Cross-Paradigm Compilation across Programming Models: from Imperative to Asynchronous Graph

by

Mahsa Layeghi

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

Recent years have seen a rapid evolution in multi-core processor architectures. However, programming multi-core processors efficiently is a challenging endeavor [1], [2]. Either programmers empirically or ad hoc identify parallelization opportunities in their sequential code, then manually parallelize the implementation, or parallel programming paradigms (with appropriate tooling and support) must be employed to automate parallelization efforts. Despite encouraging efforts such as OpenMP [3] and OpenCL [4], we are far from automated parallelization.

Asynchronous Graph Programming (AGP) is a novel parallel paradigm that is amenable to automated parallelization for multi-core processors. However, its semantics are both foreign to most programmers, and too low-level to support software development at scale. Thus, this thesis explores the development of a cross-paradigm compiler, that can translate code in an imperative language (C) to AGP, allowing existing code-bases to be re-deployed and automatically parallelized, despite the semantics of their original language being sequential. Towards this, we define semantic transformations from (a sub-set of) C to AGP, and demonstrate the implementation of a compiler that implements those transformations. Our results show that it is indeed possible to transform C code into AGP code, and that transformed parallel implementations running on a multi-core system, for a suite of testing benchmarks,

outperform sequential code substantially.

Acknowledgements

First and foremost, I am grateful to Almighty God for leading me down the path of knowledge. I would like to express my deep and sincere gratitude to my research supervisor, Dr. Paulo Garcia for giving me the opportunity to do research and providing invaluable guidance throughout this research. I can't say thank you enough for his tremendous support and help. In every meeting we had together, I have been inspired by his continuance support, patience, and guidance which was vital for the completion of this project. I appreciate his help throughout the whole thesis at times when he was trying to put me in the right path. Special thanks to students of our research group for their valuable comments and suggestions especially to Sebastien with whom I have worked for a period of time before our thesis begins. I would like to express my heartfelt gratitude to my parents and my brother and sister, for their wise counsel and widespread support throughout my study in Canada. I would like to thank my fiance, Soroush, who has been supporting me all the time and he has always been source of inspiration and strength for me. Finally, I would like to thank my friends, Amirali, Parnia and Jasmine, for providing me with happy distractions to rest my mind outside of my research and having a group study sessions together on campus.

Contents

List of Tables

List of Figures

List of Source Codes

Acronyms

AGP Asynchronous Graph Programming

ARM Advanced RISC Machines

DSP Digital Signal Processing

GPU Graphical Processing Unit

CPU Computer Processing Unit

API Application Programming Interface

OpenMP Open Multi-Processing

OpenCL Open Computing Language

CUDA Compute Unified Device Architecture

HiCUDA High-level Compute Unified Device Architecture

LEO Language Extensions for Offload

GPGPU General-Purpose Graphics Processing Units

Chapter 1

Introduction

1.1 Multi-Core Heterogeneous Architectures

Over the last years, the design of Multi-processor System-on-Chips has moved towards heterogeneous architectures [5]. A heterogeneous architecture is defined as an architecture where multiple types of cores with various attributes (such as Instruction Set Architectures, or clock frequencies) are merged together; They are generally comprised of two types of processors; A) general-purpose processors such as ARMs (called host cores), and B) special-purpose cores such as DSPs or GPUs (called accelerators). For applications to be able to exploit the full potential of multi-core heterogeneous architectures, some directions are being pursued:

(i) New parallel programming paradigms

(ii) Automatic parallelization frameworks

Directive-based programming models are one of the new programming paradigms which are used for parallelization. In heterogeneous architectures, these paradigms

usually follow a host-centric approach in which compiler directives are used to indicate code sections that could be offloaded to accelerators. ([6])

The current programming practices require several manual steps for developers despite the effort to provide convenient programming paradigms. Among these manual steps, there are a few ones that are extremely time-consuming and error-prone processes. Some notable examples of the steps are identifying computationally intensive code sections, data dependency analysis, identifying parallelization opportunities, determining the target processor to execute a specific code region on, and implementing the optimized code. Thus, some frameworks have been introduced to automate these processes.

To run parallel programs or have multi-programmed workloads, a multicore system can be used. In this scenario, each core runs a different application than the other ones, and also the the individual applications usually have nonidentical characteristics which makes using a multicore system even more useful. In 1990s, multicore processors had been introduced in order to improve the performance efficiency. By that time, chipmakers had realized that improving the performance of a single-core processor was becoming a challenging task since it was resulting in more power consumption, complexity, and an increased area [7].

1.2 Parallelization Strategies

Novel heterogeneous multicore adaptable software architectures are counted as reconfigurable and adaptable systems which are meant to guarantee tomorrow's performance and power consumption [8]. However, current design practices, software frameworks,

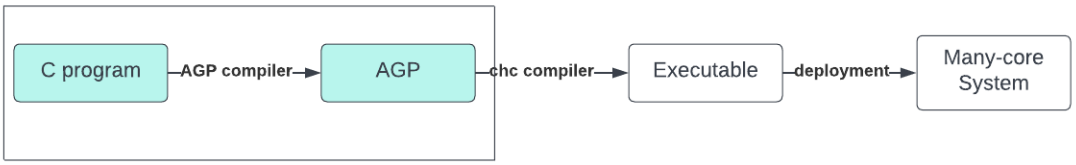
programming languages, and hardware artifacts are not developed enough to support these systems as it might require evolving hardware [9], deployment across heterogeneous platforms [10], approximate computing [11], or performance-driven design [12]. To tackle these problems, some new strategies have been utilized. Among them, a graph-based programming paradigm, called Asynchronous Graph Programming, has been suggested. AGP and its associated model of computation have been introduced to implement reconfigurable, adaptable software across heterogeneous multi-core systems [13].

As multi-core architectures become prevalent, an increasing number of researching effort is focused on building compilers that uses multi-core computers in the most efficient way and automatic palatalization has been the main point of focus in the researches [14] [15]. In a nutshell, automatic parallelization means that a compiler takes an unaltered and unannotated sequential program and compiles it into a parallel program. The goal of parallelizing compilers is to reduce the number of computations in a program, reduce the storage usage, and most importantly recognize sections of the code that could be performed in parallel. However, all these improvements come at a cost: This makes it more challenging for a programmer to effectively utilize the parallelizing compilers as these types of compilers require more knowledge and effort from the programmer to master the language and the art of utilizing it. To minimize the learning curve of a new computing language, and increase the code reusability, portability, source-to-source compilation has been presented in recent years.

1.3 Contribution

In this thesis, we have introduce a C-to-AGP compiler which is taking C as a commonly-used programming language and will generate AGP code. The AGP project toolchain is comprised of four main steps which has been depicted in figure 1.1. The chc compiler generates executable following some certain transformation rules which we are not going to explain as it is outside the scope of this research. In the next step, the executable is deployed on processing hardware components (a many-core system). However, in this thesis, we focus on presenting a modern source-to-source translator which takes a text representation of a source program (C) and will output a parallel program (Asynchronous Graph Programming) and we are interested in choosing a programming language which is widely being used as AGP may not be known to developers. This method will be described in more detail in section 2.

Figure 1.1: AGP Project toolchain



The work done in the thesis presents an algorithm to translate the source program (C) to the destination programming model (AGP). Then it will investigate parts of the code that could be parallelized and will describe strategies to convert loops, recursive functions, and so on, to corresponding instruction in the AGP programming model. Converting C to AGP, allows us to take existing code bases and generate automatic parallelization. Since we are utilizing from AGP model and the rest of the AGP project toolchain (i.e. the AGP-to-manycore compiler), the entire source

code is being parallelized under the hood without programs being worried about parallelization. Hence, the sequential C codes can benefit from parallelization and be run on a many-core heterogeneous architecture. Next, we show the output result of our compiler for a sample C code and three main benchmarks. Finally, the similarity between the generated outputs from the compiler will be compared with the expected output based on the algorithm.

Similar to what other parallelizing compilers have offered so far, our compiler is also widely usable in parallelizing research and shared memory systems as it can compile a sequential code. Specifically, in this thesis:

- we have built a compiler that takes C and outputs AGP.
- we have presented an algorithm for such a translation
- we empirically evaluate the code's output from the compiler with the results from our algorithm.

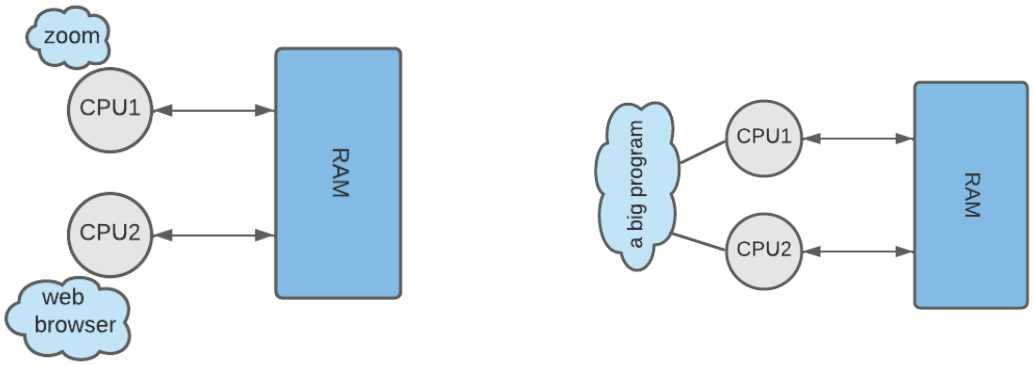
The remainder of this thesis is organized as follows: Chapter 2 describes Asynchronous Graph Programming (AGP) model which is also the output of our Compiler. Chapter 3 describes the related works that has been done in the field, to place our contribution in perspective. Chapter 4 presents our algorithm which translates one language to another. chapter 5 presents our results and discusses those results in context. Finally, in chapter 6 we conclude this thesis and highlight future possible works in this field.

Chapter 2

Background

2.1 Modern Multi-Core Processors

Modern multi-core processors (e.g. dual-core, quad-core, etc.) have shown a massive improvement in executing parallel applications; A general scheme of multi-core processors has been shown in figure 2.1 where two CPUs are connected to a same piece of memory. This multi-core architecture works perfectly fine when we have two programs that are completely independent of one another. An example of these independent applications is shown in figure 2.1 (a), where for example two separate applications (Zoom and a web browser) are running in parallel on each CPU. Conversely, executing a single giant and complicated program on such multi-core architecture for achieving a faster execution time is still an open problem (figure 2.1 (b)); In other words, how to efficiently parallelize a single program (*parallelizing the code*) is not fully solved.



(a) (b)

Figure 2.1: (a) Independent Applications (b) One Big Application

2.2 Parallelizing Compiler

A novel parallelizing compiler *chc* (publicly available here[1]) is a first step toward parallelizing the code; The compiler aims to be used in a multi-core or distributed systems and it compiles a recently proposed programming model (Asynchronous Graph Programming) which we'll refer to it as AGP in the next chapters. In summary, AGP differs from existing languages in that it can be easily parallelized. Since AGP is not a publicly-known language, proposing a compiler to translate a widely-used language to AGP would be efficient to utilize the parallelizing features of AGP. The work done in this thesis has presented a source-to-source translation from an imperative language (like C language) to AGP (the source language of the chc compiler).

In the following, the source language to the chc compiler (AGP) will be introduced.

[1] `https://github.com/paulofrgarcia-carleton/chc-public-release`

2.3 AGP Programming Model

Asynchronous Graph Programming (AGP) [16] is a novel programming model in which a program has been defined by a directed graph. With having useful features, AGP is not considered a user-programmed language; Alternatively, it accounts for an intermediate representation that other compilers could generate code in. The nodes in the graph hold data that can be of different types: They could be a special input or output of a subgraph (i.e. just like function arguments and function return values), or they could represent an *operation* on data (e.g. arithmetic, logical, relational, conditional), or an expansion node which is used for subgraph calls with special values. AGP follows a single-assignment expression meaning that only one value can be assigned to a node and for each assignment, an individual node should be *constructed*. Edges in the graph show the data transmission from one node to another. This also tells how nodes are dependent on one another and could have 0 to n number of dependents.

AGP can be evaluated asynchronously and yet guaranteeing the correct results when being executed. Hence, the order in which consecutive sequential instructions being executed on multiple evaluation threads or processes do not matter in AGP.

AGP is recognized as a programming model which has few instructions that suffice the execution of a program. The AGP instruction set is listed in table 2.1. All of the instructions have a list of inputs to perform a specific task. The number of inputs for an instruction varies; *OPERATOR* instruction, for example, takes four inputs which are the result of an operation node, the type of an operation, first and second operand respectively. Types of operators for *OPERATOR* instruction has shown in table 2.2. On the other hand, the inputs to another instruction such as *EXPAND* is arbitrary such that its inputs are the name of the subgraph to be called and a special operator

for mapping each input and output to their corresponding inputs and outputs in the callee subgraph (Table 2.3).

Instruction	Description
SUBGRAPH	defines a new subgraph
DATUM	defines a new variable
INPUT	defines a new input for a subgraph
OUTPUT	defines an output of a subgraph
CONST	defines a new constant
EXPAND	is used to call a subgraph
OPERATOR	is used to perform an operator on two nodes
TERMINATE	is used to terminate outputs of a subgraph

Table 2.1: AGP Instruction Set

Operator	Description
op_PLUS	is used to add two nodes' value
op_MINUS	is used to subtract two nodes' value
op_TIMES	is used to multiply two nodes' value
op_IF	constructs an if statement
op_ELSE	constructs an else statement
op_ISEQUAL	is used to check if two nodes' value are equal
op_ISLESS	is used to check if a node's value is less than the other one
op_GREATER	is used to check if a node's value is greater than the other one

Table 2.2: Inputs of OPERATOR instruction

Operator	Description
MAP_IN	is used to map an input to a corresponding node in the caller subgraph
MAP_OUT	is used to map an output to a corresponding node in the caller subgraph

Table 2.3: Inputs of EXPAND instruction

To realize how AGP works, we will show what AGP consists of using a few examples. AGP is said to be a procedural programming paradigm in a way that it consist of routines and subroutines which are meant to be called during the execution of a

program. In AGP, these subroutines are called *subgraphs* and are shown in listing 2.1; As it can be seen, subgraphs take names which defines what the subgraph should be called. This is almost similar to what other general-purpose languages are using to define subroutines with no parameters passed to a subgraph and no return value being different. Alternatively, an individual instruction is being used to define any possible parameter passed to and also being returned from a subgraph. This is illustrated in listing 2.1 where foo subgraph takes x as an input and will output equivalent result (*INPUT* and *OUTPUT* instructions respectively). The *DATUM* instruction is being used to declare the variables x and $result$.

```
SUBGRAPH(foo)
    DATUM(x);
    INPUT(x);
    DATUM(result);
    OUTPUT(result);
SUBGRAPH(main)
```

Source Code 2.1: Defining SUBGRAPHS in AGP

2.3.1 Arithmetic Operators in AGP

Arithmetic operators are supported in AGP. However, currently, AGP accepts integer values and will only operate on them. The available and supported arithmetic operators are Addition, Subtraction, and Multiplication. Listing 2.2 shows the equivalent AGP code for mathematical expression:

$$result = (x - 1) + (x + 1) + (x * 1)$$

This example will show some of the important features AGP has. AGP can only take two operands at a time and perform operations on them. Moreover, AGP does not support multiple assignments meaning that the variables can only be used once. So in the following example, the addition of $x - 1$ and $x + 1$ cannot be put in *result* and the old value of *result* be used again to be added to $x * 1$. Instead, a new DATUM node such as *temp* should be created to hold the result of $x - 1$ and $x + 1$.

```
SUBGRAPH(calculateSum)
    DATUM(x);
    INPUT(x);
    DATUM(result);
    OUTPUT(result);
    CONST(one,1);
    DATUM(xminus1);
    DATUM(xplus1);
    DATUM(xtimes1);
    DATUM(temp);
    OPERATOR(xminus1,op_MINUS,x,one);
    OPERATOR(xplus1,op_PLUS,x,one);
    OPERATOR(xtimes1,op_TIMES,x,one);
    OPERATOR(temp,op_PLUS,xminus1,xplus1);
    OPERATOR(result,op_PLUS,temp,xtimes1);
```

Source Code 2.2: Arithmetic Operators in AGP

2.3.2 Relational Operators in AGP

AGP also supports relational operators which includes equality, greater than and less than operations. These behaviour is shown in listing 2.3.

```
SUBGRAPH(foo)
    DATUM(x);
    INPUT(x);
    CONST(zero,0);
    DATUM(xis0);
    DATUM(xgreater0);
    DATUM(xless0);
    OPERATOR(xis0,op_ISEQUAL,x,zero);
    OPERATOR(xgreater0,op_ISGREATER,x,zero);
    OPERATOR(xless0,op_ISLESS,x,zero);
```

Source Code 2.3: Relational Operators in AGP

2.3.3 Conditional Statements in AGP

Conditional statements are vital to any program, to control the flow of the program depending whether or not given expressions are satisfied. Like other general-purpose languages, AGP also possesses if and else statements. Listing 2.4, shows the equivalent AGP code for if else statement such as:

$$if(x == 0)y = 1elsey = 2returny;$$

```
SUBGRAPH(foo)
    DATUM(x);
```

```
3     INPUT(x);
4     DATUM(y);
5     OUTPUT(y);
6     CONST(zero,0);
7     CONST(one,1);
8     CONST(two,2);
9     DATUM(xis0);
10    DATUM(resulttrue);
11    DATUM(resultfalse);
12    OPERATOR(xis0,op_ISEQUAL,x,zero);
13    OPERATOR(resulttrue,op_IF,xis0,one);
14    OPERATOR(resultfalse,op_ELSE,xis0,two);
15    OPERATOR(y,op_MERGE,resulttrue,resultfalse);
```

Source Code 2.4: Conditional Statements in AGP

In line 13, if xis0 equals to 1 (i.e. that the condition is true), one will be assigned to *resulttrue* node. Otherwise, (xis0 is 0 which means the condition is false) and therefore, two will be assigned to *resultfalse* node (line 14). The last line shows a MERGE operation which is the most important instruction in if else statements where it eventually choose one of the results over another.

2.3.4 Subgraph Calls in AGP

Any subgraphs in the AGP program would be considered pointless if they are never invoked. Therefore, an instruction to call a subgraph has been defined in AGP; As shown in listing 2.5, *EXPAND* is used to call the foo subgraph with *MAP_IN* and *MAP_OUT* as a function to map each individual input and outputs of the subgraph

with the corresponding node in the caller subgraph respectively. As a result, in the following example, x in the foo subgraph will be replace by foo_in which is 5 and the output node in foo subgraph will be transmitted to foo_out in the main subgraph.

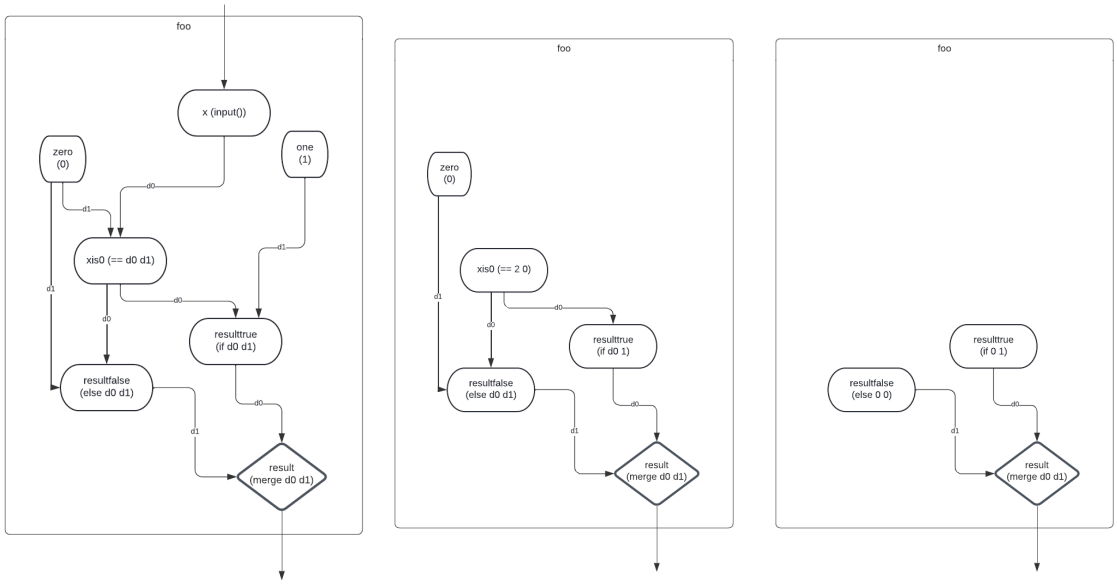
```
SUBGRAPH(foo)
     DATUM(x);
     INPUT(x);
     DATUM(y);
     OUTPUT(y);
     CONST(one,1);
     OPERATOR(y,op_PLUS,x,one);

SUBGRAPH(main)
    CONST(foo_in,5);
    DATUM(foo_out);
    EXPAND(foo,MAP_IN(x,fact_in);MAP_OUT(y,foo_out););
    OUTPUT(foo_out);
```

Source Code 2.5: Subgraph Calls in AGP

We have provided an example to illustrate what an AGP program looks like (Listing 2.6). The output of the foo subgraph is the result of merge in line 7. *Merge* is used to combine the result of if and else statement which eventually choose one result over the other. *Result_true* happens when x is equal to one. It is worth mentioning that the result of equality of x and zero will be put into another node and that explains why AGP is a single assignment language. On the other hand, *result_false* occurs when x is anything other than zero.

During the execution of a program, nodes can be constructed, unconstructed or

destroyed; A node is said to be constructed if all their dependencies have been constructed, unconstructed if their corresponding value has not yet been computed, or destroyed if it can never be constructed. For example, in an If-Else statement, which is a boolean test, one of these nodes (if or else node) is destroyed based on the if condition. Such behaviour can be seen in figure 2.2b where if node will be destroyed as its condition is not satisfied. As the result of a destruction, all the subgraphs that depend on the destroyed data, will also be pruned and the nodes and subgraphs dependent to the other node will be constructed. Merge is another special operation which combines the result of parallel paths and put the result into a new node, and the value of which is the value of the first constructed operands (figure 2.2c).



(a) (b) (c)

Figure 2.2: Visual representation of the foo graph and its evaluation. a: initial (compile time) graph. b: evaluation for input $x = 2$ after one step; c: evaluation after two steps

---

```
subgraph(foo)
  node x <- input;
```

```
node result <- output;
node xis0 <- (== x 0);
node result_true <- (if xis0 1);
node result_false <- (else xis0 0);
node result <- merge result_true result_false;
output <- result;
```

Source Code 2.6: "foo" function highlighting some aspects of AGP. Pseudo-syntax for legibility.

2.4 Conclusion

The AGP programming model has been shown to have features that make it suitable to be used in parallelizing compilers; Since AGP is mostly considered as an intermediate representation, it's preferable to provide a compiler for such a programming model that can convert any general-purpose and widely-used language to AGP. In this thesis, we will introduce a compiler (source-to-source translator) from an imperative language (C) to AGP. The chosen source language could have been anything else; however, for the purpose of this research, we have chosen C since it has more similarity to AGP and therefore can be compiled to it straightforwardly. The intermediate representation (AGP) will then be forwarded to the chc compiler to compile the graph programming model to a stack-based structure. The validity and correct execution of the generated code has then been evaluated using a simulator. However, the post-operations performed on AGP are beyond the scope of this research; instead we focus on how to generate AGP code so that it can be exploited in further research studies.

Chapter 3

Literature Review

In recent years, the proliferation in the number of processing elements has made parallel programming models and parallelizing compilers a famous approach in designing embedded devices (often heterogeneous), many-core and multi-core processors, and distributed systems comprised of thousands of interconnected processors. In this chapter, we will initially explain some of the existing parallel programming models which have been extensively used for parallelization; then, we will investigate some of the source-to-source parallelizing compilers, and will highlight their unique and striking features.

3.1 Parallelizng Compilers

A "parallelizing compiler" is typically a compiler that finds parallelism in a sequential program and generates appropriate code for a parallel computer [17]. The aim of parallelizing compilers is to increase the efficiency of a program by recognizing the segments of the code which has the potential to be paralellizable (e.g. explicitly parallel language constructs, such as array assignments or parallel loops). Facilities

offered by these compilers make shared memory parallel programming extremely easy since all programmers need to do is to use a parallelizing compiler and they will have a correct parallel program. The process of generating a parallel code would take sophisticated compiler analysis. In order to alleviate this process, some paralellizing compilers have been introduced. These compilers could help in automating some of the code analysis techniques, and their ultimate goal is to generate an optimized and parallel code which eventually will be executed on a target architecture. Before we go through parallelizing compilers, we are going to explore some of the parallel programming models which have been used as a source/destination of these compilers.

3.2 Parallel Programming Models

As we explained earlier, a programming model is an abstraction provided by the hardware to programmers and It determines how easily programmers can specify their algorithms into computation tasks that the hardware understands, and how efficiently those tasks can be executed by the hardware. [7].

There are programming models for both uniprocessor and multiprocessor systems. The focus on this chapter is on the programming models for multiprocessor systems amongst which two major programming models that are widely used are *shared memory* and *message passing*. In a shared memory model, parallel tasks (i.e. Tasks which are running independently from others and are being executed on various processors) can have access any location of the memory and as a result the communication between them is through reading and writing common memory locations. In a message passing model, the parallel tasks have their own piece of memory, which other tasks cannot access. Hence, in order to communicate with each other, tasks should send messages

to each other. [18], [19].

Although many parallel programming models exist, we only highlight some of those that are popular and have been using in designing parallel architectures. One of the most popular directive-driven programming models to write shared-memory parallel programs is OpenMP, which provides a simple and flexible interface for developing parallel applications ([20], [3]). OpenMP (Open Multi-Processing) is another programming paradigm. It is an application programming interface (API) that supports shared memory programming. C/C++ and Fortran are currently supported programming languages by OpenMP. The OpenMP standard has a set of compiler directives with which programmers can express parallelism to a compiler that supports OpenMP. The compiler then replaces these directives into code that calls library routines or reads environment variables that eventually influence the run-time behavior of the program. OpenCL (Open Computing Language) is another popular parallel programming models which provides a common language, programming interfaces, and hardware abstractions which is enabling developers to accelerate applications with task-parallel or data-parallel computations in a heterogeneous multi-core architectures ([21][4]). One of the other programming models is Compute Unified Device Architecture (known as CUDA) which is an extension to the C language that allows GPU code to be written in regular C and the code is either targeted for the host processor (CPU) or the device processor (GPU) [22]. These were only some of the famous parallel programming models that have widely being used. However, a numerous efforts and research revolves around designing a new parallel programming paradigms to make the best out of parallelism. Having said that, our new programming model (AGP) have also been introduced and proved to be a suitable paradigm which can be used in many-core, multi-core, and heterogeneous architectures.

These programming models act as a bridge between programmers and parallel architectures. [23] and many of them can be utilized to meet specific criteria such as performance in heterogeneous many-core architectures. Although parallel programming approaches such as OpenMP [3] and OpenCL [4] have emerged to help programmers to exploit parallelism in multi-core processors, they still require programmers to identify the potential parallelizable regions in their code and apply a parallel programming model to the selected regions. This is a non-trivial task which requires the programmer to have thorough knowledge of the parallel programming model and the target architecture [24]. For example, in a directive-based parallel programming model such as OpenMP, the main critical issue is that feasible parallelizable regions (e.g., loops) needs to be addressed by programmers. Consequently, to address these problems and to make the most out of parallelism, one of the methods which has been using is the **source-to-source compilation**. By translating the code in one of those programming models to another, code reusability, programmability, portability, and performance will be increased. Simultaneously, the learning curve of a new computing language will be reduced.

In the following, we will explore some existing code translation techniques between parallel programming models on heterogeneous many-core architectures. We will then introduce our novel code translator which translates a C code to AGP (which is a parallel programming model has been described in detail in [16]).

3.3 Code Translation Between Programming Models

As we explained earlier, source-to-source compilation between programming models assists developers not to worry about the challenges such as knowing the target architecture, dealing with fundamentals of parallel programming such as data dependencies, load balancing, synchronization and race condition required by some of the sophisticated parallel programming models. High-level programming frameworks, such as OpenMP, have been proposed to alleviate the difficulties of parallel programming. Yet, there is evidence that novice parallel programmers make common mistakes that may lead to performance degradation or unexpected program behavior [25].

Some of the code translations between parallel programming models on heterogeneous many-core architectures is trying to support the developers ([26], [27], [25]) or to provide automatic parallelization ([28], [29]). For example, authors in [25] have presented a cognitive parallel programming assistant (PAPA) which aims at educating and assisting novice parallel programmers to avoid common OpenMP mistakes.

3.3.1 CUDA Compilers

Some research has been done to design a parallelizing compiler that outputs CUDA. As we mentioned earlier, CUDA (developed by NVIDIA) provides a multi-threaded parallel programming model which enables writing scalable parallel programs using a straightforward extension of the C language [30]. Although CUDA offers a bunch of features which has been facilitating implementations of general-purpose computations on GPUs, the manual development of high-performance CUDA code still requires a large amount of effort from programmers. Hence, the automatic code translation

from sequential input programs into CUDA, would not let the programmers bothering themselves to learn CUDA, plus benefiting from what CUDA offers.

The researches in [31], [32], [33] and [34] have generated a source-to-source compiler which converts sequential codes to CUDA. The works done in these researches have tried to benefit from parallelism in different ways, including code generation for the parallelization of imperfectly nested loops, disambiguation of pointers which could hinder automatic parallelization, or presenting a new algorithm to exploit memory hierarchy. In order for manufacturers to invest largely in CUDA codes and yet take advantage of wider deployment opportunities afforded by OpenCL, a compiler to transfer CUDA to OpenCL (and vice versa) has been introduced. The authors in [35] have introduced a translator which translates the code from CUDA to OpenCL and from OpenCL to CUDA by identifying the similarities and differences between these two programming models. Authors in [36], have implemented an automated CUDA-to-OpenCL source-to-source translator (CU2CL) which enables portability of CUDA software to platforms that are not supported by CUDA.

3.3.2 Optimization Compilers

Other code translation methods have focused on optimization and reducing programming barrier. Some of these programming barriers has been identified in [37] and [38]. For example, Ueng et al. [37] introduced a translator whose aim is to relieve programmers of the burden of dealing with the complex memory hierarchy of CUDA; CUDA-Lite is an enhancement tool to CUDA which takes a program that has been parallelized for CUDA using only global memory and outputs a CUDA code with explicit memory-type declarations.

When programming in CUDA, some of the barriers programmers face are explicitly

managing data transfer between the host memory and various components of the GPU memory, and also to manually optimize the utilization of the GPU memory. Authors in [38] have developed HiCUDA (high-level CUDA) which is a directive-based language for programming NVIDIA GPUs and aims to alleviate the tasks mentioned above in a simple manner, and directly to the original sequential code by providing high-level abstractions.

Apricot [39] is another optimizing compiler which aim to minimize developer effort. Apricot is a compiler for x86-compatible many-core coprocessors and it assists programmers by performing strategies such as offloading some of the code regions to coprocessors at runtime based on a cost model that the authors have developed, as well as minimizing the data communication overhead and improving performance by applying a set of optimization techniques. Moreover, the Apricot compiler automatically inserts LEO (Language Extensions for Offload) clauses for parallelizable code regions. Although Apricot covers simple pointers and arrays, it lacks supporting nested arrays and structures of pointers.

Authors in [40] have presented a compiler framework from OpenMP to General-Purpose Graphics Processing Units (GPGPU). In order to have an automatic source-to-source translation, the authors have used several strategies that will result in a conversion from the loop-level parallelism of OpenMP into the data parallelism model of CUDA. Like many other compilers introduced, this compiler aims at offering an easier programming model for general computing on GPGPUs, as well as transformation techniques for efficient GPU global memory access.

3.3.3 Adaptation Compilers

The CUDA programming model has limited programmers to write specialized code which only executes on certain GPU devices. This is considered undesirable, as the application aimed to be written in a data-parallel language for GPU cannot be effectively parallelized across multiple CPU cores. In order to address this issue, a number of compilers have been introduced. These compilers are trying to adapt CUDA or OpenCL programming model to multi-core CPUs.

CUDA programming model has proven effective in programming GPUs. One of the compilers that falls into the category of adaptation compilers is MCUDA. MCUDA [41] is a framework which allows CUDA programs to be executed efficiently on shared memory, multi-core CPUs by operating some transformation techniques such as transforming a thread block into a serial function or replicating thread-local data. Diamos et al. [42] designed a dynamic compiler from PTX to Multi-core x86 CPUs (Ocelot) which maps the explicitly data parallel execution model used by NVIDIA CUDA applications onto diverse multi-threaded platforms. MCUDA and Ocelot are similar as they allow for CUDA kernels to be run on CPUs. However, Ocelot takes the approach of performing translations on lower-level byte-codes.

3.4 Conclusion

In this section, several compilers have been introduced; these source-to-source compilers have designed to optimize programmability, portability, and performance. Moreover, some of them aimed to minimize the learning curve of a new computing language as writing codes using lower-level programming models such as CUDA is a burden for application scientists.

In this thesis, we have designed a new compiler that translates an imperative language (C) to a programming model (AGP). AGP is a programming model that has been described in section 2.3. The compiler takes a C code as an input and will output AGP code which will then be benefited from the features of the parallelization that AGP offers. Just like many of the proposed compilers so far, our compiler minimize the burden of learning a new language (AGP) for programmers and will benefit from all the parallelization features of the AGP.

Chapter 4

Implementation

In this chapter, we will propose an algorithm for a translator that takes a C code and will output an AGP programming model. Before introducing the algorithm, it is important to show that not all features of C code are translatable to AGP. In the following, we will point out the characteristic in the source language and will study whether they are supported in the destination language and if so, what would be the equivalent instruction.

4.1 AGP Semantics

In this section, the semantics of AGP will be introduced. As mentioned in section 2.3, AGP is not a user-programmed language meaning that it can be used within compilers as an intermediate representation; other popular programming languages can be compiled into AGP and will benefit from the features it provides. In the following, we will cover how AGP works.

A *FUNCTION* in C can output modified data in three different ways; local

variables, global variables or pointers. In AGP, however, they are all referred to as an *OUTPUT*. Whether a function is writing to a global variable, or writing to an address pointed by a pointer, or simply returning a value, it all can be translated as a data being transferred from one place to another and the OUTPUT suffices in AGP to satisfy each possible return value in C. A subgraph must have at least one output and cannot be of void type. The same applies for function parameters which could be actual/global variables and pointers; An *INPUT* is used to represent all of the function parameters.

```
function foo
    (x)
    return
        (x*2)
endFunction
```

Source Code 4.1: Local

```
x, Global
function foo ()
    return
        (x*2)
endFunction
```

Source Code 4.2: Global

```
function foo
    (*x)
    return
        (*x)*2
endFunction
```

Source Code 4.3: Pointer

In listing [4.1 - 4.3], three ways of outputting a variable in C have been shown. Listing 4.1 doubles and returns the input locally. In the global example (4.2), the foo function does not take an input and instead it doubles a global variable and returns that. Listing 4.3, however, will get the address of a variable and and will double it in that specific address. All of these three ways of outputting the variable in C, will be translated as OUTPUT statement in AGP. In other word, in AGP, a datum is passed from one place to another place.

LOOPS: The traditional for and while loops are absent from AGP. Before going into the conversion of loops in AGP, we need to figure out how many types of loops we have:

```
function foo (i)
    for (i = 0, i <= N, i++)
        array[i] <- array[i] +
            1
    endFor
return array
endFunction
```

Source Code 4.4: Data Parallel Loop

```
function foo (i)
while (i >= 0)
    acc <- acc + i
    i <- i - 1
endWhile
return acc
endFunction
```

Source Code 4.5: Processing Loop

- *Data Parallel Loop* where no data is being reused and the loop is just for traversing something (e.g. the array). For converting these types of loops to an AGP code, a method called loop unrolling is being used. This is shown in listing 4.4 where the for loop can be unrolled and a set of multiple data parallel statements will be replaced to perform the same functionality.

$$array[0] = array[0] + 1$$

$$array[1] = array[1] + 1$$

. . .

$$array[N] = array[N] + 1$$

In the example of matrix multiplication which consists of three nested for loops, the resultant matrix which is the product of two matrices, could be computed with parallel additions and multiplications. Hence, matrix multiplication falls into the data parallel loop category.

- *Processing Loop:* in this type of loop, the same memory location is being used

to do some computations. In other words, the computation which is done in the next iteration of the loop is dependent on the current iteration of the loop. So as it is shown in listing 4.5, the foo function is supposed to return the sum from 0 to i; The while loop iteration is completely dependent to the value of i. Therefore, the computations are not going to be parallelized due to reusing of the same memory (acc) in the next iterations of the loop. We will see shortly that this types of loops have the potential to be rewritten as a recursive function; Little further in this chapter when explaining the conversion algorithm from C to AGP, we will show that the conversion of recursive functions in C to an equivalent code in AGP is straightforward and well-supported.

One of the features of AGP is that it is single assignment meaning that variables could not be reused and we are not allowed to re-assign a new value to an existed variable. Instead, we should define a new variable and put the updated value in the newly-defined variable. The reason for that is that unlike C language which is a multiple assignment language, AGP is a single assignment language. For example, for the $x++$ statement in C, there should be two variables ($x1$ and $x2$) where $x1$ posses the current value of x and $x2$ will store the new value of x. ($x2 = x1 + 1$)
To convert an array in C to AGP, each element of an array would be counted as a single variable definition. So an array arr of size 4 ($arr[4]$), as an example, would be considered as four individual variable definition ($arr[0]$, $arr[1]$, $arr[2]$, $arr[3]$). Later on we will see how variable definitions in C are converted to AGP code.

```

function foo (x)

if (x%2 == 0)
```

```
        y = 1
    else
        y = 0
    endIf

        return y

    endFunction
```

Source Code 4.6: if else statement

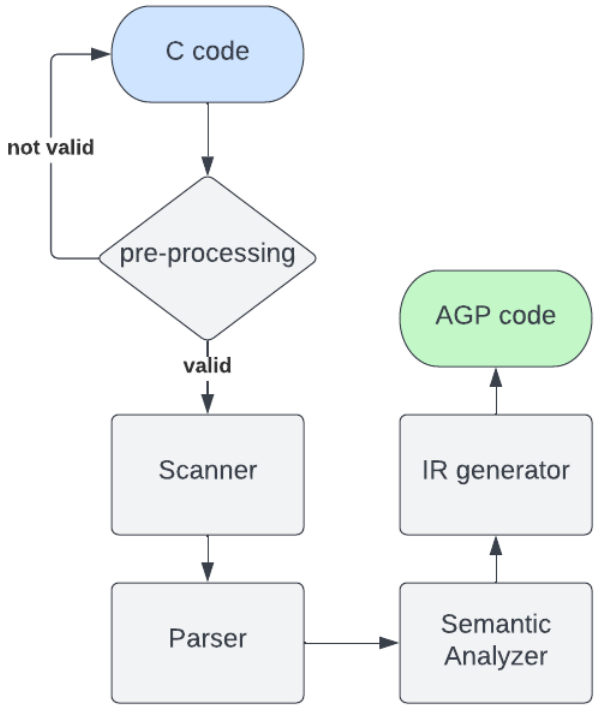
There is no if/else statement in AGP. Instead there is a *MERGE* operator which merges the results of either an if or else statement in a node. In listing 4.6, y is the output of the function which will eventually be either 0 or 1; An equivalent statement in AGP, is a MERGE operator with y being 1 (if statement's result) or 0 (else statement's result). The C switch statement which does the same as multiple if/else statements, can obey from the same rule in AGP. The exact instruction for if/else statement will be shown in C to AGP algorithm.

Each function call in C could be converted to an equivalent *EXPAND* instruction in AGP. AGP also allows to have a recursive function call by expanding a subgraph into it self with its own name. When using *EXPAND* instruction, name of the subgraph to be called, mapping of inputs and outputs should be specified. The exact instruction for a function call will be shown in C to AGP algorithm.

4.2 Compiler Process

The whole process of translating input C code to the AGP has been shown in figure 4.1. The C code initially goes through a pre-processing step where it checks the code only contains the data structures and control flow statements which are supported by AGP. Although this step has been shown in compiler process, the process of checking this would take great amount of language processing which is beyond the scope of this thesis. So we will assume that the input C code only contains supported structures in AGP (e.g. no pointers, structs, unions, etc.). The next step is scanning which is the process of tokenizing an input stream of characters, and analyzing whether all lexical rules have been respected. Tokens in a language are the basic components of that language such as keywords, separators, identifiers, etc. Tokenization is the process of taking an input text and transforming it into stream of tokens. Tokeinzing, however, is transforming each individual string to one token. So we tokenized those strings in the C codes through the process of tokenization. Then, by having the stream of tokens, the parser organizes them using grammar rules (syntax). Next is the semantic analyzer which primarily checks whether the code contains an errors such as function/variable re-declaration, variable assignments which has not been defined and so on. Finally, should it be no errors found in semantic analyzer step, the code will output an AGP code by going through the intermediate representation generator. This would be the major step of the compiler process which should be accurate on generating the exact same code as expected.

Figure 4.1: Compiler Process to generate AGP code given C input



4.3 C to AGP Translator

Throughout this section, we will introduce an algorithm for conversion of C to AGP in step-by-step manner. The translation algorithm would be much understandable if we explain how each line of C program is translated to corresponding AGP instruction. Hence, throughout the rest of this chapter, examples in C program and the corresponding AGP has been illustrated.

In Algorithm 1, a simple C function is shown. We have divided a simple C program into three various blocks. An if/else block which controls the flow of the program. A data parallel loop (while or for loop), and processing loop.

The idea is to eventually reach to a potential (recursive) function without any

Algorithm 1 Simple C Program

```
function SAMPLE(input)
    if (if condition) then                                  ▷ if/else block
    else
    end if

    for (init; condition; increment) do                     ▷ a possible data parallel loop
    end for

    while (while condition) do                              ▷ a possible processing loop
    end while
end function
```

arrays, pointers, etc which only has if/else statements that controls the flow of the program. To achieve this, we will follow the bellow steps:

First, we will evaluate the loops in a C program. As explained in section 4.1, we will decide whether the loop is a data-parallel or a processing loop.

A) If it is a processing loop, it has the potential to be rewritten as a recursive function. If that will be satisfied, then we will have a converted recursive function with at least one if/else statement, which is our ultimate goal. Otherwise, we will check whether we can unroll the loop (proceeding to step B).

B) On the other hand, if the loop proves to be a data parallel loop, a loop unrolling will help nullifying it and will result in parallel statements being executed in AGP.

4.4 Algorithm to Convert C to AGP

After knowing what AGP supports and understanding its the preliminary rules, we could go through the algorithm to convert C to AGP. We assume that we have a function with zero to multiple if/else statements, variable assignment, variable declaration, and no loops (not supported in our current implementation) which will

take at least one input and will output at least one value. Now, we can convert the C statements to equivalent AGP statement by following the rules mentioned below.

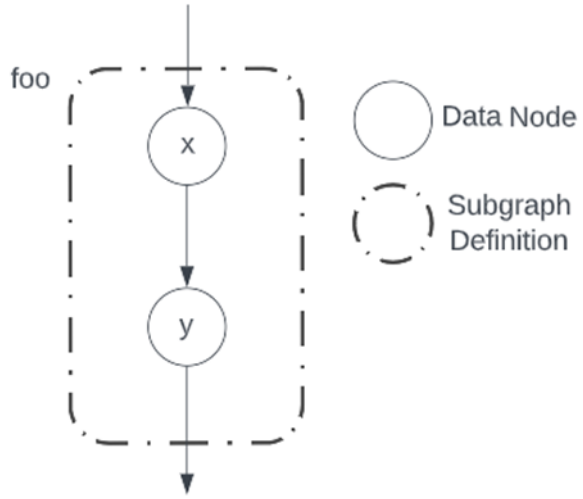
- **Function Declaration**

  1. Convert each function to subgraph by writing:

  SUBGRAPH(*name_of_the_subgraph*);

  Figure 4.2 shows how a function declaration looks like in AGP semantics.

  Figure 4.2: Function Declaration in AGP semantics



  2. For each function input, use a *DATUM* to define the parameter and an *INPUT* to make the parameter to be known as an input. If the C function takes an array of size n as its input, there should be n DATUM and INPUT statement defined for each element of the array.

  DATUM(*name_of_the_first_input*);

  INPUT(*name_of_the_first_input*);

3. AGP can have multiple but not zero *OUTPUTS*. Similar to function input, an *OUTPUT* is utilized to represent return statement in C language.

DATUM(*name_of_the_first_output*);

OUTPUT(*name_of_the_first_output*);

- **Variable Declaration**

4. C variable declaration will be converted to *DATUM* in AGP.

DATUM(*name_of_the_variable*);

- **Variable Initialization and Variable Assignment**

5. C variable initialization will be converted to either *CONST* or a series of OPERATOR instructions if there is a mathematical expression (not a single value is assigned to the variable).

CONST(*name_of_the_variable*, *value*);

6. The keyword OPERATOR is used for any operation in AGP. The syntax is similar to that of three-address machines in assembly language.

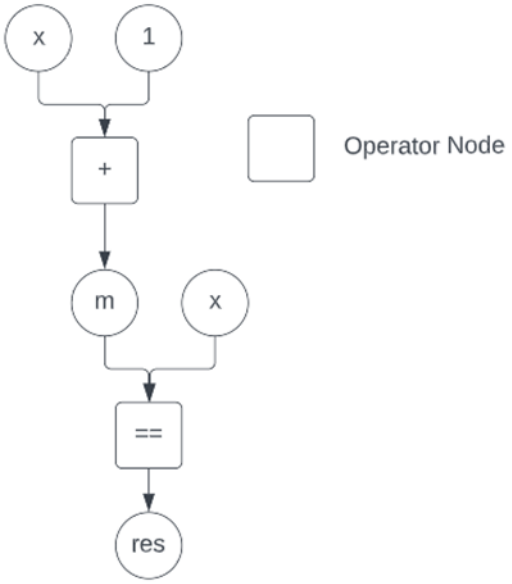
OPERATOR(*resultant*, op_*name*, *first_operand*, *second_operand*);

Figure 4.3 shows how relational and arithmetic operations look like in AGP semantics.

Arithmetic operators defined in AGP so far are addition, subtraction, and multiplication. The following statements are the AGP syntax for these three operators respectively:

- OPERATOR(*resultant*, op_*PLUS*, *first_operand*, *second_operand*);
- OPERATOR(*resultant*, op_*MINUS*, *first_operand*, *second_operand*);

Figure 4.3: Arithmetic and Relational operations in AGP semantics



- OPERATOR(*resultant*, op_*TIMES*, *first_operand*, *second_operand*);

Similarly, the existing relational operators are equality, greater than and less than. That would be sufficient to convert any relational operator in C to AGP. In the cases where there are operators such as greater equal ($\geq$) , or less equal ($\leq$), another instruction of *op_PLUS* or *op_MINUS* could be added to satisfy the expression assuming the only data type which is supported by AGP is integer.

- OPERATOR(*resultant*, op_*ISEQUAL*, *first_operand*, *second_operand*);
- OPERATOR(*resultant*, op_*ISGREATER*, *first_operand*, *second_operand*);
- OPERATOR(*resultant*, op_*ISLESS*, *first_operand*, *second_operand*);

Listing 4.7 has illustrated the equivalent AGP instruction for function declaration, variable declaration, variable initialization, function return statement, and variable assignment respectively.

- **If/Else Statement**

  The only conditional operators in AGP is if/else statement. Having said that, a switch case in C should initially be converted to nested C if/else statements. Then, it could be replaced with the following statements:

  - OPERATOR(*If_result*, op_*IF*, *if_condition_true*, *value*);
  - OPERATOR(*else_result*, op_*ELSE*, *if_condition_false*, *value*);

  if_condition_true or *if_condition_false* will be obtained from any relational expressions described above.

  Merge operation is used to unify the result of IF and ELSE statement together. Since there is no sequential operation in AGP, IF and ELSE statements are being processed concurrently.

  - OPERATOR(*final_if/else_result*, op_*MERGE*, *if_result*, *else_result*);

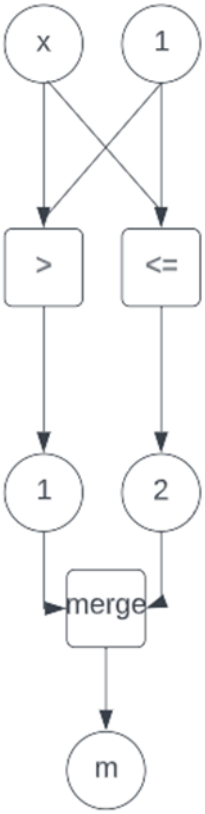
  Figure 4.4 shows how if/else statement looks like in AGP semantics.

  An example of if/else translation has been shown in listing 4.8. Line 11 calculates the relational statement and will put the result in $xis1$ node. Then in line 12 and 13, result_true and result_false node will be initialized based on the value of $xis1$. Finally, in the last line, the value of y will be determined by merging the result of if/else statement.

- **Function Calls**

  7. Function call will be achieved by EXPAND statement in AGP:

Figure 4.4: If/Else statement in AGP semantics



- EXPAND(*function's_name*,

  MAP_IN(callee_variable_name , caller_variable_name);

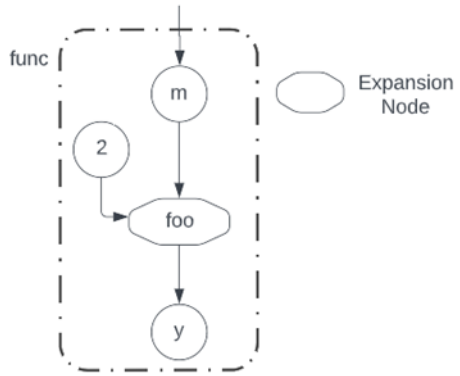
  MAP_OUT(callee_variable_name , caller_variable_name);));

For each parameter in callee subgraph, a MAP_IN() would be required. Similarly, multiple MAP_OUT() would be required for each of the output node in the callee subgraph. The name of the INPUT/OUTPUT variables in caller subgraph can be similar to their counterparts in callee subgraph.

Figure 4.5 shows how calling a subgraph looks like in AGP semantics.

An example of function calls translation has been shown in listing 4.9. We could

Figure 4.5: Subgraph Call in AGP semantics



see that in line 13, EXPAND operation has been used to demonstrate a function call. The input of EXPAND statement is foo which is the callee subgraph's name, followed by mapping of the inputs (i, the input node of foo subgraph, will be replaced by x), and mapping of the outputs (j, the output node of foo subgraph, will put its value in node j).

- **Recursive Function Calls**

  8. The recursive function call is slightly different from a usual function call with respect to the way *MAP_IN* and *MAP_OUT* are determined. The expansion instruction is as before. However, there should be a strategy to control the function call; Otherwise, it will be stuck in an infinite loop where a selected function will be called infinitely.

  While mapping the inputs, *caller_variable_name* (in the above EXPAND instruction) should be replaced with a new constructed node (*new_node*).

  The *new_node* will be constructed based on the if condition result and the reason for that is to update the input of a subgraph expansion call to avoid any infinite

loop in the code. The new instruction is as follows:

- OPERATOR(**new_node**, op_*IF*, *if_condition_result*, **new_value**);

  or

- OPERATOR(**new_node**, op_*ELSE*, *if_condition_result*, **new_value**);

The *new_value* in the above instruction is the result of a relational expression. This will be better explained when we provide an example of a recursive function such as factorial. We have shown that in listing 4.10.

For mapping the outputs, *callee_variable_name* (in the above EXPAND instruction) should be updated with *output* node. This means that the final result of a function call should be the exact output node of the caller function. For this to be realized, the return value of a function should be found and be used in EXPAND instruction.

- EXPAND(*function's_name*,

  MAP_IN(callee_variable_name , caller_variable_name);

  MAP_OUT(**OUTPUT_NODE** , caller_variable_name););

An example of a recursive function call (factorial) has been shown in listing 4.10. Fact function is being called with a new argument $(x - 1)$ which then would result in a new node in AGP (Line 21-22 in listing 4.10 (b)). Line 21 calculates $x - 1$ and will put it in *xminus1* node. However, this node will not be directly mapped into **MAP_IN** of the EXPANSION node; To control the number of times the fact function is being called, a node *xminus1cond* will be constructed which accepts the result of *xminus1* node only if x is greater than zero. Afterwards, *xminus1cond* node will be mapped into EXPAND instruction

(Line 19).

While mapping the function output nodes, it is important to find the return value of a function and then map the result into the output node. Therefore, node *result* in **MAP_OUT** function of (line 19 of listing 4.10 (b)), has been updated with the name of the output node of callee subgraph (line 20).

- **Return Statement**

  9. To illustrate the return statement in C, a TERMINATE statement in AGP has been conducted to exit the program. This is how the syntax in AGP works.

     - DATUM(*result*);
     - TERMINATE(*result*, *output_list*);

  The output list is the entire outputs in the main subgraph that should be terminated when they are processed and quantified.

```c
int foo (int x) {
    int y;  ▷ Variable Declaration
    int m = 1;          ▷ Variable Initialization
    y = x + 1;          ▷ Variable Assignment
    return y;       ▷ Return value
}
```

```
SUBGRAPH(foo)
    DATUM(x);
    INPUT(x);
    DATUM(y);
    OUTPUT(y);
    CONST(m,1);
    CONST(one,1);
    OPERATOR(y,op_PLUS,x,one);
```

Source Code 4.7: Function/variable declaration, variable assignment in C (a) and AGP (b)

```c
int foo (int x) {
    int y;
    if(x == 1)
        y = 1;
    else
        y = 2;
    return y;
}
```

```
SUBGRAPH(foo)
    DATUM(x);
    INPUT(x);
    DATUM(y);
    OUTPUT(y);
    CONST(one,1);
    CONST(two,2);
    DATUM(xis1);
    DATUM(result_true);
    DATUM(result_false);
    OPERATOR(xis1,op_ISEQUAL,x,one);
    OPERATOR(result_true,op_IF,xis1,one);
    OPERATOR(result_false,op_ELSE,xis1,two);
    OPERATOR(y,op_MERGE,result_true,result_false);
```

Source Code 4.8: If-else statements in C (a) and AGP (b)

```c
int foo (int i){
    int j;
    j = i + 1;
    return j;
}
int func (int x) {
     int y;
     y = foo(x);
     return y;
}
```

```
SUBGRAPH(foo)
    DATUM(i);
    INPUT(i);
    DATUM(j);
    OUTPUT(j);
    CONST(one,1);
    OPERATOR(j,op_PLUS,i,one);
SUBGRAPH(func)
     DATUM(x);
     INPUT(x);
     DATUM(y);
     OUTPUT(y);
     EXPAND(foo, MAP_IN(i,x); MAP_OUT(j,y););
```

Source Code 4.9: function/subgraph call in C (a) and AGP (b)

```c
int fact (int x){
    if(x > 0){
        return
            x*fact(x-1);
    }else{
        return 1;
    }
}
```

```
SUBGRAPH(fact)
    DATUM(x);
    INPUT(x);
    DATUM(result);
    OUTPUT(result);
    DATUM(resulttrue);
    DATUM(resultfalse);
    DATUM(xisg0);
    DATUM(iter);
    DATUM(nextiter);
    DATUM(xminus1);
    DATUM(xminus1cond);
    CONST(zero,0);
    CONST(one,1);
    OPERATOR(xisg0,op_ISGREATER,x,zero);
    OPERATOR(resulttrue,op_IF,xisg0,iter);
    OPERATOR(resultfalse,op_ELSE,xisg0,one);
    OPERATOR(iter,op_TIMES,x,nextiter);
    EXPAND(fact,MAP_IN(x,xminus1cond);
           MAP_OUT(result,nextiter););
    OPERATOR(result,op_MERGE,resulttrue,resultfalse);
    OPERATOR(xminus1,op_MINUS,x,one);
    OPERATOR(xminus1cond,op_IF,xisg0,xminus1);
```

Source Code 4.10: recursive function/subgraph call in C (a) and AGP (b)

Chapter 5

Results

In this chapter, the results are described and discussed.

To explore the results, we are interested to see, given a sample C as an input, what would be the output of a compiler. Therefore, for the rest of this chapter we have provided the C code snippets that have been given to our compiler and have shown the generated AGP code for the given input. First, we will show the actual AGP code for the C examples shown in chapter 4. Then, some micro benchmarks are being tested to explore whether the AGP output generated from our compiler is similar to what was expected. Lastly, we introduce three main benchmarks that are given to AGP compiler as well as chc compiler for further timing analysis purpose.

Listing A.1 is the generated AGP for the example shown in listing 4.7 (function/-variable declaration, variable assignment). This was the same AGP code that was expected, with the name of the nodes being the only difference. Our compiler uses a generate_label function that creates arbitrary names for nodes. This is useful to avoid any multiple assignments in C language as well. Because, AGP is creating a node with exclusive name that can be distinguished in the rest of the code. Also, the order

of the generated code is various from what has been seen in listing 4.7 (b); however, this does not interfere with the execution of AGP since AGP uses an address of each nodes to send and receive the data.

It is important to note that *TERMINATE* statement has not been generated and the reason is that *TERMINATE* is used when we want to link all outputs we wish to see (e.g. the outputs in the main subgraph) and we do not want the program end prematurely. However, since there is no main subgraph in the examples given to our compiler, the generation of *TERMINATE* statement has been ignored.

Listing A.2 is the output AGP code for the if/else example seen in listing 4.8 which is similar to what we expected except for node names and the order of instructions. For example, *node_1* ($xis1$ in listing 4.8) is the result of equal statement. There are two constants for number one in the generated AGP code which could be reduced to only one *CONST* instruction. However, AGP does not keep track of the nodes that have been already exist in the code.

With the differences mentioned above, listing A.3 is the output AGP code for the function call example seen in listing 4.9.

5.1 C Micro-Benchmarks

Two micro-benchmarks have been studied which are bigger in size compared to previous C snippets. The first micro-benchmark is Foo function shown in listing A.4; It is one of the benchmarks that was given to the compiler which includes almost all the possible statements in C (variable declaration/assignment, conditional statement (if/else) and function call within a variable assignment). Listing A.5 which is the generated AGP code of listing A.4, has partitioned to sections in the order that has appeared in C

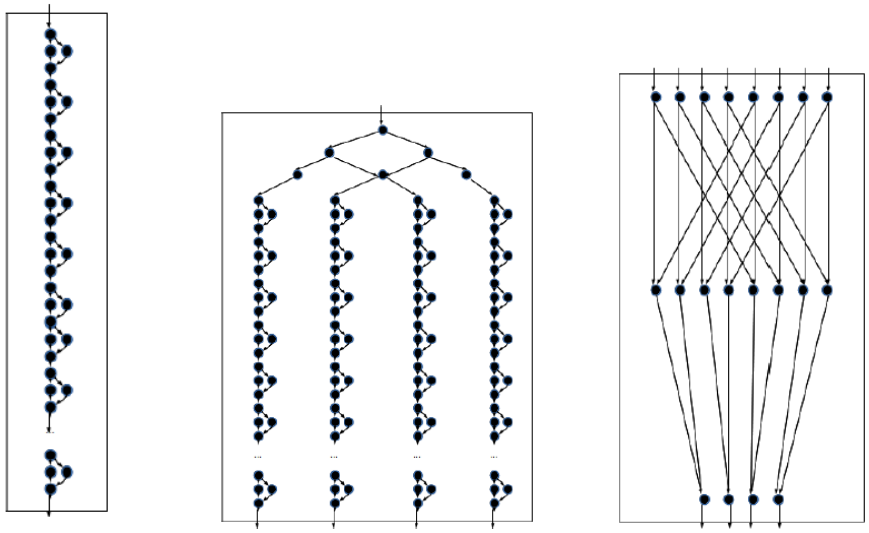
example. Hence, the function definition, variable declaration and variable assignment section of AGP have been mapped to that of line 1-10 of listing A.4. Subsequently, if/else statement section is the result of line 11-18.

As illustrated in the listing A.5, there are two pairs of *op_IF* and *op_ELSE* and the reason for that is that AGP is using a merge statement to converge the result of one pair of if/else statement; Since there are two variable assignments inside if and else statement, AGP needs two pairs of if/else operations and *op_MERGE* respectively. Almost at the bottom of if/else statement section of AGP-generated code, the if condition statement of C has been translated and the result of which is put on *node_8* which has been used as the third input of *op_IF* and *op_ELSE* operator. Line 20 in C example shows a variable assignment which holds the result of a function call and the corresponding AGP statement for that is *EXPAND*. Eventually, the return statement (line 21 of C) that is corresponded to OUTPUT statement in AGP.

The Factorial function in listing 4.10 is the second micro-benchmark which will show how a recursive function call in C would be translated into AGP. Our compiler initially generates the innermost statements, meaning that the statements in if/else statements, for example, are being produced first. Listing A.6, is the generated output code for factorial. *node_1* is the result of if condition which is used to control three other nodes. These nodes are as follows: a) *node_8* which perform the recursive function call, b) *node_9* which holds the base condition value and c) *node_12* which holds the new function argument while calling a factorial function recursively. The output of the Factorial is put on *node_10* (Merge of if condition being true or false). The factorial function is recursively being called using EXPAND instruction and *node_12* is mapped to x and the final value of expansion call (*nextiter*) will be put on *node_10* as the output node of the function.

5.2 C Benchmarks

We have selected a set of C benchmarks to give them to our AGP generator compiler. Then we have given the output of AGP compiler to *chc* compiler and have performed a time measurement on them. The selection of C benchmarks have been conducted in a way that include a range of C programs from sequential in essence, to those that have the potential to be parallelizable and so on. We have investigated whether our compiler could generate appropriate AGP code for these benchmarks (cascade, double fact (linear), and matrix multiplication 2x2, 4x4 and 8x8) and we compared the generated AGP code from our compiler with the expected code.



(a) Linear (b) Cascade (c) Matrix Multiplication 2x2

Figure 5.1: Benchmarks' execution profile. These plots do not represent canonical program implementations: rather, they depict full graph expansions throughout program runtime. Input nodes are depicted on the top, and output nodes at the bottom.

5.2.1 Double Factorial (Linear)

One of the benchmarks we have tested is double factorial, which offers very few parallelization opportunities. However, we include this example to our benchmarks to illustrate how a program which has a direct sequential dependency between data will be executed on the chc compiler. Our compiler will take the C code for double fact (shown in listing A.7) and will output equivalent AGP code shown in listing A.8. The benchmark's execution profile for double factorial (linear) is depicted in figure 5.1a.

5.2.2 Matrix Multiplication

As explained in section 4.1, Matrix Multiplication is a great example for a data parallel loop and it consists of a great number of parallel computations which we have included them in our benchmarks. However, our AGP compiler does not support loops at the moment; As a result, we unrolled the loop and generated multiple lines of variable assignment and used that as the input file of matrix multiplication C code. The input C file given to our compiler for matrix multiplication 2x2 has been shown in listing A.9. Listing A.10, is the generated output code for matrix multiplication 2x2. Two other benchmarks (matrix multiplication 4x4 and 8x8) have also been evaluated and the result of which have been brought in Appendix A.

Benchmark's execution profile for matrix multiplication depicted in figure 5.1c (matrix multiplication for sizes 4x4 and 8x8 are ommitted, for brevity). The execution profile depicts the lifetime of the complete program, and illustrates runtime data dependencies for matrix multiplication 2x2.

5.2.3 Cascade

The last benchmark we have chosen is cascade. It presents four identical, and independent, computational streams as shown in figure 5.1b . The input C file given to our compiler for matrix multiplication 2x2 has been shown in listing A.13. As a result, Listing A.14, is the generated output code for cascade shown in AGP programming model.

5.3 Time and Analysis

In table 5.1, the compilation time for each of the benchmarks has been shown. Our compiler goes through the steps shown in figure 4.1 to generate AGP. The compilation time have been measured 10 times and then the average time has been calculated for each benchmark. mm_8x8 is taking the most time to be compiled with 12.101 milliseconds. This is due to the fact that mm_8x8 contains abundant statements in C (long variable assignment with long arithmetic operators). Cascade compilation time takes 1.198 (ms) on average and is the third largest in regard to compilation time and that is mainly because in the cascade example, there are multiple function calls which should be assigned to a variable. This takes a considerable amount of time for both the time it takes the compiler to generate multiple expansion calls and then refactoring the inputs and outputs of the expansion accordingly. We can conclude that the longer the C program and the more refactoring needed, the longer it takes to be compiled by our compiler.

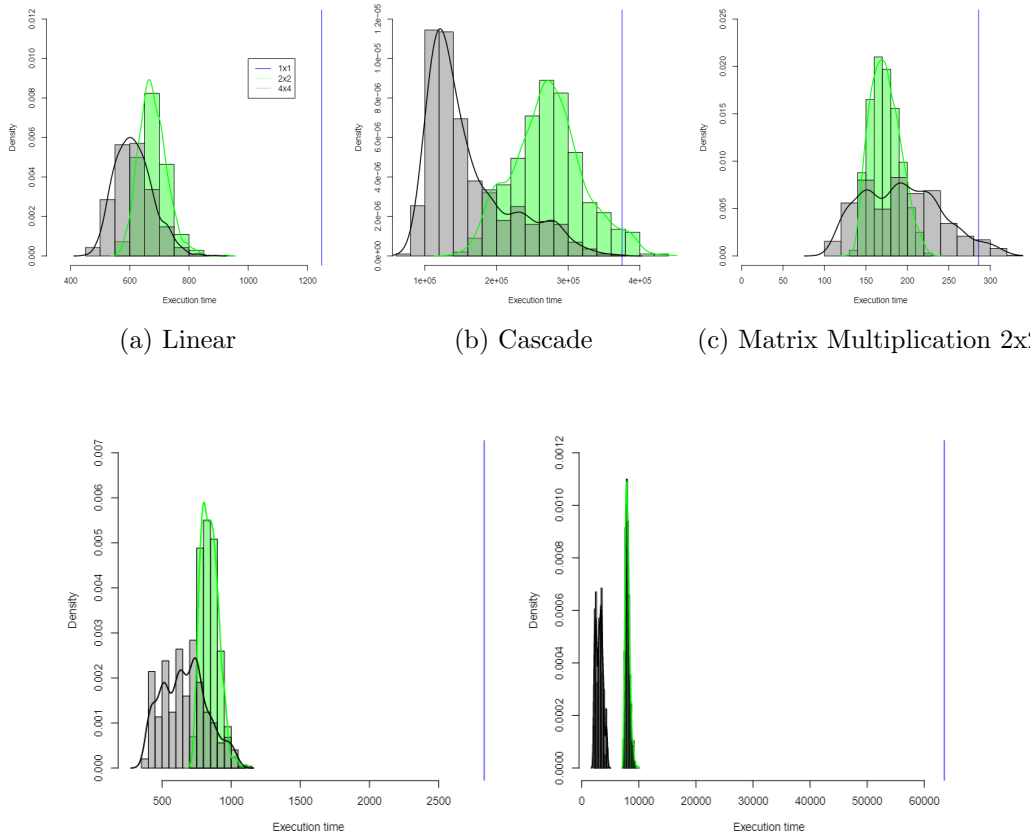
There isn't a specific way or formula of saying that a parallelization strategy or a model of computation is better than the others. Thus, the approach of measuring the effectiveness of parallelization, is to take a suite of benchmarks and parallelize

Benchmark	itr_1	itr_2	itr_3	itr_4	itr_5	itr_6	itr_7	itr_8	itr_9	itr_10	Average
listing 13	0.05	0.02	0.08	0.30	0.07	0.03	0.06	0.03	0.08	0.07	0.08
listing 14	0.04	0.10	0.03	0.06	0.06	0.15	0.02	0.03	0.10	0.10	0.07
listing 15	0.05	0.22	0.03	0.08	0.18	0.06	0.05	0.05	0.09	0.02	0.08
foo	0.15	0.12	0.09	0.18	0.10	0.15	0.51	0.26	0.30	0.13	0.20
factorial	0.12	0.07	0.29	0.16	0.15	0.07	0.04	0.07	0.13	0.10	0.12
linear	0.50	0.51	0.23	0.46	0.29	0.49	0.20	0.20	0.57	0.40	0.38
cascade	0.68	0.60	4.37	0.46	0.48	1.10	1.27	1.66	0.80	0.53	1.19
mm_2x2	0.06	0.12	0.26	0.12	0.23	0.09	0.12	0.16	0.12	0.37	0.16
mm_4x4	1.76	0.81	1.97	0.81	1.78	1.81	0.81	1.10	1.99	2.24	1.51
mm_8x8	13.06	11.82	13.49	11.58	13.48	12.44	11.01	11.18	11.41	11.49	12.10

Table 5.1: Compile time in (ms) for each benchmark

all of them, and then compare their execution times between different parallelizing compilers. The major benchmarks described previously (linear, cascade and matrix multiplication) have been given as the input file to the chc compiler. Each of the benchmarks were ran 1000 times across cores, for 1, 2x2, 4x4 and 8x8 multi-core configurations. For every run, we observed that programs terminated and returned correct results which also supports that the second compiler (chc compiler) works properly. It is important to mention that the single-core execution is used to provide a baseline for comparison. Figure 5.2 depicts the execution time of all configurations of each benchmark. Blue represents the single-core execution, green represents the two-by-two (4-core) configuration, and gray shows the four-by-four (16-core) configuration. The single-core execution of every benchmark is a vertical line meaning that the core execution is deterministic. It can be seen that in the linear case, the improvement from a single-core to a multi-core configuration is considerable. However, there is not much difference between 2x2 and 4x4 configuration as double fact (linear) provides little opportunity for parallel execution. By increasing the number of cores in Cascade benchmark, we see some improvements which is considerable. Matrix Multiplication 4x4 and 8x8 similarly benefit from the increase in the number of cores. Only in the

matrix multiplication 2x2, we see that there is not much difference between 2x2 and 4x4 configurations which might due to the communication overhead between cores.



(a) Linear (b) Cascade (c) Matrix Multiplication 2x2

(d) Matrix Multiplication 4x4 (e) Matrix Multiplication 8x8

Figure 5.2: Execution time histograms for each benchmark. Single core execution time is marked by the vertical blue line. Green bars represent runs in the 2x2 configuration; Gray bars represent runs in the 4x4 configuration.

Although the functionality of the chc compiler is outside the scope of our research, the correct answer out of chc compiler proves two things: A) our C to AGP compiler has generated a correct AGP code, and B) the validity of a claim that the generated AGP code will produce a correct answer as we expected and also the ultimate goal of Asynchronous Graph Programming model that is AGP programs can be arbitrarily

parallelized.

Chapter 6

Conclusion and Future Work

The thesis presented the algorithm to translate an imperative language ,C, to an intermediate representation, AGP, which is a programming model. What will be produced from our compiler will then be passed to chc compiler described in chapter 2. The chc compiler is used in a multi-core or distributed systems whose aim is to parallelize the code by taking AGP as an input and turning it into a stack-based structure. However, the focus on this thesis lies on one step earlier, where AGP code should be produced.

Like other existing parallelizing compilers that convert a source code to a destination code (e.g. sequential code to CUDA/OpenCL), our compiler also aim to convert C code to a new parallel programming model described in (AGP) [16]. Since some of these target programming models are not known worldwide and take time to be learnt and utilized, introducing a compiler to translate a well-known and general-purpose language to such a programming language is of great importance.

In this work, we have briefly introduced the AGP as our target programming model, the algorithm to convert C to AGP and then we showed the concordance

between the results generated from compiler and what was expected in AGP. As part of AGP project, I was responsible for half of that which was designing and developing a C-to-AGP compiler. For evaluation purposes, we have used the other half of project in which an AGP to many-core compiler had been developed; What we examined in this thesis was to evaluate how well the generated AGP programs out from our compiler will perform under parallelization using this compiler.

We have explained how we measure parallelization. However, identifying parallelization is outside the scope of our research. Because identifying parallelization is related to node allocation problems (i.e. where to correctly allocate each node in an AGP program) and there is plenty of work to be done on this topic which we did not focus on them in our research.

Our compiler has been shown to support most of the substantial features in C. These features include, but are not limited to, variable/function declaration, variable assignment, mathematical expressions, and conditional statements. AGP does not handle a loop structure (for loop and while loop) and it uses other ways such as recursive subgraph calls and if else statement to cover them. Although not all the loops could be implemented using a recursive function call (e.g. data parallel loops), most of them have the potential to be rewritten in a recursive format and therefore be translatable to AGP. Currently, our compiler does not have support for data parallel loops. In future works, the support for how to unroll the loops and convert the statement to AGP could be added to the compiler.

At the moment, the proposed compiler will create nodes based on the statements written in C, and therefore some nodes might be created multiple times, which might not be efficient. In future, we could add a post-processing step into our compiler to

evaluate how the final code could become more efficient with minimum number of nodes.

Currently, our compiler does not support loops and we have to recognize what type of loop would that be (e.g. data parallel loop or processing loop) and unroll the loop or convert it to a recursive function accordingly. In future, we can add the support for loops and also introduce an algorithm to identify the type of the loops and convert the c input to a simpler version so that compiler could generate AGP output of the simplified version of C code. Introducing such algorithm would take a lot of effort and could be another topic of a master thesis and therefore we will not cover that in our research.

Appendix A

Appendices

A.1 AGP codes, Expected vs Algorithm

```
SUBGRAPH(foo)
    DATUM(x);
    INPUT(x);
    DATUM(y);
    OUTPUT(y);
    CONST(m,1);
    CONST(one,1);
    OPERATOR(y,op_PLUS,x,one);
```

```
SUBGRAPH(foo)
    DATUM(x);
    INPUT(x);
    DATUM(y);
    CONST(node_0,1);
    OPERATOR(y,op_PLUS,x,node_2);
    CONST(node_2,1);
    OUTPUT(y);
```

Source Code A.1: AGP code for Listing 4.7 (a) Expected (b) Compiler Output

```
SUBGRAPH(foo)
 DATUM(x);
 INPUT(x);
 DATUM(y);
 OUTPUT(y);
 CONST(one,1);
 CONST(two,2);
 DATUM(xis1);
 DATUM(result_true);
 DATUM(result_false);
 OPERATOR(xis1,op_ISEQUAL,x,one);
 OPERATOR(result_true,op_IF,xis1,one);
 OPERATOR(result_false,op_ELSE,xis1,two);
 OPERATOR(y,op_MERGE,
        result_true,result_false);
```

```
SUBGRAPH(foo)
  DATUM(x);
  INPUT(x);
  DATUM(y);
  OPERATOR(node_4,op_IF,node_1,node_2);
  OPERATOR(node_5,op_ELSE,node_1,node_3);
  OPERATOR(y,op_MERGE,node_4,node_5);
  DATUM(node_4);
  DATUM(node_5);
  CONST(node_2,1);
  CONST(noce_3,2);
  OPERATOR(node_1,op_ISEQUAL,x,node_0);
  CONST(node_0,1);
  DATUM(node_1);
  OUTPUT(y);
```

Source Code A.2: AGP code for Listing 4.8 (a) Expected (b) Compiler Output

```
SUBGRAPH(foo)
    DATUM(i);
    INPUT(i);
    DATUM(j);
    OUTPUT(j);
    CONST(one,1);
    OPERATOR(j,op_PLUS,i,one);
SUBGRAPH(func)
     DATUM(x);
     INPUT(x);
     DATUM(y);
     OUTPUT(y);
     EXPAND(foo, MAP_IN(i,x);
         MAP_OUT(j,y););
```

```
SUBGRAPH(foo)
    DATUM(i);
    INPUT(i);
    DATUM(j);
    OPERATOR(node_2,op_PLUS,i,node_1);
    CONST(node_1,1);
    DATUM(node_2);
    OUTPUT(j);
SUBGRAPH(func)
     DATUM(x);
     INPUT(x);
     DATUM(y);
     EXPAND(foo,
         MAP_IN(i,x);
         MAP_OUT(j,y););
     OUTPUT(y);
```

Source Code A.3: AGP code for Listing 4.9 (a) Expected (b) Compiler Output

```
int foo ( int k, int f) {

 int y;
 int m;
 int t;
 int x;

 y = 4;
 y = (t-1) + 2;
```

```
10
11 if( y == (t−1)){
12  m = 2;
13  x = 3;
14 } else {
15  m = 9;
16  x = 7;
17  int l;
18 }
19
20 t = foo(m, x);
21 return t;
22
23 }
```

Source Code A.4: C example covering variable assignment, if/else statement, function call

```
1  SUBGRAPH(foo)                          ▷ // Function Declaration line [1-5]
2  DATUM(k);
3  INPUT(k);
4  DATUM(f);
5  INPUT(f);
6  DATUM(y);                              ▷ // Variable Declaration line [6-9]
7  DATUM(m);
8  DATUM(t);
9  DATUM(x);
10 CONST(node_0, 4);                      ▷ // Variable Assignment line [10-16]
```

```
11 OPERATOR(node_5, op_PLUS, node_3, node_4);
12 OPERATOR(node_3, op_MINUS, t, node_2);
13 CONST(node_2, 1);
14 DATUM(node_3);
15 CONST(node_4, 2);
16 DATUM(node_5);
17 OPERATOR(node_13, op_IF, node_8, node_9);        ▷ // If/Else Statement line
       [17-36]
18 OPERATOR(node_14, op_ELSE, node_8, node_11);
19 OPERATOR(m, op_MERGE, node_13, node_14);
20 DATUM(node_13);
21 DATUM(node_14);
22 OPERATOR(node_16, op_IF, node_8, node_10);
23 OPERATOR(node_17, op_ELSE, node_8, node_12);
24 OPERATOR(x, op_MERGE, node_16, node_17);
25 DATUM(node_16);
26 DATUM(node_17);
27 CONST(node_9, 2);
28 CONST(node_10, 3);
29 CONST(node_11, 9);
30 CONST(node_12, 7);
31 DATUM(l);
32 OPERATOR(node_8, op_ISEQUAL, y, node_7);
33 OPERATOR(node_7, op_MINUS, t, node_6);
34 CONST(node_6, 1);
35 DATUM(node_7);
36 DATUM(node_8);
```

```
37 EXPAND(foo, MAP_IN(k,m);MAP_IN(f,x);MAP_OUT(t,t););        ▷ // Function Call
38 OUTPUT(t);                                                  ▷ //line [37-38]
```

Source Code A.5: Generated AGP code for listing A.4 (Foo Function)

```
1  SUBGRAPH(fact)
2  DATUM(x);
3  INPUT(x);
4  OPERATOR(node_8, op_IF, node_1, node_4);
5  OPERATOR(node_9, op_ELSE, node_1, node_6);
6  OPERATOR(node_10, op_MERGE, node_8, node_9);
7  OUTPUT(node_10);
8  DATUM(node_8);
9  DATUM(node_9);
10 DATUM(node_10);
11 OPERATOR(node_12, op_IF, node_1, node_3);
12 DATUM(node_12);
13 OPERATOR(node_4, op_TIMES, x, nextiter);
14 EXPAND(fact, MAP_IN(x,node_12);MAP_OUT(node_10,nextiter););
15 OPERATOR(node_3, op_MINUS, x, node_2);
16 CONST(node_2, 1);
17 DATUM(node_3);
18 DATUM(node_4);
19 CONST(node_6, 1);
20 OPERATOR(node_1, op_ISGREATER, x, node_0);
21 CONST(node_0, 0);
22 DATUM(node_1);
```

Source Code A.6: Generated AGP code for listing 4.10 (Factorial)

A.2 Benchmarks' C to AGP translation

```
int plus1(int x){
  int result;
  result = x+1;
  return result;
}

int fact (int x){
  if(x > 0){
    return x*fact(x-1);
  }else{
    return 1;
  }
}

int test(int x, int y){

  int z;

  if(x == y){
    z = fact(x);
  }else{
```

```
    z = 0;
  }

  return z;
}

int main(){

  int x = 7;
  int y = 7;
  int z[2];

  int x2;
  int y2;

  x2 = x + 1;
  y2 = y + 1;

  z[0] = test(x,y);
  z[1] = test(x2,y2);

  return z;

}
```

Source Code A.7: C example for double factorial

```
SUBGRAPH(plus1)
DATUM(x);
INPUT(x);
DATUM(result);
OPERATOR(node_2, op_PLUS, x, node_1);
CONST(node_1, 1);
DATUM(node_2);
OUTPUT(node_2);

SUBGRAPH(fact)
DATUM(x);
INPUT(x);
OPERATOR(node_12, op_IF, node_5, node_8);
OPERATOR(node_13, op_ELSE, node_5, node_10);
OPERATOR(node_14, op_MERGE, node_12, node_13);
OUTPUT(node_14);
DATUM(node_12);
DATUM(node_13);
DATUM(node_14);
OPERATOR(node_16, op_IF, node_5, node_7);
DATUM(node_16);
OPERATOR(node_8, op_TIMES, x, nextiter);
EXPAND(fact, MAP_IN(x,node_16);MAP_OUT(node_14,nextiter););
OPERATOR(node_7, op_MINUS, x, node_6);
CONST(node_6, 1);
DATUM(node_7);
```

```
27 DATUM(node_8);
28 CONST(node_10, 1);
29 OPERATOR(node_5, op_ISGREATER, x, node_4);
30 CONST(node_4, 0);
31 DATUM(node_5);
32
33 SUBGRAPH(test)
34 DATUM(x);
35 INPUT(x);
36 DATUM(y);
37 INPUT(y);
38 DATUM(z);
39 OPERATOR(node_21, op_IF, node_17, node_19);
40 OPERATOR(node_22, op_ELSE, node_17, node_20);
41 OPERATOR(z, op_MERGE, node_21, node_22);
42 DATUM(node_21);
43 DATUM(node_22);
44 EXPAND(fact, MAP_IN(x,x);MAP_OUT(z,node_19););
45 DATUM(node_19);
46 CONST(node_20, 0);
47 OPERATOR(node_17, op_ISEQUAL, x, y);
48 DATUM(node_17);
49 OUTPUT(z);
50
51 SUBGRAPH(main)
52 DATUM();
```

```
CONST(node_25, 7);
CONST(node_26, 7);
DATUM(z_0);
DATUM(z_1);
DATUM(x2);
DATUM(y2);
OPERATOR(node_29, op_PLUS, x, node_28);
CONST(node_28, 1);
DATUM(node_29);
OPERATOR(node_32, op_PLUS, y, node_31);
CONST(node_31, 1);
DATUM(node_32);
EXPAND(node_33, MAP_IN(x,x);MAP_IN(y,y);MAP_OUT(z_0,node_34););
DATUM(node_34);
EXPAND(node_35, MAP_IN(x,node_29);MAP_IN(y,node_32);MAP_OUT(z_1,node_36););
DATUM(node_36);
OUTPUT(z_0);
OUTPUT(z_1);
```

Source Code A.8: Generated AGP code for listing A.7 (double factorial)

Matrix Multiplication

```
int foo (int a [2][2], int b [2][2]) {

 int c [2][2];

```

```
c [0][0] = ( a [0][0] * b [0][0] ) + ( a [0][1] * b [1][0] );
c [0][1] = ( a [0][0] * b [0][1] ) + ( a [0][1] * b [1][1] );
c [1][0] = ( a [1][0] * b [0][0] ) + ( a [1][1] * b [1][0] );
c [1][1] = ( a [1][0] * b [0][1] ) + ( a [1][1] * b [1][1] );

return c;

}
```

Source Code A.9: C example for matrix multiplication 2x2

```
SUBGRAPH(foo)
DATUM(a_00);
INPUT(a_00);
DATUM(a_01);
INPUT(a_01);
DATUM(a_10);
INPUT(a_10);
DATUM(a_11);
INPUT(a_11);
DATUM(b_00);
INPUT(b_00);
DATUM(b_01);
INPUT(b_01);
DATUM(b_10);
INPUT(b_10);
DATUM(b_11);
```

```
17 INPUT(b_11);
18 DATUM(c_00);
19 DATUM(c_01);
20 DATUM(c_10);
21 DATUM(c_11);
22 OPERATOR(c_00, op_PLUS, node_0, node_1);
23 OPERATOR(node_0, op_TIMES, a_00, b_00);
24 DATUM(node_0);
25 OPERATOR(node_1, op_TIMES, a_01, b_10);
26 DATUM(node_1);
27 OPERATOR(c_01, op_PLUS, node_3, node_4);
28 OPERATOR(node_3, op_TIMES, a_00, b_01);
29 DATUM(node_3);
30 OPERATOR(node_4, op_TIMES, a_01, b_11);
31 DATUM(node_4);
32 OPERATOR(c_10, op_PLUS, node_6, node_7);
33 OPERATOR(node_6, op_TIMES, a_10, b_00);
34 DATUM(node_6);
35 OPERATOR(node_7, op_TIMES, a_11, b_10);
36 DATUM(node_7);
37 OPERATOR(c_11, op_PLUS, node_9, node_10);
38 OPERATOR(node_9, op_TIMES, a_10, b_01);
39 DATUM(node_9);
40 OPERATOR(node_10, op_TIMES, a_11, b_11);
41 DATUM(node_10);
42 OUTPUT(c_00);
```

```
OUTPUT(c_01);
OUTPUT(c_10);
OUTPUT(c_11);
```

Source Code A.10: Generated AGP code for listing A.9 (matrix multiplication 2x2)

```
int foo (int a [4][4], int b [4][4]) {

 int c [4][4];


c [0][0] = ( a [0][0] * b [0][0] ) + ( a [0][1] * b [1][0] ) + ( a [0][2] * b [2][0] ) + ( a
    [0][3] * b [3][0] );
c [0][1] = ( a [0][0] * b [0][1] ) + ( a [0][1] * b [1][1] ) + ( a [0][2] * b [2][1] ) + ( a
    [0][3] * b [3][1] );
c [0][2] = ( a [0][0] * b [0][2] ) + ( a [0][1] * b [1][2] ) + ( a [0][2] * b [2][2] ) + ( a
    [0][3] * b [3][2] );
c [0][3] = ( a [0][0] * b [0][3] ) + ( a [0][1] * b [1][3] ) + ( a [0][2] * b [2][3] ) + ( a
    [0][3] * b [3][3] );
c [1][0] = ( a [1][0] * b [0][0] ) + ( a [1][1] * b [1][0] ) + ( a [1][2] * b [2][0] ) + ( a
    [1][3] * b [3][0] );
c [1][1] = ( a [1][0] * b [0][1] ) + ( a [1][1] * b [1][1] ) + ( a [1][2] * b [2][1] ) + ( a
    [1][3] * b [3][1] );
c [1][2] = ( a [1][0] * b [0][2] ) + ( a [1][1] * b [1][2] ) + ( a [1][2] * b [2][2] ) + ( a
    [1][3] * b [3][2] );
c [1][3] = ( a [1][0] * b [0][3] ) + ( a [1][1] * b [1][3] ) + ( a [1][2] * b [2][3] ) + ( a
    [1][3] * b [3][3] );
```

```
c [2][0] = ( a [2][0] * b [0][0] ) + ( a [2][1] * b [1][0] ) + ( a [2][2] * b [2][0] ) + ( a
    [2][3] * b [3][0] );
c [2][1] = ( a [2][0] * b [0][1] ) + ( a [2][1] * b [1][1] ) + ( a [2][2] * b [2][1] ) + ( a
    [2][3] * b [3][1] );
c [2][2] = ( a [2][0] * b [0][2] ) + ( a [2][1] * b [1][2] ) + ( a [2][2] * b [2][2] ) + ( a
    [2][3] * b [3][2] );
c [2][3] = ( a [2][0] * b [0][3] ) + ( a [2][1] * b [1][3] ) + ( a [2][2] * b [2][3] ) + ( a
    [2][3] * b [3][3] );
c [3][0] = ( a [3][0] * b [0][0] ) + ( a [3][1] * b [1][0] ) + ( a [3][2] * b [2][0] ) + ( a
    [3][3] * b [3][0] );
c [3][1] = ( a [3][0] * b [0][1] ) + ( a [3][1] * b [1][1] ) + ( a [3][2] * b [2][1] ) + ( a
    [3][3] * b [3][1] );
c [3][2] = ( a [3][0] * b [0][2] ) + ( a [3][1] * b [1][2] ) + ( a [3][2] * b [2][2] ) + ( a
    [3][3] * b [3][2] );
c [3][3] = ( a [3][0] * b [0][3] ) + ( a [3][1] * b [1][3] ) + ( a [3][2] * b [2][3] ) + ( a
    [3][3] * b [3][3] );

return c;

}
```

Source Code A.11: C example for matrix multiplication 4x4

```
SUBGRAPH(foo)
DATUM(a_00);
INPUT(a_00);
DATUM(a_01);
```

```
INPUT(a_01);
DATUM(a_02);
INPUT(a_02);
DATUM(a_03);
INPUT(a_03);
DATUM(a_10);
INPUT(a_10);
DATUM(a_11);
INPUT(a_11);
DATUM(a_12);
INPUT(a_12);
DATUM(a_13);
INPUT(a_13);
DATUM(a_20);
INPUT(a_20);
DATUM(a_21);
INPUT(a_21);
DATUM(a_22);
INPUT(a_22);
DATUM(a_23);
INPUT(a_23);
DATUM(a_30);
INPUT(a_30);
DATUM(a_31);
INPUT(a_31);
DATUM(a_32);
```

```
INPUT(a_32);
DATUM(a_33);
INPUT(a_33);
DATUM(b_00);
INPUT(b_00);
DATUM(b_01);
INPUT(b_01);
DATUM(b_02);
INPUT(b_02);
DATUM(b_03);
INPUT(b_03);
DATUM(b_10);
INPUT(b_10);
DATUM(b_11);
INPUT(b_11);
DATUM(b_12);
INPUT(b_12);
DATUM(b_13);
INPUT(b_13);
DATUM(b_20);
INPUT(b_20);
DATUM(b_21);
INPUT(b_21);
DATUM(b_22);
INPUT(b_22);
DATUM(b_23);
```

```
57 INPUT(b_23);
58 DATUM(b_30);
59 INPUT(b_30);
60 DATUM(b_31);
61 INPUT(b_31);
62 DATUM(b_32);
63 INPUT(b_32);
64 DATUM(b_33);
65 INPUT(b_33);
66 DATUM(c_00);
67 DATUM(c_01);
68 DATUM(c_02);
69 DATUM(c_03);
70 DATUM(c_10);
71 DATUM(c_11);
72 DATUM(c_12);
73 DATUM(c_13);
74 DATUM(c_20);
75 DATUM(c_21);
76 DATUM(c_22);
77 DATUM(c_23);
78 DATUM(c_30);
79 DATUM(c_31);
80 DATUM(c_32);
81 DATUM(c_33);
82 OPERATOR(c_00, op_PLUS, node_0, node_5);
```

```
83 OPERATOR(node_0, op_TIMES, a_00, b_00);
84 DATUM(node_0);
85 OPERATOR(node_5, op_PLUS, node_1, node_4);
86 OPERATOR(node_1, op_TIMES, a_01, b_10);
87 DATUM(node_1);
88 OPERATOR(node_4, op_PLUS, node_2, node_3);
89 OPERATOR(node_2, op_TIMES, a_02, b_20);
90 DATUM(node_2);
91 OPERATOR(node_3, op_TIMES, a_03, b_30);
92 DATUM(node_3);
93 DATUM(node_4);
94 DATUM(node_5);
95 OPERATOR(c_01, op_PLUS, node_7, node_12);
96 OPERATOR(node_7, op_TIMES, a_00, b_01);
97 DATUM(node_7);
98 OPERATOR(node_12, op_PLUS, node_8, node_11);
99 OPERATOR(node_8, op_TIMES, a_01, b_11);
100 DATUM(node_8);
101 OPERATOR(node_11, op_PLUS, node_9, node_10);
102 OPERATOR(node_9, op_TIMES, a_02, b_21);
103 DATUM(node_9);
104 OPERATOR(node_10, op_TIMES, a_03, b_31);
105 DATUM(node_10);
106 DATUM(node_11);
107 DATUM(node_12);
108 OPERATOR(c_02, op_PLUS, node_14, node_19);
```

```
OPERATOR(node_14, op_TIMES, a_00, b_02);
DATUM(node_14);
OPERATOR(node_19, op_PLUS, node_15, node_18);
OPERATOR(node_15, op_TIMES, a_01, b_12);
DATUM(node_15);
OPERATOR(node_18, op_PLUS, node_16, node_17);
OPERATOR(node_16, op_TIMES, a_02, b_22);
DATUM(node_16);
OPERATOR(node_17, op_TIMES, a_03, b_32);
DATUM(node_17);
DATUM(node_18);
DATUM(node_19);
OPERATOR(c_03, op_PLUS, node_21, node_26);
OPERATOR(node_21, op_TIMES, a_00, b_03);
DATUM(node_21);
OPERATOR(node_26, op_PLUS, node_22, node_25);
OPERATOR(node_22, op_TIMES, a_01, b_13);
DATUM(node_22);
OPERATOR(node_25, op_PLUS, node_23, node_24);
OPERATOR(node_23, op_TIMES, a_02, b_23);
DATUM(node_23);
OPERATOR(node_24, op_TIMES, a_03, b_33);
DATUM(node_24);
DATUM(node_25);
DATUM(node_26);
OPERATOR(c_10, op_PLUS, node_28, node_33);
```

```
OPERATOR(node_28, op_TIMES, a_10, b_00);
DATUM(node_28);
OPERATOR(node_33, op_PLUS, node_29, node_32);
OPERATOR(node_29, op_TIMES, a_11, b_10);
DATUM(node_29);
OPERATOR(node_32, op_PLUS, node_30, node_31);
OPERATOR(node_30, op_TIMES, a_12, b_20);
DATUM(node_30);
OPERATOR(node_31, op_TIMES, a_13, b_30);
DATUM(node_31);
DATUM(node_32);
DATUM(node_33);
OPERATOR(c_11, op_PLUS, node_35, node_40);
OPERATOR(node_35, op_TIMES, a_10, b_01);
DATUM(node_35);
OPERATOR(node_40, op_PLUS, node_36, node_39);
OPERATOR(node_36, op_TIMES, a_11, b_11);
DATUM(node_36);
OPERATOR(node_39, op_PLUS, node_37, node_38);
OPERATOR(node_37, op_TIMES, a_12, b_21);
DATUM(node_37);
OPERATOR(node_38, op_TIMES, a_13, b_31);
DATUM(node_38);
DATUM(node_39);
DATUM(node_40);
OPERATOR(c_12, op_PLUS, node_42, node_47);
```

```
OPERATOR(node_42, op_TIMES, a_10, b_02);
DATUM(node_42);
OPERATOR(node_47, op_PLUS, node_43, node_46);
OPERATOR(node_43, op_TIMES, a_11, b_12);
DATUM(node_43);
OPERATOR(node_46, op_PLUS, node_44, node_45);
OPERATOR(node_44, op_TIMES, a_12, b_22);
DATUM(node_44);
OPERATOR(node_45, op_TIMES, a_13, b_32);
DATUM(node_45);
DATUM(node_46);
DATUM(node_47);
OPERATOR(c_13, op_PLUS, node_49, node_54);
OPERATOR(node_49, op_TIMES, a_10, b_03);
DATUM(node_49);
OPERATOR(node_54, op_PLUS, node_50, node_53);
OPERATOR(node_50, op_TIMES, a_11, b_13);
DATUM(node_50);
OPERATOR(node_53, op_PLUS, node_51, node_52);
OPERATOR(node_51, op_TIMES, a_12,b_23);
DATUM(node_51);
OPERATOR(node_52, op_TIMES, a_13, b_33);
DATUM(node_52);
DATUM(node_53);
DATUM(node_54);
OPERATOR(c_20, op_PLUS, node_56, node_61);
```

```
OPERATOR(node_56, op_TIMES, a_20, b_00);
DATUM(node_56);
OPERATOR(node_61, op_PLUS, node_57, node_60);
OPERATOR(node_57, op_TIMES, a_21, b_10);
DATUM(node_57);
OPERATOR(node_60, op_PLUS, node_58, node_59);
OPERATOR(node_58, op_TIMES, a_22, b_20);
DATUM(node_58);
OPERATOR(node_59, op_TIMES, a_23, b_30);
DATUM(node_59);
DATUM(node_60);
DATUM(node_61);
OPERATOR(c_21, op_PLUS, node_63, node_68);
OPERATOR(node_63, op_TIMES, a_20, b_01);
DATUM(node_63);
OPERATOR(node_68, op_PLUS, node_64, node_67);
OPERATOR(node_64, op_TIMES, a_21, b_11);
DATUM(node_64);
OPERATOR(node_67, op_PLUS, node_65, node_66);
OPERATOR(node_65, op_TIMES, a_22, b_21);
DATUM(node_65);
OPERATOR(node_66, op_TIMES, a_23, b_31);
DATUM(node_66);
DATUM(node_67);
DATUM(node_68);
OPERATOR(c_22, op_PLUS, node_70, node_75);
```

```
OPERATOR(node_70, op_TIMES, a_20, b_02);
DATUM(node_70);
OPERATOR(node_75, op_PLUS, node_71, node_74);
OPERATOR(node_71, op_TIMES, a_21, b_12);
DATUM(node_71);
OPERATOR(node_74, op_PLUS, node_72, node_73);
OPERATOR(node_72, op_TIMES, a_22, b_22);
DATUM(node_72);
OPERATOR(node_73, op_TIMES, a_23, b_32);
DATUM(node_73);
DATUM(node_74);
DATUM(node_75);
OPERATOR(c_23, op_PLUS, node_77, node_82);
OPERATOR(node_77, op_TIMES, a_20, b_03);
DATUM(node_77);
OPERATOR(node_82, op_PLUS, node_78, node_81);
OPERATOR(node_78, op_TIMES, a_21, b_13);
DATUM(node_78);
OPERATOR(node_81, op_PLUS, node_79, node_80);
OPERATOR(node_79, op_TIMES, a_22, b_23);
DATUM(node_79);
OPERATOR(node_80, op_TIMES, a_23, b_33);
DATUM(node_80);
DATUM(node_81);
DATUM(node_82);
OPERATOR(c_30, op_PLUS, node_84, node_89);
```

```
OPERATOR(node_84, op_TIMES, a_30, b_00);
DATUM(node_84);
OPERATOR(node_89, op_PLUS, node_85, node_88);
OPERATOR(node_85, op_TIMES, a_31, b_10);
DATUM(node_85);
OPERATOR(node_88, op_PLUS, node_86, node_87);
OPERATOR(node_86, op_TIMES, a_32, b_20);
DATUM(node_86);
OPERATOR(node_87, op_TIMES, a_33, b_30);
DATUM(node_87);
DATUM(node_88);
DATUM(node_89);
OPERATOR(c_31, op_PLUS, node_91, node_96);
OPERATOR(node_91, op_TIMES, a_30, b_01);
DATUM(node_91);
OPERATOR(node_96, op_PLUS, node_92, node_95);
OPERATOR(node_92, op_TIMES, a_31, b_11);
DATUM(node_92);
OPERATOR(node_95, op_PLUS, node_93, node_94);
OPERATOR(node_93, op_TIMES, a_32, b_21);
DATUM(node_93);
OPERATOR(node_94, op_TIMES, a_33, b_31);
DATUM(node_94);
DATUM(node_95);
DATUM(node_96);
OPERATOR(c_32, op_PLUS, node_98, node_103);
```

```
OPERATOR(node_98, op_TIMES, a_30, b_02);
DATUM(node_98);
OPERATOR(node_103, op_PLUS, node_99, node_102);
OPERATOR(node_99, op_TIMES, a_31, b_12);
DATUM(node_99);
OPERATOR(node_102, op_PLUS, node_100, node_101);
OPERATOR(node_100, op_TIMES, a_32, b_22);
DATUM(node_100);
OPERATOR(node_101, op_TIMES, a_33, b_32);
DATUM(node_101);
DATUM(node_102);
DATUM(node_103);
OPERATOR(c_33, op_PLUS, node_105, node_110);
OPERATOR(node_105, op_TIMES, a_30, b_03);
DATUM(node_105);
OPERATOR(node_110, op_PLUS, node_106, node_109);
OPERATOR(node_106, op_TIMES, a_31, b_13);
DATUM(node_106);
OPERATOR(node_109, op_PLUS, node_107, node_108);
OPERATOR(node_107, op_TIMES, a_32, b_23);
DATUM(node_107);
OPERATOR(node_108, op_TIMES, a_33, b_33);
DATUM(node_108);
DATUM(node_109);
DATUM(node_110);
OUTPUT(c_00);
```

```
OUTPUT(c_01);
OUTPUT(c_02);
OUTPUT(c_03);
OUTPUT(c_10);
OUTPUT(c_11);
OUTPUT(c_12);
OUTPUT(c_13);
OUTPUT(c_20);
OUTPUT(c_21);
OUTPUT(c_22);
OUTPUT(c_23);
OUTPUT(c_30);
OUTPUT(c_31);
OUTPUT(c_32);
OUTPUT(c_33);
```

Source Code A.12: Generated AGP code for listing A.11 (matrix multiplication 4x4)

```

int fact (int x){
  if(x > 0){
    return x*fact(x-1);
  }else{
    return 1;
  }
}

```

```
10 int single(int t11){
11   int fact_in;
12   fact_in = t11+20;
13   int t12;
14   t12 = fact(fact_in);
15   int t13;
16   t13 = fact(t11);
17   int result;
18   result = t12+t13;
19
20   result = fact(fact_in);
21   result = fact(fact_in);
22   result = fact(fact_in);
23   result = fact(fact_in);
24   result = fact(fact_in);
25   result = fact(fact_in);
26   result = fact(fact_in);
27   result = fact(fact_in);
28   result = fact(fact_in);
29   result = fact(fact_in);
30   result = fact(fact_in);
31   result = fact(fact_in);
32   result = fact(fact_in);
33   result = fact(fact_in);
34   result = fact(fact_in);
35   result = fact(fact_in);
```

```
  result = fact(fact_in);
  result = fact(fact_in);
  result = fact(fact_in);
  result = fact(fact_in);
  result = fact(fact_in);
  result = fact(fact_in);

  return result;
}

int main(){

  int i1;
  int i2;
  int i3;
  int i4;
  i1 = 1;
  i2 = 1;
  i3 = 1;
  i4 = 1;

  int o[4];

  o[0] = single(i1);
  o[1] = single(i2);
  o[2] = single(i3);
```

```
  o[3] = single(i4);

  return o;

}
```

Source Code A.13: C example for cascade

```
SUBGRAPH(fact)
DATUM(x);
INPUT(x);
OPERATOR(node_8, op_IF, node_1, node_4);
OPERATOR(node_9, op_ELSE, node_1, node_6);
OPERATOR(node_10, op_MERGE, node_8, node_9);
OUTPUT(node_10);
DATUM(node_8);
DATUM(node_9);
DATUM(node_10);
OPERATOR(node_12, op_IF, node_1, node_3);
DATUM(node_12);
OPERATOR(node_4, op_TIMES, x, nextiter);
EXPAND(fact, MAP_IN(x,node_12);MAP_OUT(node_10,nextiter););
OPERATOR(node_3, op_MINUS, x, node_2);
CONST(node_2, 1);
DATUM(node_3);
DATUM(node_4);
CONST(node_6, 1);
```

```
20 OPERATOR(node_1, op_ISGREATER, x, node_0);
21 CONST(node_0, 0);
22 DATUM(node_1);
23
24 SUBGRAPH(single)
25 DATUM(t11);
26 INPUT(t11);
27 DATUM(fact_in);
28 OPERATOR(node_15, op_PLUS, t11, node_14);
29 CONST(node_14, 20);
30 DATUM(node_15);
31 DATUM(t12);
32 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(t12,node_17););
33 DATUM(node_17);
34 DATUM(t13);
35 EXPAND(fact, MAP_IN(x,t11);MAP_OUT(t13,node_19););
36 DATUM(node_19);
37 DATUM(result);
38 OPERATOR(node_21, op_PLUS, fact, fact);
39 DATUM(node_21);
40 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_23););
41 DATUM(node_23);
42 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_25););
43 DATUM(node_25);
44 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_27););
45 DATUM(node_27);
```

```
46 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_29););
47 DATUM(node_29);
48 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_31););
49 DATUM(node_31);
50 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_33););
51 DATUM(node_33);
52 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_35););
53 DATUM(node_35);
54 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_37););
55 DATUM(node_37);
56 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_39););
57 DATUM(node_39);
58 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_41););
59 DATUM(node_41);
60 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_43););
61 DATUM(node_43);
62 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_45););
63 DATUM(node_45);
64 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_47););
65 DATUM(node_47);
66 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_49););
67 DATUM(node_49);
68 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_51););
69 DATUM(node_51);
70 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_53););
71 DATUM(node_53);
```

```
72 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_55););
73 DATUM(node_55);
74 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_57););
75 DATUM(node_57);
76 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_59););
77 DATUM(node_59);
78 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_61););
79 DATUM(node_61);
80 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_63););
81 DATUM(node_63);
82 EXPAND(fact, MAP_IN(x,node_15);MAP_OUT(result,node_65););
83 DATUM(node_65);
84 OUTPUT(result);
85
86 SUBGRAPH(main)
87 DATUM();
88 DATUM(i1);
89 DATUM(i2);
90 DATUM(i3);
91 DATUM(i4);
92 CONST(node_67, 1);
93 CONST(node_68, 1);
94 CONST(node_69, 1);
95 CONST(node_70, 1);
96 DATUM(o_0);
97 DATUM(o_1);
```

```
98  DATUM(o_2);
99  DATUM(o_3);
100 EXPAND(node_71, MAP_IN(t11,i1);MAP_OUT(o_0,node_72););
101 DATUM(node_72);
102 EXPAND(node_73, MAP_IN(t11,i2);MAP_OUT(o_1,node_74););
103 DATUM(node_74);
104 EXPAND(node_75, MAP_IN(t11,i3);MAP_OUT(o_2,node_76););
105 DATUM(node_76);
106 EXPAND(node_77, MAP_IN(t11,i4);MAP_OUT(o_3,node_78););
107 DATUM(node_78);
108 OUTPUT(o);
```

Source Code A.14: Generated AGP code for listing A.13 (cascade)

Bibliography

[1] Peng Wang and John McAllister. Streaming elements for fpga signal and image processing accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(6):2262–2274, 2016.

[2] Re'em Harel, Idan Mosseri, Harel Levin, Lee-or Alon, Matan Rusanovsky, and Gal Oren. Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming*, 48(1):1–31, 2020.

[3] Openmp. `https://www.openmp.org/`. Accessed: 2022-04-26.

[4] Opencl. `https://opencl.org/`. Accessed: 2022-04-26.

[5] Miguel Angel Aguilar, Rainer Leupers, Gerd Ascheid, and Luis Gabriel Murillo. Automatic parallelization and accelerator offloading for embedded applications on heterogeneous mpsocs. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[6] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling a high-level directive-based programming model for gpgpus. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 105–120. Springer, 2013.

[7] Yan Solihin. *Fundamentals of parallel multicore architecture*. CRC Press, 2015.

[8] Tom Guillaumet, Aayush Sharma, Eric Feron, Madhava Krishna, Ranjani Narayan, Philippe Baufreton, Francois Neumann, and Emmanuel Grolleau. Using reconfigurable multi-core architectures for safety-critical embedded systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–6. IEEE, 2016.

[9] Millad Ghane, Sunita Chandrasekaran, Robert Searles, Margaret S Cheung, and Oscar Hernandez. Path forward for softwarization to tackle evolving hardware. In *Disruptive Technologies in Information Sciences*, volume 10652, page 106520O. International Society for Optics and Photonics, 2018.

[10] Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, et al. A hardware/software stack for heterogeneous systems. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):243–259, 2017.

[11] Farshad Baharvand and Seyed Ghassem Miremadi. Lexact: Low energy n-modular redundancy using approximate computing for real-time multicore processors. *IEEE Transactions on Emerging Topics in Computing*, 8(2):431–441, 2017.

[12] Moritz Kreutzer, Jonas Thies, Melven Röhrig-Zöllner, Andreas Pieper, Faisal Shahzad, Martin Galgon, Achim Basermann, Holger Fehske, Georg Hager, and Gerhard Wellein. Ghost: building blocks for high performance sparse linear algebra on heterogeneous systems. *International Journal of Parallel Programming*, 45(5):1046–1072, 2017.

[13] Joshua Fryer and Paulo Garcia. Strategies for heterogeneous multi-core processing based on graph programming. In *2021 7th International Conference on Computing and Data Engineering*, pages 54–60, 2021.

[14] Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua. *Languages and Compilers for Parallel Computing: 5th International Workshop, New Haven, Connecticut, USA, August 3-5, 1992. Proceedings*, volume 757. Springer Science & Business Media, 1993.

[15] Samuel P Midkiff. Automatic parallelization: an overview of fundamental compiler techniques. *Synthesis Lectures on Computer Architecture*, 7(1):1–169, 2012.

[16] Joshua Fryer and Paulo Garcia. Towards a programming paradigm for reconfigurable computing: Asynchronous graph programming. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1721–1728. IEEE, 2020.

[17] Michael Wolfe. Parallelizing compilers. *ACM Computing Surveys (CSUR)*, 28(1):261–262, 1996.

[18] Thomas J LeBlanc and Evangelos P Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *[1992] Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 254–263. IEEE, 1992.

[19] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, 2003.

[20] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[21] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[22] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[23] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing*, 2(4):382–400, 2020.

[24] Hamid Arabnejad, João Bispo, João MP Cardoso, and Jorge G Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of c applications. *The Journal of Supercomputing*, 76(9):6753–6785, 2020.

[25] Suejb Memeti and Sabri Pllana. Papa: A parallel programming assistant powered by ibm watson cognitive computing technology. *Journal of computational science*, 26:275–284, 2018.

[26] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 128–138, 2016.

[27] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A McKee, Sven Karlsson, and Ayal Zaks. Parallelizing more loops with compiler guided refactoring. In *2012 41st International Conference on Parallel Processing*, pages 410–419. IEEE, 2012.

[28] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, Samuel P Midkiff, et al. The cetus source-to-source compiler infras-

tructure: overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 2013.

[29] Marek Palkowski and Wlodzimierz Bielecki. Traco parallelizing compiler. In *Soft Computing in Computer and Information Science*, pages 409–421. Springer, 2015.

[30] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE micro*, 28(4):13–27, 2008.

[31] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. Automatic c-to-cuda code generation for affine programs. In *International Conference on Compiler Construction*, pages 244–263. Springer, 2010.

[32] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

[33] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.

[34] Alok Mishra, Martin Kong, and Barbara Chapman. Kernel fusion/decomposition for automatic gpu-offloading. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 283–284. IEEE, 2019.

[35] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. Bridging opencl and cuda: a comparative analysis and translation. In *SC'15:*

Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12. IEEE, 2015.

[36] Paul Sathre, Mark Gardner, and Wu-chun Feng. On the portability of cpu-accelerated applications via automated source-to-source translation. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 1–8, 2019.

[37] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and Wen-Mei W Hwu. Cuda-lite: Reducing gpu programming complexity. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. Springer, 2008.

[38] Tianyi David Han and Tarek S Abdelrahman. hi cuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 52–61, 2009.

[39] Nishkam Ravi, Yi Yang, Tao Bao, and Srimat Chakradhar. Apricot: an optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 47–58, 2012.

[40] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[41] John A Stratton, Sam S Stone, and Wen-Mei W Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30. Springer, 2008.

[42] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364. IEEE, 2010.