

The counter is a global variable declared and initialised using the **`__device__`** qualifier as shown:

```
__device__ int gpump_atomic_counter = 0;
```

To allow the barrier code to run twice, the **`gpump_atomic_counter`** counter must be returned to zero. A suitable point for the reset operation is immediately after the counter has reached its maximum. The process described above must therefore be expanded to include each "thread zero" decrementing the counter back *to zero, before* the common call to **`__syncthreads`**.

A contrasting approach might switch the "sense" of the counter from *rising*, to *falling* on alternate barrier calls. This would involve atomic read and write access to a new "sense" variable, and may not *necessarily* improve performance; performance was also not a priority. The essential factor in this global CUDA barrier's successful operation is that *every* thread makes the same call to **`__syncthreads`**.

### `flush` Construct

CUDA provides no **`flush`** operation, and a method of implementing its operation was not found. Specific details regarding the CUDA memory model have in fact only been documented in recent versions of the programming guide[2], and unfortunately solely

concern the __**shared**__ memory of individual blocks. Updates to __shared__ memory have sequential consistency for an individual thread, and relaxed consistency between threads.

Consequently, to ensure a consistent view across threads, even of __**shared**__ memory variables, *all* block member threads must call __**syncthreads**[23]. However, block level access to threads is not exposed by the project's OpenMP implementation, and so this is of little practical value for a **flush** construct. No information about the memory consistency of main memory on the device is currently available.

### 8.2.5 Data Environment

The sharing attributes of variables referenced within a **parallel** construct may either be *predetermined*, *explicitly detemined*, or *implicitly determined*[1]. Using the language of the OpenMP specification, the following rules for *predetermined* variable sharing attributes have been implemented:

- Variables with automatic storage duration which are declared in a scope inside the construct are private.
- Variables with heap allocated storage are shared.
- The loop iteration variable in the *for-loop* of a **for** or **parallel for** construct is private in that construct.

Variables declared in a region, but not in a construct, are private.

The **threadprivate** directive has not been implemented; and consequently the **copyin** and **copyprivate** clauses too are absent. The static storage class has also not been implemented.

The following sections describe the OpenMP data-attribute clauses, and their implementation using CUDA.

#### **default** clause

As part of the **data_clause** grammar rule, the two alternatives relating to the default clause simply update an internal **boolean** variable for later consultation. Here is the relevant excerpt from the grammar:

```
| 'default' '(' 'shared' ')'
  { g_pt.m_ScopeHandler.SetDefault(true); }
| 'default' '(' 'none' ')'
  { g_pt.m_ScopeHandler.SetDefault(false); }
```

[23] __**syncthreads** must always be called in this way.

### `shared` clause

At the parsing stage, the **`shared`** clause is also handled as part of the **`data_clause`** rule; the relevant excerpt is shown below.

```
| 'shared' '(' identifier_list ')'
  { g_pt.m_ScopeHandler.Store(g_IdList, DataClause.SHARED); }
```

While parsing the **`shared`** clause, another common rule **`identifier_list`**, shown below, is referenced. Note that the **`data_clause`** rule is resposible for the state of **`g_bCollectList`**.

```
identifier_list
  : { if (g_bCollectList) g_IdList.clear(); }
    a=IDENTIFIER      { g_IdList.add($a.text); }
    (',' b=IDENTIFIER { g_IdList.add($b.text); } )*
  ;
```

An OpenMP program variable with **`shared`** attribute requires not only for space to be allocated on the device, it will also require the value it has upon entry to a **`parallel`** construct to be copied across, and upon exit, to be copied back again; for use in the subsequent sequential part.

Memory allocation for the **`shared`** variable *twins* on the device is obtained immediately prior to the kernel call, using **`cudaMalloc`**. For each **`parallel`** region, a new structure is created; for the first **`parallel`** region, the structure is named **`gpump_params0`**, for the second **`gpump_params1`**, and so on. Each shared variable has its name, type, and extents used to describe a member of the new structure. The following example code assumes that 256 CUDA threads are launched,

```
void minimal_shared_par()
  int p[256];
  #pragma omp parallel shared(p)
  {
    int tid = omp_get_thread_num();
    p[tid] = tid;
  }
}
```

The newly defined **`gpump_params0`** structure, created to accommodate the shared array, **`p`**, is declared as,

```
typedef struct {
  int      p[256];
  void *__p;
}  gpump_params0;
```

**cudaMalloc** is then called to allocate sufficient space for the structure in device memory. More advanced codes will naturally create more fields in the generated structure, while still using one **cudaMalloc** call. Consequently, each variable is also given a *second* structure member, to store its *device* address. In the following kernel example code, generated from the **minimal_shared_par** function above, the address of **p** is obtained on the last line. This would be useful when, for example, a **shared** pointer targets **p** from within a kernel.

```
__global__ void gpump_parallel0(gpump_params0 *gpump_li0)
{
  {
    int tid = gpump_get_thread_num_d();
    (gpump_li0->p)[tid] = tid;
  }

  gpump_li0->__p = &gpump_li0->p;
}
```

The above kernel's formal parameter is thus the device address of this structure of variable names, and all references to **shared** variables, such as **p**, within this construct, are preceded by the string **gpump_li0->**, and bracketed. Such translation of **shared** variable references is accomplished within the parse tree, providing say, the **parallel_construct** rule's ANTLR actions, with the necessary structured block text; now in altered form.

### private clause

Parsing of the **private** clause is handled by the **data_clause** rule in much the same way as with the **shared** clause. During source translation however, things are quite different. Variables with the **private** sharing attribute, are neither required to initialise their twin variable on the device, nor update the original list item upon the kernel's termination.

**private** variables are declared using the automatic storage class, at the start of each kernel. Each variable's original name and type are used, and references to its use, while **private**, within the relevant construct remain unaltered.

Should a clause of an enclosed construct alter a variable's data-sharing attribute from **shared** to **private**, the two reference styles may co-exist. Variables referenced there as **shared** will be preceded with the **gpump_li0->** pointer decoration, while the enclosed construct's **private** references remain unaltered. Consider the earlier Figure 8.1. Were its loop construct now immediately followed by a reference to variable **i**, such as, **i = 0;**, the translated code would differ from the previous output, of Figure 8.2, only by the addition of **(gpump_li0->i) = 0;**, at the equivalent location.

As described previously, handling **private** variables in this fashion is not only elementary, but also allows the variable to either be placed optimally within a register, or instead safely within local memory, should space be limited; something a **__shared__** memory realisation could not provide.

### firstprivate clause

The project's implementation of the **firstprivate** clause performs a private variable's distinctive initialisation, by assignment from the **gpump_params** value holding the original list item's value. This value was stored by the host immediately prior to the start of the kernel; in a manner identical to that used by **shared** data-sharing clause. In the following example, the **firstprivate** clause is used on both a **parallel** construct, and an enclosed loop construct.

```
void first_private()
{
  int i, num = 0, first = 128;
  #pragma omp parallel shared(i,num) firstprivate(first)
  {
    first -= omp_get_thread_num();
    #pragma omp for firstprivate(num)
    for (i = 20; i >= 0; i = i - 2)
    {
      num += i;
    }
  }
}
```

It may be noticed in the output below[24] how the variable made **firstprivate** on the *loop* construct, **num**, is still declared at the start of the kernel. Were further work-sharing constructs to follow, again using the **firstprivate** clause to declare **num**, the same automatic storage declaration would be used again.

[24]This translated code example has again been tidied slightly to aid readability. In this case, the code handling different chunk sizes has been removed from the loop construct.

```
__global__ void gpump_parallel0(gpump_params0 *gpump_li0)
{
  int     first = (gpump_li0->first); // parallel:firstprivate
  int     i;
  int     num;
  int gpump_lb, gpump_b, gpump_incr;

  {
    first -= gpump_get_thread_num_d();

    num = gpump_li0->num;   // for:firstprivate
    gpump_lb =              20;
    gpump_b =                       0 - 1;
    gpump_incr =    -(2);
    gpump_utils_calc_static_for_bounds(&gpump_lb, &gpump_b,
                                       gpump_incr);
    for (i = gpump_lb; i != gpump_b; i += gpump_incr)
    {
    num += i;
   }
     gpump_barrier_d(); // implicit loop barrier
   }

  gpump_li0->__first = &gpump_li0->first;
  gpump_li0->__i = &gpump_li0->i;
  gpump_li0->__num = &gpump_li0->num;
}
```

Support for **`firstprivate`** array variables is only operational for data-sharing clauses attached to a **`parallel`** directive.

### `lastprivate` clause

The **`lastprivate`** data-sharing clause has not been implemented. No problems are foreseen however, with a translation process which can, like the **`firstprivate`** clause, borrow heavily from the methods of the **`private`** and **`shared`** clause implementations.

### `reduction` clause

Implementing the **`reduction`** clause can again borrow from earlier translation techniques. Those used upon clauses providing copies of list items, made private to each thread, can be used again. Only the addition operator, applied to an integer type, has

been realised. This is due to a relatively straightforward implementation which again employs CUDA's atomic routines.

Of all the data-sharing clauses parsed within the **`data_clause`** rule, the **`reduction`** clause is the only one to reference another rule; **`reduction_operator`**. The data-sharing clause and operator are then represented as a single combined enumeration; in this case, **`DataClause.REDUCTION_ADD`**.

The following example should complete with x holding a value of 4950,

```
void reduce()
{
  int x, i;
  #pragma omp parallel shared(x)
  {
    #pragma omp for reduction(+:x)
    for (i = 0; i < 100; i++)
    {
      x += i;
    }
  }
}
```

The translated output code below shows both the **`shared`**, and later, **`private`** representations of the **`x`** variable working together. Having both been set to the relevant initialisation value for the operator, the loop construct is followed immediately by a barrier; though this is probably unneeded[25]. Following this, the CUDA **`atomicAdd`** function is used to safely add the private copy of **`x`** to the shared location accessible through the variable structure, **`gpump_li0`**, for subsequent retrieval by the host, upon the kernel's return.

[25] Sequential consistency, or a memory model for device memory in CUDA is undocumented

```
__global__ void gpump_parallel0(gpump_params0 *gpump_li0)
{
  int     i;
  int     x;
  int gpump_lb, gpump_b, gpump_incr;

  {
    x = 0; // reduction:init:+
    (gpump_li0->x) = 0; // reduction:init:+
    gpump_lb =              0;
    gpump_b =                       100;
    gpump_incr =    1;
    gpump_utils_calc_static_for_bounds(&gpump_lb, &gpump_b,
                                       gpump_incr);
    for (i = gpump_lb; i != gpump_b; i += gpump_incr)
    {
    x += i;
  }
    gpump_barrier_d();  // flush to init reductions
    atomicAdd(&(gpump_li0->x), x); // reduction:+
    gpump_barrier_d(); // implicit loop barrier
  }

  gpump_li0->__i = &gpump_li0->i;
  gpump_li0->__x = &gpump_li0->x;
}
```

## 8.3 Runtime Library Routines

The following three sections describe the CUDA implementation of the OpenMP runtime library routines. Starting with the execution environment routines, the lock routines are considered next, with the timing routines last. In most cases, two versions, one for the host, and one for the device have been implemented. The host, or sequential implementation is often rudimentary. The device routines have their names appended with **_d**, while both exchange the initial **omp_** string with **gpump_**.

### 8.3.1 Execution Environment Routines

The project compiler does not support nested parallelism, and also lacks the related **omp_get_nested** and **omp_set_nested** routines. **omp_get_dynamic** and omp_set_dynamic are also absent, though the routine indirect mappings between threads and

processor numbers with CUDA would ensure a straightforward implementation. The execution environment routines which have been implemented follow.

### `omp_get_num_threads`

On the device, **`gpump_get_num_threads_d`** returns the product of the number of threads per block, **`blockDim.x`** with the number of blocks, **`gridDim.x`**. The host function, **`gpump_get_num_threads`**, returns 1.

### `omp_get_max_threads`

Only the host version has been implemented. **`gpump_get_max_threads`**, consults the environment variable, **`$OMP_NUM_THREADS`**, and then ensures that, if required, this value is rounded up to the next multiple of **`nprocs`**; the number of processors on the device. The returned value will also be capped, should the environment variable be larger than the maximum CUDA thread limit of $2^{41}$.

### `omp_get_thread_num`

The device version, **`gpump_get_thread_num_d`**, calculates the product of the number of threads in a block, with the zero-based block index. To this is added the thread's index within its block, before the result is returned. The host version returns 0.

### `omp_get_num_procs`

There is no CUDA routine to obtain the number of processors, or multiprocessors. Consequently, the "magic" number of 32, true for the project hardware, has for now been inserted. An environent variable would be a preferable alternative. Only the host version has been implemented.

### `omp_in_parallel`

No function or variable alone appears capable of reporting from CUDA on this apparently mundane question. However, due to the active role played by the project compiler in translating and relocating **`omp_in_parallel`**, the device implementation simply returns **`1`**, while the host version returns **`0`**.

### 8.3.2 Lock Routines

A lock implemented on a shared memory architecture will usually involve atomic instructions. Although there are examples of locking algorithms which do not require atomic instructions, such as Leslie Lamport's "Bakery" algorithm [7], these require a memory consistency model which is not available for the main memory of NVIDIA's CUDA hardware.

Included in the June release of CUDA version 0.9 were a set of eleven atomic routines. These routines operate upon integer data types, and so the possibility of writing a global "spinlock", based on a unique thread id semaphore seemed possible. In practise, the atomic functions are still at the behest of the undefined memory memory, which is itself affected by the thread scheduler. Implementations of **`omp_set_lock`** using this method then failed, by not returning; the **`while`** loop never finding the shared integer in the necessary state. This would often cause the system to hang.

It should be mentioned that **`omp_test_lock`** *was* found to be possible. The requirement for a loop within **`omp_set_lock`** seems to be the crucial distinction. The code for this is below, where **`gpump_lock_t`** an integer **`typedef`**.

```
__device__ int gpump_test_lock_d(gpump_lock_t *lock)
{
  int tid = gpump_get_thread_num_d();
  if (*lock == GPUMP_INIT || *lock == tid)
    return 0;

  if (atomicCAS(lock, GPUMP_UNLOCKED, tid) == GPUMP_UNLOCKED)
    return 1;  // Success

  return 0;
}
```

### 8.3.3 Timing Routines

**`omp_get_wtime`** was implemented using the CUDA utility library function, **`cutGetTimerValue`**. This function returns a value in milliseconds, and so is first multiplied by 0.001 to obtain a time in seconds. **`omp_get_wtick`** has not yet been realised.

## 8.4 Host Runtime Library

A host runtime library was created to allow the generated code to both be less verbose, and to utilise greater modularity. This library, **gpump_malloc**, must be linked with the CUDA C compiler output; which will itself have a **#include** for the relevant header file, **gpump_all.h**, by then placed within it. The compiler is thereby provided with a small collection of routines for the runtime transfer of data between host and device. Those routines from this library referenced directly by the compiler are briefly discussed below.

### 8.4.1 gpump_copy_to_device

The first parameter of **gpump_copy_to_device** is a pointer to an array of **gpump_copy_config_t** elements. **gpump_copy_config_t** is a structure used to store all data obtained by the parser about each variable in the source code, prior to translation. The second parameter is an integer specifying the number of such array elements.

**gpump_copy_to_device** allocates space for, and copies to the device, any variable specified with the necessary OpenMP sharing attributes. This function also locates and transfers any **accessible** data referenced by specified pointers; and updates the transferred device pointers for the new address space.

**gpump_copy_from_device** has a similar function to **gpump_copy_to_device**, though with data transfer now in the opposite direction. **gpump_copy_from_device** usually precedes the kernel call, while **gpump_copy_to_device** follows it.

### 8.4.2 gpump_alloc_list_add

This function is used to identify a variable that is referenced by a pointer later included as an OpenMP list item. The first parameter is a **void** pointer accepting the address of the referenced variable, while the second parameter is an integer identifying its length in bytes. This pair of data items are added to a list which may later be consulted by **gpump_copy_from_device** or **gpump_copy_from_device**.

### 8.4.3 gpump_malloc

All four of the common C **malloc** family functions are wrapped by an equivalent **gpump_** routine. The utility of this is discussed in Chapter 6. Internally **gpump_malloc** implementation relies on **gpump_alloc_list_add**.

## 8.5 Implementing the Accessible Extensions

Code marked with either of the two new **`accessible`** declarative directives must be converted into a form compatible with the CUDA runtime. The compiler translation of each directive is now described.

### 8.5.1 `accessible` data

The **`accessible`** directive is followed by a comma-separated list of variable names. For each variable name, the compiler inserts a call to **`gpump_alloc_list_add`**; each placed in sequence at the site of the directive. The first parameter is the variable name, preceded by the **`&`** symbol. The second parameter is the **`sizeof`** operator followed by the same variable name. The use of this **`accessible`** directive to specify global data is not yet realised.

### 8.5.2 `accessible` functions

Functions preceded with the **`accessible`** directive have their definitions replaced by the compiler, in favour of two similar functions. The first targets the device, while the second, remaining essentially unaltered, targets the host. The device function declaration is preceded with the CUDA function qualifier, **`__device__`**, and given a new name; based on the original, though appended with **`_d`**. All references to this renamed function, called from within what becomes the **`__global__`** function body, are renamed in the same way. Furthermore, all functions called from inside the new device function, are also appended with **`_d`**.

# Chapter 9

# Performance Analysis

Beyond targeting the placement of **`private`** variables within registers, the translation effort has not, *per se*, been concerned with execution times and runtime optimisation. Nevertheless, as the target user group for the compiler is the HPC community, it is sensible to give succour to our curiosity and examine the performance characteristics of a translated OpenMP example code.

The loop construct example code[26] taken and modified from the OpenMP specification is chosen as the benchmark code. This code has been discussed already and has the qualities of simplicity and scalability while also demonstrating the use of the new declarative directive, **`accessible`**. The project compiler is currently not robust enough to tackle arbitrary or classic OpenMP benchmarks, and development of a new suite, suitable for the compiler as it stands, would at this stage be hard to justify.

The host and device specifications of the benchmark system are given in Appendix E.

## 9.1 The Watchdog Timer

Windows and Linux based operating systems have a safety mechanism known as a "watchdog" timer[16]. If a program causes a system's primary graphics adapter not to respond for between five and ten seconds, the program is aborted. Unfortunately this limit applies to CUDA programs. Under Windows, it is also possible that such conditions will cause the machine to freeze; requiring a "hard" reset. More frequently, the monitor's display will become afflicted with pixel-scale visual artifacts; for the duration of the session.

A solution recommended by NVIDIA is to ensure the card running the CUDA program is *not* used as the primary graphics adapter. However, despite an integrated graphics adapter on the development system's motherboard, the BIOS would not permit any configuration for a display adapter in the x16 PCI Express slot, other than *primary*.

[26] Appendices C and D list this code prior to instrumentation with **`omp_get_time`**

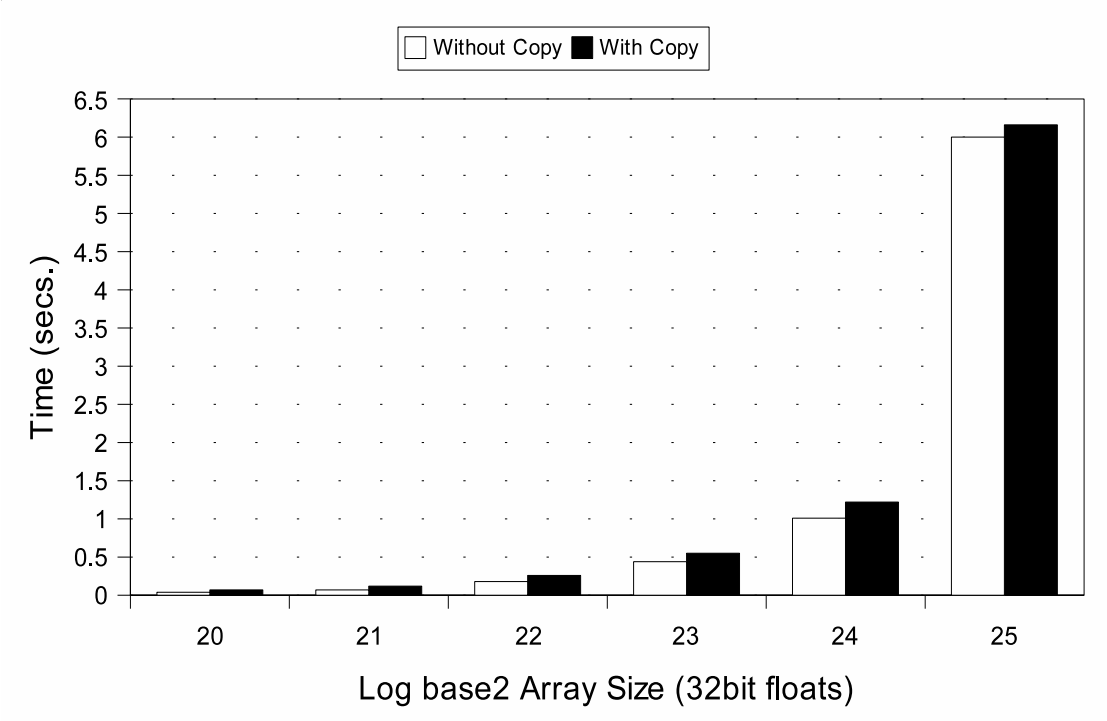


Figure 9.1: Array Size on Performance with 256 Threads on 4 Blocks

This will not be an issue for NVIDIA's HPC Tesla[8] range, which have no external display connectors.

The presence of the watchdog timer therefore requires that benchmark runs are designed to complete within approximately five seconds or less.

## 9.2 Host to Device Transfer

The published bandwidth of the PCI Express x16 bus is 4Gbps[17]. This *peak* rate will bound the transfer rate of list items to and from the main memory of the host and device. To examine how this will in practise affect the performance of the more granular transfers in our test code, the wall clock times, taken before and after the translated loop construct are noted, while the size of the two arrays, **a** and **b**, are increased. Each test is then repeated, with the placement of the timing routines altered to measure only the kernel execution; and not the data transfer. NVIDIA guidlelines on block and grid sizes are met by choosing one block for each multiprocessor, and 64 threads per block.

Figure 9.1 charts these results. Until the final arraysize of $\mathbf{2}^{25}$, a smoothly increasing, direct relationship between runtime and data size is observed. Also encouraging is the fact that as the data size increases, the proportion of time taken to copy data falls, as highlighted in the following table,

| Array Size | Copy Runtime Percent. |
|---|---|
| $2^{20}$ | 42.8 |
| $2^{21}$ | 41.7 |
| $2^{22}$ | 30.8 |
| $2^{23}$ | 20.0 |
| $2^{24}$ | 9.0 |
| $2^{25}$ | 2.6 |

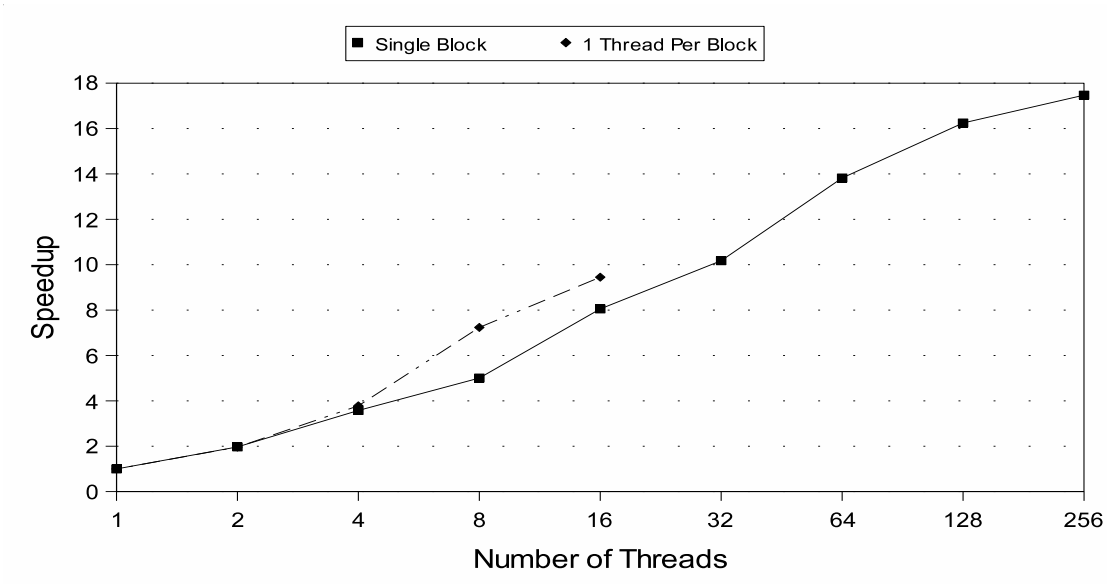


Figure 9.2: Contrast of Block and Thread Scaling

## 9.3 Speedup

To observe how thread numbers affect runtime and therefore speedup, a suitable array size must be chosen. Although a consistent incremental drop in proportional copy time is observed through the previous benchmark, the largest data size there, $\mathbf{2}^{25}$, should not be chosen as its six second runtime is already on the watchdog timer boundary.

An array size of $\mathbf{2}^{25}$ provides a good range of thread values capable of performing within the alloted time. With the CUDA system, there are two simple and distinct ways to increase thread counts: by increasing the number of threads in a block, or by increasing the number of grids. Both are represented in Figure 9.2. Starting with one thread, the first benchmarks examines speedup while increasing only the number of threads in a CUDA block, while the second benchmark increases only the number of blocks.

Of the two, the benchmark which only increases the threads in a block displays poor scaling as early as eight threads. With eight processors per multiprocessor, this may correspond to the point where thread scheduling begins. The second benchmark's speedup

scales beyond the equivalent thread count of four. At this stage the active number of threads exceeds the number of available multiprocessors, but strong sclaing continues to eight. After sixteen threads, the program became unreliable, showing wildly varied runtimes of over ten seconds.

## 9.4 Comparison to Traditional OpenMP

The CUDA system is though a data parallel system, and effectively crippled by the previous benchmark's thread restrictions. A valuable evaluation of its relative performance will first determine the ideal thread count and configuration for the benchmark, and then determine the largest data size which will run without watchdog interference. The runtime of this CUDA benchmark could then be compared to that of one compiled with a traditional OpenMP compiler, executing on the host machine.

| **Num. of Blocks** | 8 Threads | 16 Threads | 32 Threads | 64 Threads |
|---|---|---|---|---|
| **4** | 1.23 secs. | 1.05 secs. | 1.24 secs. | 1.22 secs. |
| **8** | 0.97 secs. | 1.02 secs. | 1.21 secs. | 1.22 secs. |
| **16** | 0.99 secs. | 1.00 secs. | 1.20 secs. | 1.14 secs. |

The matrix above lists the runtime of sixteen thread configurations, all operating upon the previous benchmark code, using $\mathbf{2}^{24}$ array size. These results indicate that an arrangement of 64 threads, arranged in 8 blocks should be optimal for this benchmark.

The same OpenMP benchmark code was then compiled[27] and executed with two threads on the dual-core host machine, giving a runtime of 0.12 seconds; a factor of eight times faster.

## 9.5 Analysis

The first explanation of the slow performance of the loop construct example on the device, compared to the host, is that the code's performance is bound heavily to main memory accesses. In contrast, CUDA is documented to work most effectively, and primarily, on codes displaying arithmetic intensity.

Secondly, the random access of global memory is in no way coalesced. The CUDA programming guide recommends that careful, communal access to contiguous regions of memory, by sequential thread ranges be used, to gain performance advantage.

The clock rate of the individual serial processors on each multiprocessor are also lower; though only 1450MHz to 2200MHz.

No sequential optimisation has been performed by our OpenMP to CUDA compiler.

[27] Using Microsoft Visual Studio 2005

Neither host nor device is operating in a dedicated fashion. The host is running the operating system, and the device is running the display.

## 9.6 Compiler Execution Runtime

When applied to the task of compiling the benchmark code, the bash **`time`** command, applied to **`java Main a1.c > out.cu`**, returns an elapsed time of 0.28, a user cpu time of 0.02, and a system cpu time of 0.02; the units are in seconds.

# Chapter 10

# Conclusions

The project sought to prove the concept of a C-based OpenMP compiler to generate parallelised output code in NVIDIA's general-purpose graphics processing language, CUDA. In combination with the hardware system's dual address space architecture, the free use of pointers in C presented a challenging environment for OpenMP, ultimately met by language extension and a single new directive, **`accessible`**.

A brief summary of development work undertaken as part of the project is described below. This is succeeded by considering some possible directions for future work which may advance the project's thesis. Finally a conclusive evaluation of the project's outcomes is presented.

## 10.1 Development Summary

- A language design was developed to allow the straightforward use of OpenMP parallel directives within C programs to target the NVIDIA G80 series of graphics cards. This was accomplished with the provision of a single new declarative directive, **`accessible`**, and four memory allocation library routines.
- Developed and tested a source to source compiler capable of translating C programs, using a subset of OpenMP, into a parallel form compatible with NVIDIA's CUDA C language.
- The grammar from the OpenMP specification appendix was converted to the ANTLR parser generator's Extended BNF-based syntax, and integrated with an existing C language grammar.

## 10.2 Future Work

The project has shown the feasibility of implementing a translator between the OpenMP and CUDA languages, though the performance advantage of this approach has not yet been demonstrated. It would therefore be valuable to focus on this, perhaps looking at the performance of more hardware-targeted data-parallel kernel translations, displaying a greater reliance on arithmetic intensity.

Persisting with a model of translation *outside* of the host compiler could be reconsidered. Were an integrated compiler developed, information about variable size and location could be obtained directly, and the need to extend OpenMP would disappear.

The use of OpenMP pointer list items has been a significant hurdle for the project. An alternative choice of input language could entirely avoid the issue. The HPC language, Fortran 77 already supports OpenMP, and is free of pointers. Translating Fortran to CUDA would though require a more elaborate translation process.

There are two lower-level NVIDIA CUDA APIs which could be considered as alternative back-end targets: the CUDA driver API, and the PTX virtual assembly language. Preliminary investigation could quickly evaluate the worth of such a proposition.

With new 256 and 512 processor versions of the G80 architecture imminent, NVIDIA CUDA is itself appearing as a worthy platform for HPC, at least in terms of Gflop ratings. It would be sensible to identify, develop, and benchmark computing science applications or kernels targeting the strengths of the CUDA system. It should also soon be possible to update the main memory while a kernel is running. Applications which take advantage of this could allow each kernel to run for longer, substantially reducing the relative cost of data transfer via the PCI Express bus.

Continued development of the existing translator also remains a valuable proposition. Outside of locks, simply implementing the remainder of OpenMP is a valid development goal. Sequential optimisation, such as coalesced device memory access, pinned host memory for transfers, or greater use of specialised CUDA elements are areas worth attention. Targeting a less granular transfer of data from host to device should also benefit performance.

## 10.3 Conclusion

The development of an OpenMP compiler, capable of translating parallel OpenMP C code into NVIDIA CUDA C code has been successfully completed. To accommodate the distinctive hardware architecture, OpenMP was extended by the addition of the **`accessible`** directive, allowing for the identification of both data and functions which ar eexpected by a user from within each **`parallel`** region.

Parts of OpenMP relating to synchronisation have proven intractable. There seems no likely route around this issue, and therefore the only solution is to accept the limitations it imposes on the valid code base.

Although performance has only begun to be evaluated, it does appear that arbitrary parallel code will not *automatically* gain advantage through the system. Sequential optimisation, and the targeting of less memory-bound application areas could help reduce these concerns.

# Appendix A

# ANTLR OpenMP Grammar

The following ANTLR EBNF grammar excerpt contains the rules required to parse the OpenMP v2.5 API. The embedded ANTLR “actions” used to aid the project translation have been omitted for clarity.

---

```
openmp_pragma
  : ( '#pragma' | '#' 'pragma' ) 'omp'
  ;

openmp_construct
  : parallel_construct
  | for_construct
  | sections_construct
  | single_construct
  | parallel_for_construct
  | parallel_sections_construct
  | master_construct
  | critical_construct
  | atomic_construct
  | ordered_construct
  ;

openmp_directive
  : barrier_directive
  | flush_directive
  ;

structured_block
  : statement
  ;

parallel_construct
```

```
  : parallel_directive structured_block
  ;

parallel_directive
  : openmp_pragma 'parallel' parallel_clauseoptseq
  ;

parallel_clauseoptseq
  : parallel_clause*
  ;

parallel_clause
  : unique_parallel_clause
  | data_clause
  ;

unique_parallel_clause
  : 'if' '(' expression ')'
  | 'num_threads' '(' expression ')'
  ;

for_construct
  : for_directive iteration_statement
  ;

for_directive
  : openmp_pragma 'for' for_clauseoptseq
  ;

for_clauseoptseq
  : for_clause*
  ;

for_clause
  : unique_for_clause
  | data_clause
  | 'nowait'
  ;

unique_for_clause
  : 'ordered'
  | 'schedule' '(' schedule_kind ')'
  | 'schedule' '(' schedule_kind ',' expression ')'
  ;

schedule_kind
  : 'static'
```

```
  | 'dynamic'
  | 'guided'
  | 'runtime'
  ;

sections_construct
  : sections_directive section_scope
  ;

sections_directive
  : openmp_pragma 'sections' sections_clauseoptseq
  ;

sections_clauseoptseq
  : sections_clause*
  ;

sections_clause
  : data_clause
  | 'nowait'
  ;

section_scope
  : '{' section_sequence '}'
  ;

section_sequence
  : ( section_directive? structured_block )+
  ;

section_directive
  : openmp_pragma 'section'
  ;

single_construct
  : single_directive structured_block
  ;

single_directive
  : openmp_pragma 'single' single_clauseoptseq
  ;

single_clauseoptseq
  : single_clause*
  ;

single_clause
```

```
  : data_clause
  | 'nowait'
  ;

parallel_for_construct
  : parallel_for_directive iteration_statement
  ;

parallel_for_directive
  : openmp_pragma 'parallel' 'for' parallel_for_clauseoptseq
  ;

parallel_for_clauseoptseq
  : parallel_for_clause*
  ;

parallel_for_clause
  : unique_parallel_clause
  | unique_for_clause
  | data_clause
  ;

parallel_sections_construct
  : parallel_sections_directive section_scope
  ;

parallel_sections_directive
  : openmp_pragma 'parallel' 'sections'
    parallel_sections_clauseoptseq
  ;

parallel_sections_clauseoptseq
  : parallel_sections_clause*
  ;

parallel_sections_clause
  : unique_parallel_clause
  | data_clause
  ;

master_construct
  : master_directive structured_block
  ;

master_directive
  : openmp_pragma 'master'
  ;
```

```
critical_construct
  : critical_directive structured_block
  ;

critical_directive
  : openmp_pragma 'critical' region_phrase?
  ;

region_phrase
  : '(' IDENTIFIER ')'
  ;

barrier_directive
  : openmp_pragma 'barrier'
  ;

atomic_construct
  : atomic_directive expression_statement
  ;

atomic_directive
  : openmp_pragma 'atomic'
  ;

flush_directive
  : openmp_pragma 'flush' flush_vars?
  ;

flush_vars
  : '(' identifier_list ')'
  ;

ordered_construct
  : ordered_directive structured_block
  ;

ordered_directive
  : openmp_pragma 'ordered'
  ;

threadprivate_directive
  : openmp_pragma 'threadprivate' '(' identifier_list ')'
  ;

data_clause
  : 'private' '(' identifier_list ')'
```

```
  | 'copyprivate' '(' identifier_list ')'
  | 'firstprivate' '(' identifier_list ')'
  | 'lastprivate' '(' identifier_list ')'
  | 'shared' '(' identifier_list ')'
  | 'default' '(' 'shared' ')'
  | 'default' '(' 'none' ')'
  | 'reduction' '(' reduction_operator ':' identifier_list ')'
  | 'copyin' '(' identifier_list ')'
  ;

reduction_operator
  : '+' | '*' | '-' | '&' | '^' | '|' | '&&' | '||'
  ;
```

---

# Appendix B

# Entry-Point Grammar Rules

The following two grammar rules parse the new OpenMP declarative directive, **`accessible`**. The first, **`gpump_pragma`** is required for compatibility with other OpenMP compilers, while the second matches the new **`accessible`** directive.

The alternative use of **`#pragma accessible`**, as a function qualifier, is handled by the C grammar's **`external_declaration`** and **`function_definition`** rules which follow. These, and the remaining rules are those of the ANTLR ANSI C Grammar which have been modified to accommodate the use of both the **`accessible`** and other OpenMP directives.

---

```
gpump_pragma
  : ( '#pragma' | '#' 'pragma' ) 'gpump'
  ;

accessible_directive
  : gpump_pragma 'accessible' '(' identifier_list ')'
  ;

external_declaration
options {k=1;}
  : ( (gpump_pragma 'accessible')?
      declaration_specifiers? declarator declaration* '{' )=>
      function_definition
  | declaration
  ;

function_definition
scope Symbols;
@init {
  $Symbols::types = new HashSet();
}
```

```
  : ( (gpump_pragma 'accessible')? // OpenMP extension 2 of 2
      declaration_specifiers? declarator
      ( declaration+ compound_statement // K&R style
      | compound_statement              // ANSI style
      )
  ;

declaration
scope {
  boolean isTypedef;
}
@init {
  $declaration::isTypedef = false;
}
  : 'typedef' declaration_specifiers?
    {$declaration::isTypedef=true;}
    init_declarator_list ';'
  | declaration_specifiers init_declarator_list? ';'
  | threadprivate_directive // OpenMP entrypoint 1 of 3
  | accessible_directive    // OpenMP extension 1 of 2
  ;

statement
  : labeled_statement
  | compound_statement
  | expression_statement
  | selection_statement
  | iteration_statement
  | jump_statement
  | openmp_construct // OpenMP entrypoint 2 of 3
  ;

statement_list
  :  (  openmp_directive // OpenMP entrypoint 3 of 3.
  | statement
  )+
  ;
```

---

# Appendix C

# OpenMP Loop Construct Example

The following example code is a slightly modified version of the loop construct example from the OpenMP specification document[1]. Array variable **a** is here initialised, to allow **b** to obtain comprehensible values; and the new **accessible** declarative directive is used, as is required when translating code using pointer list items.

An instrumented version of this code is used in the performance measurements. The next appendix entry also shows the translated, CUDA compatible version of this example.

```
void a1(int n, float *a, float *b);

int main(int argc, char *argv[])
{
  float a[1024], b[1024];
  #pragma gpump accessible(a,b)

  for (i = 0; i < 1024; i++)
    a[i] = 0.5f + (float)i;

  a1(1024, a, b);
}

void a1(int n, float *a, float *b)
{
  int i;

#pragma omp parallel for
  for (i=1; i<n; i++) /* i is private by default */
    b[i] = (a[i] + a[i-1]) / 2.0;
}
```

# Appendix D

# Translated OpenMP Loop Construct Example

The following NVIDIA CUDA code is the result of the command line instruction: **`java Main a1.c > a1t.c`**, which transforms the OpenMP loop construct example, **`a1.c`**, listed in the previous appendix section. The code has only been modified by the insertion of newlines, to ensure the listing remains within the document margins.

---

```
#include <stdio.h>
#include <cutil.h>
#include <gpump_all.h>
#include <gpump.cu>
#include <gpump_utils.cu>

typedef struct {
  int  i;
  int  n;
  float  *b;
  float  *a;
  void *__i, *__n, *__b, *__a;
}  gpump_params0;


__global__ void gpump_parallel0(gpump_params0 *gpump_li0);

void a1(int n, float *a, float *b);

int main(int argc, char *argv[])
{
  int i;
  float a[1024], b[1024];
```

```
  gpump_alloc_list_add(&a, sizeof a);
  gpump_alloc_list_add(&b, sizeof b);

  for (i = 0; i < 1024; i++)
    a[i] = 0.5f + (float)i;

  a1(1024, a, b);
}

void a1(int n, float *a, float *b)
{
  int i;


  gpump_params0 _gpump_li0;
  gpump_params0 *gpump_li0;

  gpump_copy_config_t cfg0[] = {
    { &_gpump_li0.i, &_gpump_li0.__i, &i, sizeof i,
            sizeof(int), {-1}, 0, 1 },
    { &_gpump_li0.n, &_gpump_li0.__n, &n, sizeof n,
            sizeof(int), {-1}, 0, 8 },
    { &_gpump_li0.b, &_gpump_li0.__b, &b, sizeof b,
            sizeof(float), {-1}, 1, 8 },
    { &_gpump_li0.a, &_gpump_li0.__a, &a, sizeof a,
            sizeof(float), {-1}, 1, 8 },
  };

  gpump_copy_to_device(cfg0,
            sizeof cfg0 / sizeof(gpump_copy_config_t));

  cudaMalloc((void **)&gpump_li0, sizeof _gpump_li0);
  cudaMemcpy(gpump_li0, &_gpump_li0, sizeof _gpump_li0,
            cudaMemcpyHostToDevice);

  gpump_parallel0<<<gpump_utils_get_num_blocks(),
                    gpump_utils_get_threads_per_block()>>>(
                    gpump_li0);
  cudaThreadSynchronize();

  cudaMemcpy(&_gpump_li0, gpump_li0, sizeof _gpump_li0,
            cudaMemcpyDeviceToHost);

  gpump_copy_from_device(cfg0,
            sizeof cfg0 / sizeof(gpump_copy_config_t));

}
```

```
__global__ void gpump_parallel0(gpump_params0 *gpump_li0)
{
  int gpump_temp;
  int  i;
  int gpump_lb, gpump_b, gpump_incr;
  int gpump_chunk_block, gpump_niters, gpump_iters_left;
  int gpump_npasses, gpump_pass;


  gpump_lb =    1;
  gpump_b =      (gpump_li0->n);
  gpump_incr =  1;
  gpump_niters = ceil((float)abs(gpump_lb - gpump_b) /
            abs(gpump_incr));
  gpump_chunk_block = (0) ? (0) * gpump_get_num_threads_d() :
            gpump_niters;
  gpump_iters_left = gpump_niters % gpump_chunk_block;
  gpump_npasses = (gpump_niters / gpump_chunk_block) +
            (gpump_iters_left ? 1 : 0);
  for (gpump_pass = 0; gpump_pass < gpump_npasses;
            gpump_pass++) {
    gpump_lb = 1 +
            (gpump_pass * gpump_incr * gpump_chunk_block);
    gpump_b = gpump_lb + ((gpump_pass == (gpump_npasses - 1) &&
            gpump_iters_left) ?  gpump_incr * gpump_iters_left :
            gpump_incr * gpump_chunk_block);
    gpump_utils_calc_static_for_bounds(&gpump_lb, &gpump_b,
            gpump_incr);
    for (i = gpump_lb; i != gpump_b; i += gpump_incr)
    (gpump_li0->b)[i] = ((gpump_li0->a)[i] +
            (gpump_li0->a)[i-1]) / 2.0;
  } // gpump_pass loop end
  gpump_barrier_d(); // implicit loop barrier



  gpump_li0->__i = &gpump_li0->i;
  gpump_li0->__n = &gpump_li0->n;
  gpump_li0->__b = &gpump_li0->b;
  gpump_li0->__a = &gpump_li0->a;
}
```

---

# Appendix E

# System Specifications

The following specifications describe the project's development and benchmark system. The information was obtained from the Windows XP System Properties, the CUDA runtime routine, **`cudaGetDeviceProperties`**, and the NVIDIA GeForce 8 Series webpage[15].

| **Attribute** | **Value** |
|---|---|
| Operating System | Windows XP SP2 |
| Processor | AMD Athlon 64 X2 Dual Core 4200+ |
| Clock Speed | 2.20GHz |
| Main Memory | 2GB RAM |
| Display Adapter | NVIDIA GeForce 8600 GTS |
| Global Memory | 256MB GDDR3 |
| Memory Clock | 1000GHz |
| Memory Interface | 128-bit |
| Memory Bandwidth | 32GB/sec |
| Multiprocessors | 32 |
| Firmware Version | 1.1 |
| Core Clock Speed | 675MHz |
| Shared Clock | 1450MHz |

# Bibliography


[1] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, Version 2.5, 2005.

[2] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, Version 1.0, 2007.

[3] T.Parr. *The Definitive ANTLR Reference: Building Domain Specific Languages*, First Edition, The Pragmatic Programmer, 2007, ISBN 0-9787392-5-6

[4] H.Schildt. *C++ The Complete Reference*, Second Edition, McGraw Hill, 1995 ISBN 0-07-882123-1

[5] Intel Corporation. Cluster OpenMP* User's Guide, Version 9.1, Rev 4.1, 2006.

[6] B.M.Werther & D.M.Conway. *A Modest Proposal: C++ Resyntaxed*, ACM SIGPLAN Notices 31: 11, pp. 74-82, 1996.

[7] H.Boehm, A.J.Demers & C.Uhler. *Implementing multiple locks using Lamport's mutual exclusion algorithm*. In ACM Letters on Programming Languages and Systems. Volume 2, Issue 1-4, 1993. pp. 46-58.

[8] NVIDIA Corporation. *NVIDIA Tesla - GPU Computing Technical Brief*, Version 1.0.0, 2007.

[9] R.Fernando, M.Harris, M.Wloka & C.Zeller. *Programming Graphics Hardware*, Eurographics 2004 Presentations, 2004.

[10] D.Göddeke, R.Strzodka, & S.Turek. *Accelerating double precision FEM simulations with GPUs*, Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique, 2005.

[11] D.R.Horn, M.Houston & P.Hanrahan. *ClawHMMER: A Streaming HMMer-Search Implementation*, Proceedings of the 2005 ACM/IEEE conference on Supercomputing, pp. 11, 2005.

[12] G.I.Egri, Z.Fodor, C.Hoelbling, S.D.Katz, D.Nogradi & K.K.Szabo. *Lattice QCD as a video game*, ArXiv High Energy Physics - Lattice e-prints, 2006.

[13] J.A.Kahle, M.N.Day, H.P.Hofstee, C.R.Johns, T.R.Maeurer, & D.Shippy. *Introduction to the Cell multiprocessor*, IBM Journal of Research and Development. Volume 49, Number 4/5, 2005.

[14] J.Held, J.Bautista & S.Koehl. *From a Few Cores to Many: A Tera-scale Computing Research Overview*. Intel Corporation White Paper. 2006.

[15] NVIDIA Corporation. GeForce 8 Series, **http://www.nvidia.com/page/geforce8.html**, 2007.

[16] NVIDIA Corporation. *The NVIDIA CUDA 1.0 FAQ*, **http://forums.nvidia.com/index.php?showtopic=36286**, 2007.

[17] Microsoft Corporation. *PCI Express FAQ for Graphics*, **http://www.microsoft.com/whdc/device/display/PCIe_graphics.mspx**, 2007.

[18] The ANTLRWorks website, **http://www.antlr.org/works**, 2007.

[19] The NetBeans website, **http://www.netbeans.org/**, 2007.