HIGH-PERFORMANCE PARALLEL PROGRAMMING FRAMEWORK USING TEMPLATE-BASED STATIC OPTIMIZATION

BY

SHENGZHAO WU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Wen-mei W. Hwu

ABSTRACT

How to program a parallel machine has always been a major research problem. Many tools, languages and libraries are developed in order to make parallel programming more accessible for most users. However, no matter what approach is taken to program a parallel machine, there is always a trade-off between productivity, performance and portability. It is very hard to develop a system that only requires short and concise code to achieve close-to-optimal performance on a wide range of parallel machines.

In this thesis, a novel programming framework is developed to achieve a good combination of productivity, performance and portability. The programming framework is designed based on computation patterns that contain parallel information. The programming framework can efficiently map these computation patterns onto a parallel machine. The programming framework also utilizes the C++ templates to generate optimized code for different compositions of computation patterns. It uses a novel way to implement the computation patterns that allow automatic high-level optimization at compile time. Through the benchmarks, it shows that the programming framework can effectively express the computation kernels in few lines of code and achieve the performance of their optimized C code on multi-core CPUs.

To my parents, for their love and support

ACKNOWLEDGMENTS

I would like to express my special appreciation and thanks to my advisor Professor Wen-mei W. Hwu. You have been a tremendous mentor for me. I would like to thank you for the opportunities and the guidance that you have provided me. Your words of wisdom will continue to inspire me in the future both in my professional career and in my personal life.

A special thanks goes to my family and all my friends for their help in these educational years. I could not have gone so far without your support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

CPU	Central processing unit
CSR	Control status register
CUDA	Compute unified device architecture
DDR	Double data rate
GDDR	Graphics double data rate
GPU	Graphics processing unit
MIMD	Multiple instruction, multiple data
NOP	No operation
SIMD	Single instruction, multiple data
SPMD	Single program, multiple data

CHAPTER 1

INTRODUCTION

With the emergence of various kinds of parallel processors and accelerators, a tremendous amount of attentions has been shifted into developing parallel algorithms and parallel applications in recent years. Numerous programming systems have been built showing very promising improvement over traditional sequential machines [1, 2, 3]. We have reason to believe that the new parallel computer architectures are going to keep transforming the landscape of various areas like scientific computing, computer vision and other fields that require high throughput computation.

However, writing high-performance code on parallel machines is much harder than on traditional single-core CPUs. In order to produce close-to-optimal performance on a targeted computer architecture, it requires not only a thorough understanding of the application itself, but also the knowledge of detailed computer architecture, including memory controller, cache, hardware scheduler and instruction pipeline. The high technical entryway has become one of the biggest road blocks for the wider adoption of parallel hardware.

In addition to the high development cost of parallel applications, optimized parallel software is very hard to maintain. High-performance code is usually written in low-level programming languages like C or domain-specific languages like CUDA. The optimized code in these languages usually contains manual tiling, loop unrolling and other optimization techniques. These optimizations often obfuscate the code and decrease the readability.

Furthermore, if programmers expect good performance from several different parallel machines, they are required to manually tune the program for each targeted platform; this is even true for the hardware of different generations of the same architecture with minor specification variations. Table 1.1 shows the different optimization choices across two generations of Intel CPUs for the best performance in scan benchmark (details about the scan

algorithm are discussed in Section 2.2.3). These choices will further increase the size of the code base and the time of development for high-performance parallel applications.

Table 1.1: Target specific optimizations for two Intel CPUs for the Scan benchmark

CPUs	Intel Core 2 Duo	Intel Core i7
Algorithm	Scan-Scan-Map	Reduce-Scan-Scan
Optimizations	Vectorization	Parallelization, Vectorization
Tile Size	1 MB	4 MB

There are generally three approaches to tackle the problem. One is to define a language specifically for parallel machines. One of the most influential languages in this category is OpenCL [4]. With the support from vendors, OpenCL has become a popular choice of parallel accelerator programming. Being a low-level programming language, OpenCL shares the common drawbacks of C and CUDA. Multiple studies have shown that performance is not portable across different platforms [5]. Despite the fact that many studies show that high performance can be achieved with OpenCL on different kinds of accelerators and also in heterogeneous environments, no evidence shows OpenCL has better productivity and maintainability compared to C or CUDA [6].

Other languages and tools focus on building abstractions through high-level languages or function interfaces. Languages like Copperhead [7] and Triolet [8] and numerous parallel libraries fall into this category. Many of them report very good performance together with high productivity and ease of maintenance. In those toolchains and language frameworks, the responsibility of optimization will be solely dependent on the library developers and compiler writers. Without a wider adoption of the tools or the support from all the vendors, these languages and tools lack the ability for fine-tuning for a wide range of chips and will have a hard time keeping up with the evolving chip industry.

The third approach is to develop a compiler that can automatically transform loops in sequential code into parallel versions and apply optimizations to the verions [9, 10, 11]. Although this approach requires minimal effort

from application developers, current parallel compilers are limited by their model of analysis and can only handle very limited cases due to the limitation of static analysis. For example, one of the common ways to model a loop is using polytopes, which partially relies on alias analysis, to model the loop iterations. Since the alias analysis problem is undecidable, the compiler can't always construct the polyhedral model of the loop, even if the loop is parallelizable. Due to those limitations, these tools often can only achieve a fraction of the speedup of the hand-tuned parallel code.

In this thesis, a different approach is explored to find a compromise between the previous approaches while retaining many of their benefits. We observe that there are more similarities than differences between current parallel architectures. If we can abstract the parallelism of the application through different computation patterns (described in Section 2.2), a framework can be developed to map these computation patterns to different parallel machines while reusing as much common optimization as possible. TIC (Triolet In C++), a programming framework inspired by Triolet, is built to enable composable computation patterns in C++. With the TIC framework, application developers will be able to produce parallel programs almost as efficiently as with high-level language, and in the meantime, optimizations can be automatically applied to get better performance. The resulting program can achieve close-to-optimal performance compared to low-level programming languages like C. This will allow programmers to be truly focusing on algorithm development and problem solving rather than spending time exploring minor tweaks to get more performance out of parallel machines. The TIC framework has the following contributions:

- Generalizing the common optimization techniques for the shared-memory parallel computer architectures.
- Identifying the most used computation patterns in parallel computing and the different implementations of them on different architectures.
- Develop a systematic way of constructing computation kernel with common computation patterns.
- The first programming framework applying optimizations using templates utilizing the high-level information available in the source code.

- Demonstrating close-to-optimal performance on multi-core CPUs with C++ and template metaprogramming.

The thesis is organized in the following way: Chapter 2 describes the background and related work. Chapter 3 introduces the TIC programming framework at a high level, and Chapter 4 describes the implementation details of different parts in the framework. Chapter 5 analyzes the performance of the TIC framework using common computation benchmarks, and Chapter 6 discusses further improvements that can be made to the framework for better performance or usability. The thesis finally concludes with Chapter 7.

CHAPTER 2

BACKGROUND

In this chapter, background of the TIC framework is introduced including the basics of parallel architectures, general computation patterns as well as previous work in programming languages including template metaprogramming and C++ features.

2.1 Parallel Hardware

As different kinds of parallel machines emerge, it is generally believed that all the parallel machines are so distinct that completely different sets of optimization should be applied. Figure 2.1 shows a generic design of a shared-memory parallel architecture that is used in both multi-core CPUs and mainstream GPUs. Despite the distinct design execution units, their memory systems are similar. We observe that these shared-memory parallel architectures have almost identical memory hierarchies and prefer similar memory access patterns. If exploited properly, a well-designed set of memory optimizations should apply to a wide-range of shared-memory architectures. For the execution units, they are sufficiently distinct across different platforms that usually the parallel kernel is impossible to represent by the same low-level code. However, if the parallel kernel is expressed in high-level language, it is much easier to find a mapping from the kernel to the execution units. These two observations led to the design of the TIC framework. The following two subsections break up the parallel architecture into memory system and execution unit, and describe each with its own approach.

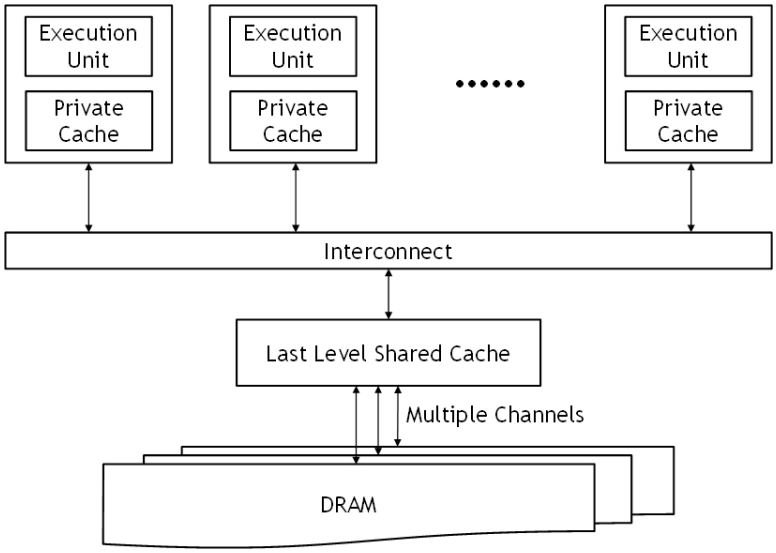


Figure 2.1: Common design for shared-memory parallel architectures.

2.1.1 Memory System

The memory system described in the thesis includes the off-chip memory (DRAM, either DDR3 or GDDR5) and the on-chip memory (caches, both shared and private, and registers). Although the real hardware implementation of a memory system on different chips can be drastically different in terms of channel numbers, burst widths, caches sizes, etc., almost all memory systems prefer the same kind of access pattern.

A good memory access pattern includes good spatial locality and good temporal locality. Spatial locality will fully utilize the channel and burst from DRAM, and maximize the efficiency of cache. Good spatial locality also means better utilization of the prefetch unit of the chip if it has one. Spatial locality mainly results from an appropriate choice of the data structure and the order of traversal. Temporal locality is often exploited by caching the data in on-chip memory for future use. It is important because it helps to reduce the number of memory requests that actually reach the DRAM, and thereby saving the precious DRAM bandwidth. Different from spatial locality, temporal locality is mainly determined by the algorithm.

However, there are some important differences between memory systems: bandwidth and cache size. The differentiations will often result in different tiling decisions and occasionally will result in different algorithms. These qualities need to be taken into consideration if the framework is to be designed

for close-to-optimal performance.

It has been known for a long time that computation power is growing much faster than memory bandwidth. It is crucial to save as much memory bandwidth as possible, and it is especially true for memory-bound applications. The TIC framework is carefully designed based on the memory system features to achieve spatial locality and temporary locality through different methods and make different tiling decisions based on the hardware specifications and the algorithm (see Chapter 4).

2.1.2 Execution Unit

Unlike memory systems share many common features and few differences, execution units are sufficiently different from each other that a distinct set of optimizations is needed. Although the framework design considers all shared-memory based parallel hardwares, platforms like CPU is the only target in this stage. For the execution unit, SIMD and MIMD is focused because modern CPUs have vector units (SIMD) and multi-core (MIMD), both of which should be exploited at the same time.

In terms of automatic compiler optimization, vectorization and parallelization are two completely different loop transformations. Shown in Listing 2.1, though vectorization and parallelization are both loop strip-mining, vectorization results in an innermost loop while parallelization results in an outermost loop. This also means both transformations are independent and can be concerned separately in the framework.

Listing 2.1: A simple vector add as the example of vectorization and parallization, the innermost loop is mapped to a vector unit and the outermost loop is distributed across CPUs

```
1  void vector_add(int* a, int* b, int* result, int size)
2  {
3    // assume size is a multiple of CORE_NUM*VECTOR_SIZE
4    // parallelization, map each iteration to a CPU core
5    for (int core = 0; core < CORE_NUM; core ++)
6    {
7      for (int i = 0; i < size/(CORE_NUM*VECTOR_SIZE); i ++)
8      {
9        // vectorization, map the entire loop to a vector unit
10       for (int v = 0; v < VECTOR_SIZE; v ++)
11       {
12         int index = core * (size/CORE_NUM) + i * VECTOR_SIZE + v;
13         result[index] = a[index] + b[index];
14       }
15     }
16   }
17 }
```

In addition to the difference in terms of the granularity of parallelism, SIMD and MIMD also handle branches differently. Consider a parallel loop with a branch condition inside. Branch divergence is when the branch in the loop is taken in some iterations but not taken in some other iterations. Because SIMD units execute the same instruction across all data, SIMD has to execute both branches with predication. MIMD does not have the limitation of SIMD, thus MIMD allows individual threads to behave different at the branch. However, by allowing the different behavior across the threads, the threads will be out of synchronization over time, so it is not possible to expect coordinations across the threads without explicit synchronization barriers.

2.2 Computation Patterns

Although there is much research showing that the optimizations can be done either automatically [12] or semi-automatically with user annotation [13], it is difficult and time-consuming for the compiler to determine all the opti-

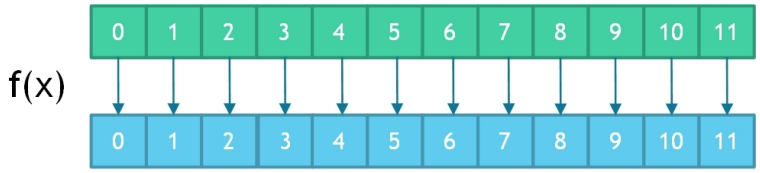


Figure 2.2: Map computation pattern.

mization opportunities. On the other hand, if the computation is expressed in a high-level language, the function itself might contain all the information needed to do optimization.

Rather than use a **for** loop to generalize all the parallelism and rely on the compiler to recover the information, it is more efficient to maintain as much parallel information as possible in the source code. To express the parallelism, programmers can use functions that implicitly suggest the parallel information, memory access information or the dependency information. These functions are called *computation patterns*.

Some of the basic computation patterns will be discussed in the following subsections, including map (Section 2.2.1), reduce (Section 2.2.2) and scan (Section 2.2.3). These functions are commonly used in various benchmarks and implemented in many other languages and frameworks. These computation patterns will serve as basics in the TIC framework. Many other patterns are supported in the TIC framework including stencil, histogram, sorting and permutation, but they will not be discussed here.

2.2.1 Map

Map is a computation pattern that addresses perfect parallelism (shown in Figure 2.2). An operation is applied to every element of the input, and there are no dependencies between each operation. In this case, the order of traversing the elements and the order of computation can be freely rearranged.

Parallelization is trivial for map since all outputs can be computed independently. The output is naturally partitioned according to the number of available cores and distributes these partitions evenly across all the cores. If there are branches in the map, there can be load balancing issues. Fortunately, there are existing tools like OpenMP and TBB [14] that take load balancing into consideration. These tools are directly used in the TIC frame-

work to map the workload to multi-core CPUs.

Vectorization is also obvious except when branches exist in the map. Several consecutive input elements can be packed into a vector, and SIMD instructions can be used to compute the corresponding output from the input vector. There is research showing how vectorization can be achieved by transforming high-level language [15]. In this thesis, multi-tier dynamic parallelism [16] is used to effectively exploit vector units when static branches are encountered. The technique will be described in detail in Section 4.4.

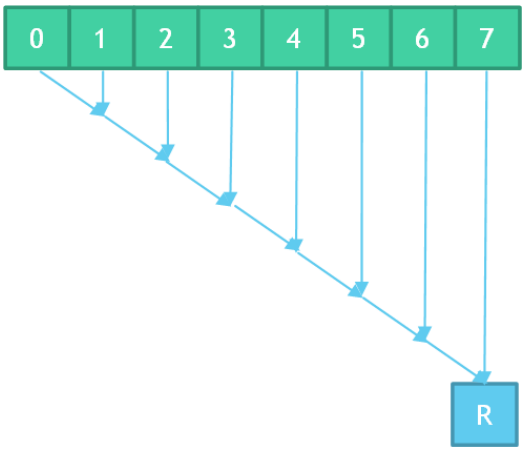
2.2.2 Reduce

Reduce in the TIC framework refers to the specific kind of reduce computation pattern that is associative, thus it can be parallelized with proper communication or synchronization. Figure 2.3a shows the naive sequential implementation, and Figure 2.3b shows the tree reduction algorithm that performs the parallel reduction. On shared-memory architectures, the best implementation is usually a hybrid of the two which takes the advantage of 1) the benefit of the sequential version on one core which has a good memory access pattern and computation efficiency, and 2) the benefit of the tree version that allows the distribution of work across multiple computation units [17]. Different parallel platforms may use different pairings of the two patterns.

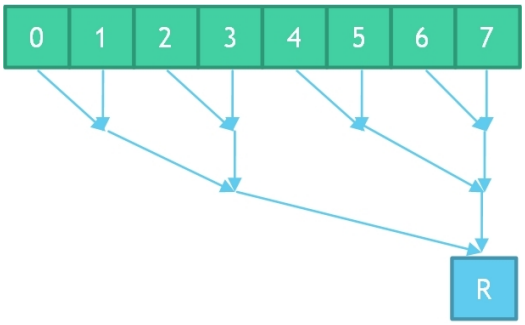
In the TIC framework, the detailed implementation of reduce is abstracted away from the programmer so there is no concern about how to partition the work between sequential or tree implementations. Also, the best communication or synchronization method is selected for the user. For multi-core CPUs, vector shuffle is used to communicate between vectors within one core, and atomics operation is used to communicate between cores but the decision can be different for the targets with different features.

2.2.3 Scan

Scan is an all-reduce computation pattern which can be considered as a hybrid of map and reduce. It produces one output for each input element like map, but it has a dependence similar to reduce. Scan is hard to parallelize



(a) Sequential reduce



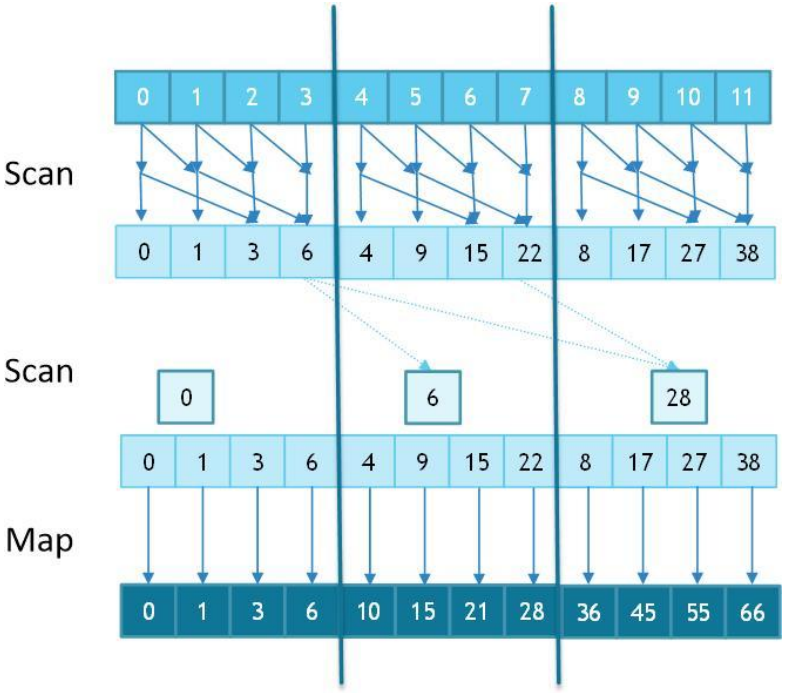
(b) Tree reduce

Figure 2.3: Reduce computation patterns. (a) A sequential reduce that has strict dependencies across the output elements but a good memory access pattern. (b) A tree reduce where each stage can be computed separately, but it has a less optimal memory access pattern.

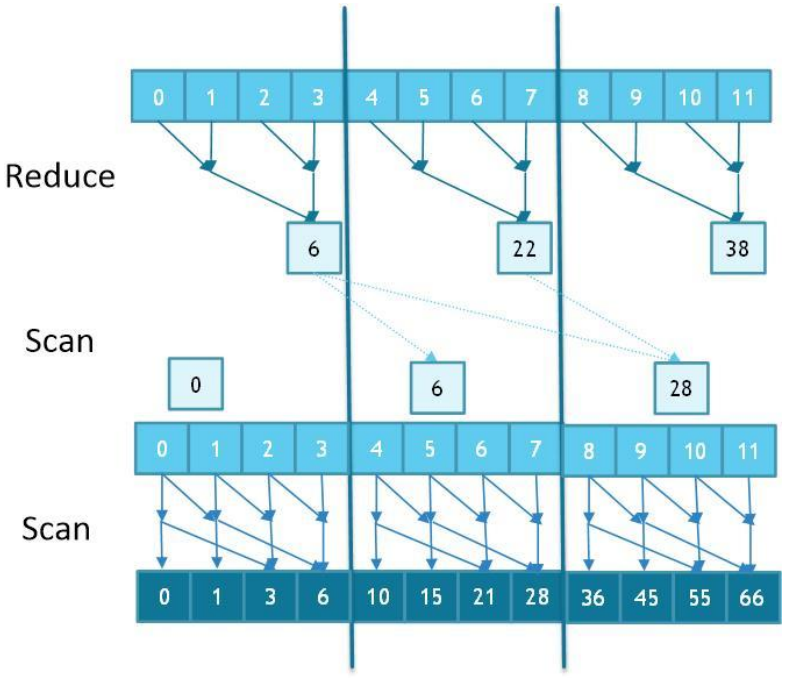
and vectorize because the naive scan is highly sequential. The most naive scan implementation also has a good memory access pattern, so it is not easy to get speedup for scan on the parallel machine. There is research to show how to implement scan on different parallel platforms [18, 19]. The approach used in the TIC framework is either scan-scan-map or reduce-scan-scan. Scan-scan-map implementation is shown in Figure 2.4a. This is the approach commonly seen in the literature. It computes the local partial scan result and stores it in the destination. Then a global communication occurs to compute the global scan using the last value from the local scan. Finally, the partial scan result from the local scan is updated using the global scan results through a map operation. Reduce-scan-scan is our own scan implementation (shown in Figure 2.4b). Reduce-scan-scan only computes the local reduction and then uses the local reduction result to compute global scan. In the final stage, a local scan is computed using the global scan result as the initial value. The later approach saves memory bandwidth, but it requires more computation. The algorithm decision can be made static based on two parameters: the computation intensity of the scan operator and the input type to the scan operation. A computation intensive scan operator will shift the scan to a computation bounded kernel, thus the memory bandwidth saving from Reduce-Scan-Scan is not benefiting the overall runtime. If the input to the scan operation is a generator, Reduce-Scan-Scan needs to call the generator twice, both in the first reduce stage and the third scan stage. Under this situation, Scan-Scan-Map is used because the generator is only called once in its local scan stage.

2.3 Template Metaprogramming

Template is a very powerful feature in C++. It is designed to achieve generic programming in C++. Programmers can write functions and classes with template types, and the compiler will generate the actual implementation when the data type is supplied during the compile time. STL [20] library is the commonly used library in C++ that uses templates to provide containers to programmers. Blitz++ uses the C++ template to drastically improve C++ array performance. It provides Fortran-like array operations, and their performance is comparable to that of Fortran [21]. Matrix Template Library



(a) Scan-scan-map.



(b) Reduce-scan-scan.

Figure 2.4: Scan computation patterns with summation operator (numbers in the boxes represent inputs or partial sums).

[22] is another widely used template library which provides high performance linear algebra through template matrix types and overloaded operators.

Template metaprogramming is a technique that fully exploits the power of the C++ template. It utilizes templates to do computation during the compile time. Although the computation power that can be achieved by a C++ compiler is much slower than that achieved by actual C++ code, template metaprogramming is still a useful technique to trade compilation time for improved runtime. Some of the most commonly used tricks with template metaprogramming include static polymorphism and static code generation.

Static polymorphism is achieved by using the C++ template system to derive the correct member function to call during the compile time. It takes advantage of the fact that many of the concrete types in the program can be deduced during the compile time. Thus the exact member function that gets called from a call site can be determined at compile time. Using this approach, the runtime cost of dynamic polymorphism (using virtual functions and vtables) can be eliminated. In the design of the TIC framework, the same approach is used to generate faster code.

Static code generation is a technique to reorder code using the C++ template. This technique is used in many libraries to provide a friendly interface to the users. With static code generation, the users can write the code in a more natural way, thus increasing the readability of the code, while the best performance can still be achieved. Listing 2.2 shows a code generation example that is used in C++ Boost library [23] and Thrust. In the example, a vector of floating point numbers is created, and its content is doubled and printed through the Thrust-style transform iterator. At line **15** and **17**, the begin and end iterators of the transform are created using the constructor provided at line **7**. The newly constructed transform iterators are used at line **19** for the comparison. In the body of the loop at line **20**, the de-reference operator is applied to the transform_iterator. The compiler will de-sugar the de-reference operator by inlining the function at line **9**, which computes the result from the vector by applying double operator. Note that no computations are done in the constructor and the transform iterator class serves as syntactic sugar for code reordering. In the TIC framework, the similar technique is used to achieve multiple optimizations, including fusion, padding, tiling, etc.

Listing 2.2: An example of code generation using templates

```
template <typename BaseIter>
class transform_iterator
{
  /* omit constructor and other member function */
  std::function<float(float)> func;
  BaseIter iter;
  transform_iterator(BaseIter a, std::function<float(float)> f)
      : iter(a), func(f) {}
  float operator* () { return f(*a); }
}

int main()
{
  std::vector<float> a(10, 1.0f);
  transform_iterator<std::vector<float>::iterator> begin
      (a.begin(), [](float x) { return x * 2; });
  transform_iterator<std::vector<float>::iterator> end
      (a.end(), [](float x) { return x * 2; });
  for (auto i = begin; i != end; ++i) {
      std::cout << *i << std::endl; // computation happens here
  }
}
```

2.4 C++ Features

The TIC framework is implemented using C++ because of the following reasons:

- C++ has a robust template system with type deduction. With some new features of C++11, a relatively clean user interface can be designed to hide type parameters in the templates away from the user and utilize type deduction to fill the types.
- C++ has a large user base. A framework implemented purely with C++ will allow it to be seamlessly integrated with existing code. Many C++ features can be used inside the framework directly as well, e.g. atomics.

- C++ has a mature and powerful compiler that can eliminate most of the language overhead. C++ is also sufficiently close to C and compatible with C that optimal low-level code can be generated.
- C++ already has many tools to allow the user to program in parallel. Examples include OpenMP, TBB, CUDA and CUB. These tools can be directly utilized by the framework to target the code to different platforms.

In the rest of the chapter, some new C++11 features are discussed. These features are all crucial to the implementation and the usability of the framework.

2.4.1 Auto Keyword

In C++11, the keyword **auto** is introduced to allow the compiler to deduce the type of variables for the programmers. This feature makes the life of the TIC framework's user much easier. As will be described in Chapter 4, the template type in the framework can include a lot of type parameters which will be extremely tedious for the user to specify. For example, the auto keyword is used at line **19** in Listing 2.2 which saves the user from specifying the long type name for the transform iterator. Moreover, the return type of a function can be different in different cases due to the fact that a different set of optimizations is applied. Unless the user fully understands every detail about the implementation of the framework, it is impossible for the users to deduce the type themselves. With C++11, the user can simply specify the return type of the function to be auto type and let the compiler to do all the work.

2.4.2 Lambda Expression

In C++11, lambda expression is introduced to allow easier construction of function objects. With lambda expression, the programmer does not need to declare a function and pass the function pointer together with all of its parameters. Instead, programmers can use a lambda expression to construct a function object inside a function. Lambda expression will also capture all

necessary variables available in the current scope and construct a **Functor** class.

Figure 2.5 shows the overhead of an add operation wrapped by a function or a lambda expression compared to the overhead of an inlined addition. When the addition is represented by a function, a function pointer is needed to pass the operation to the computation kernel. In this case, the compiler creates an indirect branch in the computation kernel. However, sometimes the compiler can statically prove that the destination function is always the same, and the compiler will optimize the program by inlining the destination function. Figure 2.5 shows that without automatic inlining, using a function pointer will cause a 11.7x slowdown. Otherwise, no overhead is observed. If a lambda expression is used to represent the addition, the lambda expression can be passed to the computation kernel as either a std::function type or a C++ template type. For the first case, the current compiler (Clang 3.5) seems not able to apply the same optimization to std::function as the one it applies to a function pointer. From the figure, a 13.7x overhead is observed. For the second case, a C++ template will capture the lambda and inline the lambda expression. There is no overhead for using lambda expressions and passing them as templates.

2.4.3 Others

Some other features used in the TIC framework include variadic templates and atomics. Variadic templates are used in tuple types and the tuples are used inside the zip operation. Atomics are used to make atomic operation easier to specify in the framework. Atomics are also used to implement the synchronization barrier and the communication between threads.

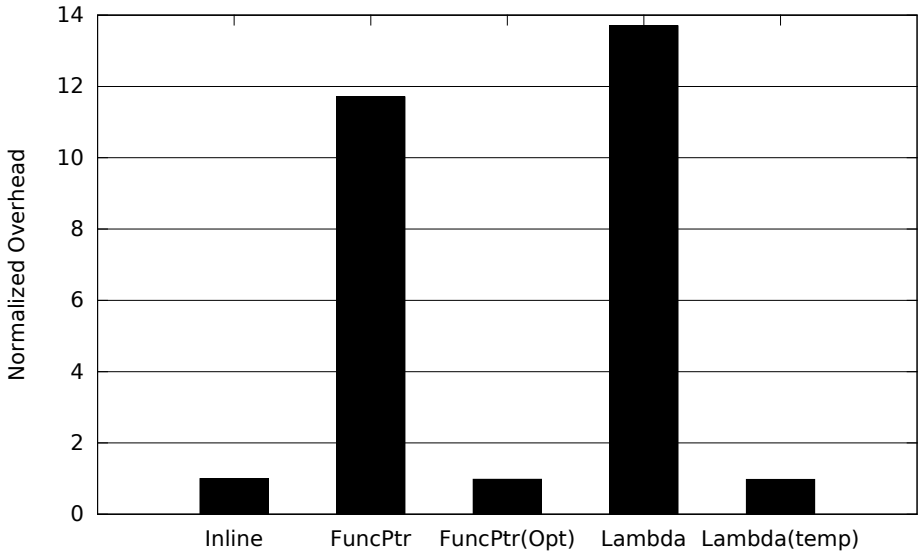


Figure 2.5: The performance comparison of a vector addition in which each add operation is represented by different methods. The result is normalized to the runtime of inlined version (compiled with Clang using -O3). Inline is the baseline with inlined addition. FuncPtr is the version which uses a function pointer to point to an add method. FuncPtr(Opt) is the version in which the function pointer is a constant so the destination function is inlined by the compiler. Lambda is the version which uses lambda expression to represent the addition, and the lambda is passed to the computation kernel as **std::function** type. Lambda(temp) is the version that the lambda is passed as template.

CHAPTER 3

FRAMEWORK

The TIC framework is designed to provide a high-performance parallel programming environment which is capable of automatically applying various kinds of optimization techniques while maintaining a clean and elegant interface.

In this chapter, the key concepts of the TIC framework are introduced. The chapter begins with the definitions of the terminologies and the overall work flow of the framework, The a simple program example is introduced to show programs are written using the framework.

3.1 Terminologies

The basic objects used in the programming framework are called *computation objects*. A computation object is an object containing both the input and a set of operations that convert input to output. It has a virtual size equivalent to the output size. It also contains all the optimizations that can be applied to the computation as well as the information necessary to make optimization decisions. The advantage of the computation object is that the input data structure can be optimized and more optimization can be applied as computations are specified to the data structure. Thus, a computation object is not only functioning as a generator which generates value when called, but also performs input data layout transformation and applies static optimizations to the computation code.

Users construct, modify and transform computation objects with *transform functions*. A transform function is a function that takes one data structure or one computation object as input, applies transform and returns a new computation object. A transform function can append new computation to the existing computation objects, apply data layout transformations and

apply high-level optimizations to any previous stages of the computation. All the optimization decision is also made in the transform function. The output computation object from the transform function is always heuristically the most optimal implementation to generate the output.

To compute the final result, a *consumer function* needs to be used. A consumer function evaluates the computation object and produces the final output. If applicable, the input data layout transform also happens in the consumer function.

3.2 Programming Workflow

Figure 3.1 shows the general work flow to write a kernel in the TIC framework. Starting from the input, users need to transform the input data into a computation object. If multiple inputs are present, users need to create one computation object for each input. After that, users can specify computations by applying a transform function to the existing computation object. Users can also combine or extend the existing computation objects using a zip transform function or const transform function. At the meantime, transform functions also take care of all the type deductions so that users are not required to figure out the exact type of all the computation objects. After all the computations are coded into the computation object, users need to call a consumer function to generate the output.

3.3 Example Program

Listing 3.1 shows a code example of a simple flow of the computation objects. A computation object is constructed from the **std::vector** type. The vector computation object will not allocate any memory to store the input but only capture a pointer to the input vector. At line **13**, a stencil object is created from the vector computation object. It is done through templating so the actual type of the stencil object is an extended type from the input vector. Upon the constructing of the stencil object (in **make_stencil_transform**), it may decide to apply transformation to the input vector, e.g. padding. In that case, the different type stencil computation object will be constructed

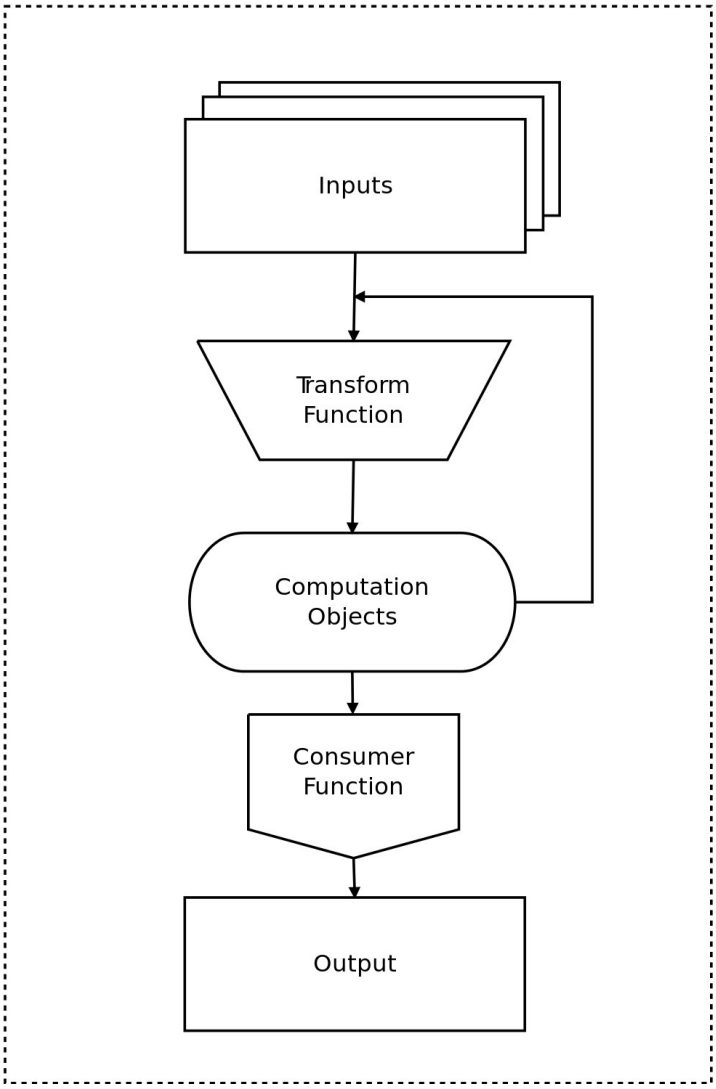


Figure 3.1: The workflow of the TIC framework.

and it will also reconstruct the vector computation object to be padded. The actual padding in the vector computation object will not be executed until the consumer stage which in this case is line **18**. Note this will introduce multiple copies of the vector computation object, but since they all contain only a pointer to the input vector and they do not have any dynamic memory allocations, the overhead is only a few extra scalar copies and is negligible. A detailed description of the technique is described in the next chapter.

Listing 3.1: Example code written in the TIC framework.

```
1  std::vector<int>
2    simple_stencil(const std::vector<int>& input)
3  {
4    // make input vector a computation object
5    // vector computation object
6    auto input_co = make_vector_object(input);
7    // transform input to 1D stencil object of width 3
8    // each element in stencil is 3 adjacent elements from input
9    // stencil object may requests padding to the input
10   // stencil computation object
11   auto stencil_co = make_stencil_transform<3>(input_co);
12   // map computation object
13   // reduce_operation will generate a reduce functor based on
14   //   element type of stencil_co and reduce operator (add_op)
15   auto map_co = make_map_transform(stencil_co, reduce_operation<add_op>());
16   // consumer that evaluate and output to vector
17   // it will pad the input vector and do map operation
18   std::vector<int> output = Evaluate<std::vector>(map_co);
19   return output;
20 }
```

CHAPTER 4

IMPLEMENTATION

In this chapter, the details of how optimizations are implemented are discussed. The key optimizations include fusion, tiling, vectorization and parallelization. In the following sections, the optimization strategies and the optimization parameters are introduced for each of the key optimization.

4.1 Static Optimization Using Templates

Section 2.2 suggests that computation patterns know more about the program itself so that using the information from the high-level computation pattern can result in better parallelization. The same argument can be made for many other high-level compiler optimizations like loop transforms and data layout transforms. Since the computation pattern itself contains program information about memory access pattern and sharing, the optimizations that rely on that program information can be done better at high-level by the language framework than at low-level by the compiler.

The approach used in the TIC framework to achieve those compile-time optimizations is through C++ templates. The template type in C++ can hold more than just type information. In the TIC framework, it is used to pass optimization information. The computation object in Listing 4.1 is an example implementation from the example code in the previous section (Listing 3.1). All the code shown is a simpler version of the code in the TIC framework and is hidden from the user. In this example, it shows a *data structure computation object* (**VectorCompObj**) which associates a data structure with its traversal order and layout transforms, a *extended computation object* (**StencilCompObj**) which extends an existing computation object with more compile time information like sharing pattern and dependency informations, and a transform function (**make_stencil_transform**)

which makes a decision whether various optimization are requried or not. In this example, the transform function makes the decision to always pad the input when the stencil computation object is created. If the data structure computation object does not suggest padding, the transform function will ask the data structure computation object to create a padded version and use the padded version to construct the stencil computation object. In reality, the padding decision can be more complicated, and a different type of stencil computation object needs to be returned. In that case, **std::enable_if** is used to make the compile time selection of the return type.

Listing 4.1: Computation object template class example

```
template <typename ElementType, /*some other types*/, bool Pad>
class VectorCompObj {
  // omit constructors and other member functions
  // ...
  // constexpr to evaluate at compile time
  constexpr bool _is_pad() { return Pad; }
  // member function to construct a padded type
  VectorCompObj<ElementType, ..., true> _pad() {
    return VectorCompObj<ElementType, ..., true> new_comp_obj(*this);
  }
  // _prepare function is called by consumer before access the class
  void _prepare() {
    if (Pad) {
      // do padding
    }
  }
}

template <int StencilSize, typename ParentType, /*some other types*/, bool Pad>
class StencilCompObj {
  // omit constructor and other member functions
  // ...
  private:
    // need a copy of parent because type conversion
    ParentType _parent;
}

template <int StencilSize, typename ParentType, /*some other types*/, bool Pad>
StencilCompObj<StencilSize,ParentType,/*some other types*/,true>
make_stencil_transform(ParentType comp_obj)
{
  if (!comp_obj._is_pad()) {
    return StencilCompObj<StencilSize,ParentType,...,true> (comp_obj._pad());
  } else {
    return StencilCompObj<StencilSize,ParentType,...,true> (comp_obj);
  }
}
```

In the extended computation object, it is required to keep a copy of the parent computation object due to the fact that a type conversion might happen. It is also required that the extended type calls the **_prepare** function before de-referencing the vector computation object. In **_prepare** function, since **Pad** can be determined in compile time, the compiler can easily reduce the function to a NOP if **Pad** is false.

Using pseudo-code, Listing 4.2 shows the sequence of execution for the example code in Listing 3.1. The transform functions in Listing 3.1 primarily apply compile-time optimizations and create computation objects. They serve as syntactic sugar to wrap the real computation. All the transform functions should be optimized away by the C++ compiler. The region of the code that is executed at runtime is inside the consumer function. At runtime, the consumer function will execute the prepare function in the map computation object, which will then call the prepare function in its parent class. This will result in a sequence of prepare function calls from the top-level computation object (vector computation object) to the bottom-level computation object (map computation object) as shown. After that, the consumer function will iterate through all the elements in parallel. Within the parallel loop, the computation for each level of the computation object hierarchy will be inlined and executed.

Many other optimizations can be applied using the same technique. The optimizations that are used in the TIC framework include padding, transposition, tiling, fission, etc. Some of the optimizations will be discussed in the later sections in this chapter.

Listing 4.2: Pseudo-code for example code in Listing 3.1 that is executed at runtime

```
1 consumer_function {
2   # a chain of prepare function
3   prepare_vector_compute_obj();
4   prepare_stencil_compute_obj();
5   prepare_map_compute_obj();
6
7    parallel_for  element in map_computation_object:
8     reduce(+, stencil_elements)
9 }
```

4.2 Fusion

Fusion is the optimization that merges multiple loops into one. Fusion helps to reduce the number of memory accesses and the control overhead. Fusion is one of the main benefits obtained from an iterator-based system. In Thrust, fusion can be achieve by concatenating multiple **transform_iterator**s. Similar to Thrust, fusion also happens automatically when the computation is chained together. By default, all the computation will be fused and computed in the consumer function. However, sometimes fusion might not give the optimal performance due to limitation of resources. In the TIC framework, fission, which splits the loops, can be applied automatically using static optimizations. In this case, an integer representing the resource usage counter is added to the template. In the transform function, if the resource usage counter is larger than some threshold, it can insert a consumer function to save the intermediate result, create a new data structure computation object and reset the resource counter.

Another problem of iterator-based fusion is show in Figure 4.1a. Figure 4.1a is a data flow graph of the computation in the kernel and the problems happen when one generator is shared. In the iterator approach, Generator A will be called by both Generator B and Generator C, and the computation within the Generator A will be executed twice. This will be far from optimal if the shared computation is very intensive. Horizontal fusion can be used to solve the this problem. There is research [24] showing how this can be achieved using an intelligent compiler, but figuring out the shared generator is very hard without the data flow analysis. There are no easy ways to figure out the data flow from C++ templates, so currently, the TIC framework relies on users to supply the information by using a horizontal fusion class. By using static optimization, the output of the shared generator is written to a temporary storage if the computation in the generator is significant enough (shown in Figure 4.1b). The computation intensity is currently implemented as a **constexpr** function provided by the user in the **Functor** class which specifies the transform operator. If the user chooses to use lambda expression, then a fixed number will be picked and the fission decision might not be accurate. Future improvement is expected to add compiler support to estimate the computation intensity. The temporary storage is implemented as a simple software cache local to each computation unit (or as shared memory

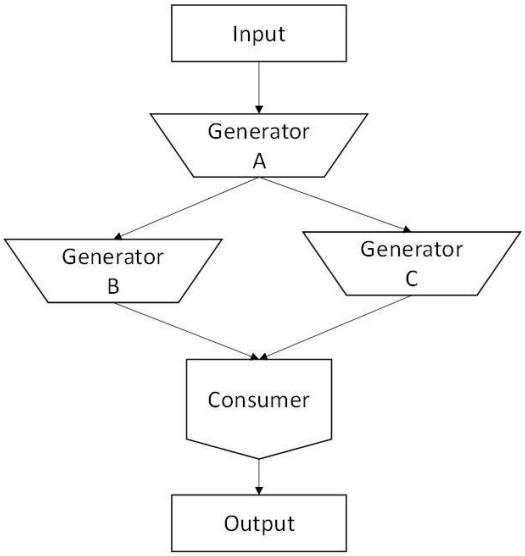
using CUDA terminology).

4.3 Tiling

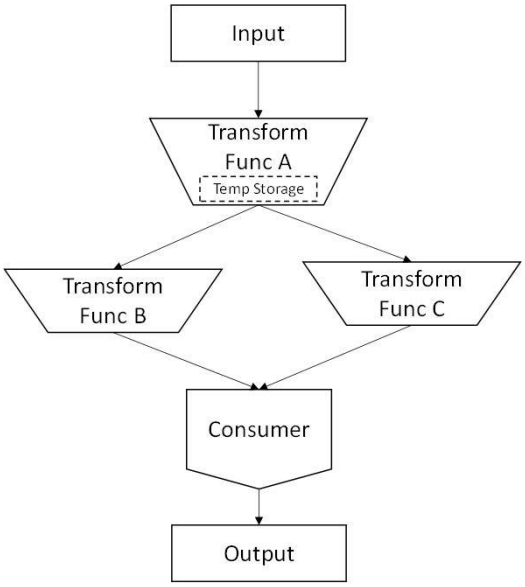
Tiling is the optimization that contains both loop strip-mining and loop reordering. Tiling will change the memory accesses order to improve the temporal locality. Tiling is another basic optimization available when the extended computation object requires tiling to improve the exploit input sharing (for example, a 2D matrix stencil). Tiling is by default disabled when constructing a 2D matrix because without proper data reuse in the cache, it only adds control overhead. When an extended computation object suggests input sharing is applied, the transform function will reconstruct the data structure computation object to enable tiling.

In the TIC framework, tiling is handled by the *data structure computation object*. A data structure computation object is the computation object output by the transform function applied directly to data structure. In the data structure computation object, multiple ways of iteration are implemented for each kind of traversal. For example, a 2D matrix can be traversed in following way: by elements, by rows and by columns. There are two versions of iteration implemented for each kind of traversal: tiled and non-tiled. The tiled iteration implementation also needs knowledge of the tile size as a parameter provided at compile time.

The iteration method is decided by the implicit access pattern provided by the combination of computation patterns. For example, a simple element traversal in 2D matrix does not require tiling. But if a stencil transformation is applied to the elements in the matrix, then it suggests boundary sharing and tiling is enabled. For the tiling size, it is selected using the same compile-time optimization technique based on the heuristic that if the kernel is memory-bound, the input accesses are tiled for the last level shared memory, else the input accesses are tiled to the private cache size. The classification of the kernel can be derived from the ratio between memory accesses and computation. In the framework, all the memory access to the data structure is abstracted away from the user so the expected memory accesses to the DRAM can be acquired from the extended computation objects like stencil objects. The computation intensity has to be defined the same way as



(a) Problem of iterator-based fusion



(b) Fusion in the TIC framework

Figure 4.1: Fusion optimization. (a) The traditional iterator-based fusion; Generator A will be evaluated twice in both B and C. (b) The fusion in the TIC framework uses a temporary storage to store the output of Generator A to avoid re-computation.

described in the previous section. Comparing the computation and memory access ratio with knowledge of the hardware, we can classify the kernel to be either memory-bound or computation-bound.

4.4 Vectorization

The vectorization in the TIC framework is achieved through the specification of the extended computation objects. Currently, the vectorization only supports **int** and **float** and these scalar types are converted to vector types through Intel vector intrinsics. Computations are converted to vector operations by overloading operators with the vector intrinsics.

An alternative approach will be using GCC or Clang OpenCL vector extensions. Unfortunately, the current vector extension lacks the ability to do type conversion. GCC vector extension will return a compilation error if an int vector is added with a float vector while Clang OpenCL vector extension will silently do a bitcast. None of the behavior matches the behavior of adding an int type to a float in C++ and neither extension provides a manual way of type conversion. With Intel vector intrinsics, the add operator of an int vector and a float vector is overloaded with a conversion intrinsics followed by an arithmetic intrinsics. Note the default behavior of converting float to int is rounded to the nearest value. To comply with the behavior of C++, the rounding mode needs to be changed to ROUND_TOWARD_ZERO by setting the corresponding bits in vector CSR.

In case of branch divergence in the vector operations, the framework uses multi-tier dynamic vectorization. Currently, only static branches can be vectorized. The static branches are the branches captured in the computation object like boundary checks. The static branches are not related to the data stored in the memory. For those static branches, convergence check is performed first. If it converges, the vector version is executed. Otherwise, the scalar version is executed.

4.5 Parallelization

The parallelization of the code in the TIC framework is achieved through Intel Thread Building Block (TBB) because TBB has a similar language interface (using functors and lambda functions) and TBB contains a set of runtime optimizations which is not considered in the TIC framework. These dynamic optimizations are extremely helpful for the load-balancing problem which is difficult to solve at compile time. In the consumer functions, the computation object will be wrapped into a functor and fed into either **parallel_for** or **parallel_reduce**.

In the TIC framework, the granularity of parallelism is also tuned using template-based static optimization. If the kernel is tiled, the granularity equals the tile size. If the kernel is not tiled, a bigger granularity is selected if no branch is present. If there are branches, smaller granularity is used to allow dynamic load balancing.

CHAPTER 5

RESULTS

In this chapter, four benchmarks are discussed together with their performance and programmability. The four benchmarks are scan, convolution, matrix multiplication and histogram equalization. The benchmarks are picked to reflect different aspects in the programming framework, and the details are shown in the following sections.

The performance comparsion is done betweem the code written in C, OpenCL and the TIC framework. There are two versions of C code compared in this chapter, a naive C implementation and an optimized C implementation. Clang is used to compile the C code as well as the code written in the TIC framework. For OpenCL implementation, the fastest existing kernels on the CPU are selected. The OpenCL kernels are compiled with both a commerical compiler (Intel) and a research compiler (MxPA) [25].

For all the benchmarks, the performance is measured on a quad-core CPU (Intel Core i7-3820) running at 3.60 GHz. Other hardware specifications and environment setups are specified in Table 5.1.

Table 5.1: The hardware specification and the environment setup

CPU	Intel(R) Core(TM) i7-3820 CPU
Frequency	3.60 GHz
Number of Cores	4
Number of Threads	8
Compiler	Clang 3.5
Optimization Level	3
OS	Debian 3.13.7-1
TBB version	2.2 Update 3
MKL version	10.2 Update 7

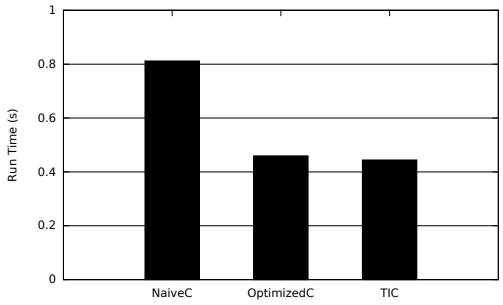
5.1 Scan

Scan is a basic computation pattern available in the TIC framework. It is one of the computation patterns that is provided in many programming languages and frameworks but believed to be very hard to be parallelized. This benchmark is picked to demonstrate the performance of the TIC framework in a small kernel with more communication and synchronization than computation. On the other hand, a naive C implementation has a decent performance due to its good memory access pattern and the minimal number of computations. In the TIC framework, vector memory access and tiling are used to further improve the DRAM bandwidth usage.

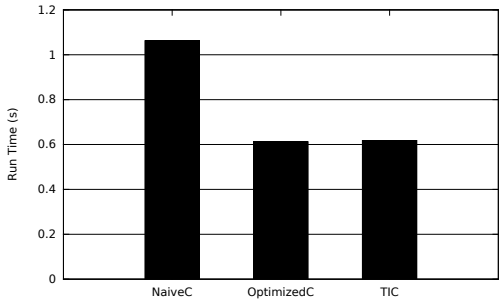
As described in Section 2.2.3, there are two implementations of scan. The speedup of the scan with a simple summation operator (prefix sum) is shown in Figure 5.1a. In this case, since the scan operator is cheap in terms of computation and the input of the scan is directly from the data structure, the reduce-scan-scan version is used. The results show that prefix sum can achieve 1.76x speedup on a quad-core machine compared to a naive version. The second benchmark shows a scan on a map computation object. In this case, the scan-scam-map implementation is used to avoid the re-computation in the map computation object. The speedup of scan-scan-map is shown in Figure 5.1b. The result shows the for a one-dimension stencil plus a prefix sum achieves a 1.66x speedup. In both implementations, the code generated from the templates has the equivalent performance compared to a hand-optimized C version using Intel vector intrinsics and TBB library.

5.2 Convolution

The 2D convolution benchmark is one of benchmarks in the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) Project [26] from DARPA. The convolution kernel implemented by the TIC framework shows its ability of handling computation-bounded kernels and how it selects tiling size based on static information. Vectorization and parallelization are important for computation-bounded kernels. Tiling is crucial to the convolution performance as well, since there is boundary sharing for nearby elements. The 2D convolution kernel can be written in the TIC framework using a com-



(a) Reduce-scan-scan



(b) Scan-scan-map

Figure 5.1: The speedup of scan compared to C implementations. (a) The runtime of a reduce-scan-scan version written in naive C, optimized C and the TIC framework. (b) The runtime of a scan-scan-map version written in naive C, optmized C and the TIC framework.

bination of stencil, map and reduce. The kernel function is shown in Listing 5.1.

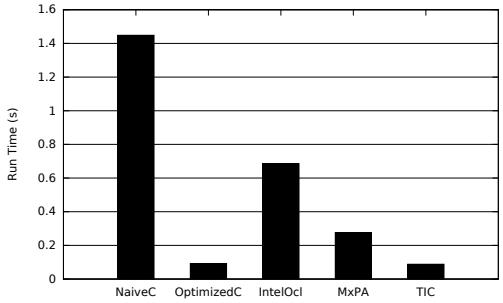
The results of convolution with different kernel sizes are shown in Figure 5.2. When scaling the convolution kernel size from 3x3 to 9x9, the benchmark turns from memory-bounded to computation-bounded, and different tile size decisions are made. Except for the 3x3 kernel size, the tiling size is half of the L1 capacity (due to hyper threading, private caches are shared by two threads) to minimized the latency. From the figures, the TIC framework achieves speedup from 16.4x to 21x. The speedup is coming from both better memory bandwidth utilization and computation resource utilization. The performance of convolution is also compared with OpenCL using both the Intel OpenCL compiler and MxPA. The TIC framework shows better performance than both OpenCL implementations. Compared to Intel OpenCL implementation, the TIC framework does better work scheduling and has a better memory access pattern. Compared to MxPA, the TIC framework generates better vector code than the CEAN extension used by MxPA.

Listing 5.1: 9x9 2D convolution user code

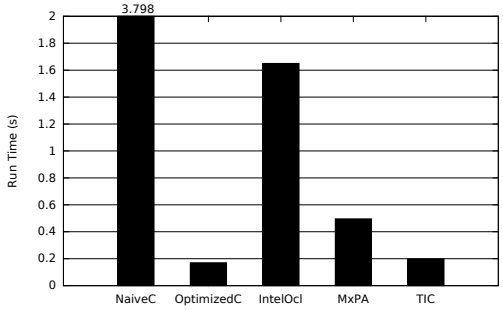
```
std::vector<int> 2dconvolution(const std::vector<int>& input,
        int x_size, int y_size,
        const std::vector<float>& kernel)
{
  using namespace TIC;
  // 2D matrix of int
  auto input_co = make_matrix<int, 2>(input, x_size, y_size);
  // expend to stencil (9x9)
  auto stencil_co = make_stencil_transform<9,9>(input_co);
  // init kernel matrix
  auto kernel_co = make_small_vector<int>(kernel);
  // extend the kernel_co to the size of matrix
  auto const_co = make_const_transform(kernel_co, x_size * y_size);
  // zip two kernel and stencil together
  auto zip_co = make_zip_transform(stencil_co, const_co);
  // multiple kernel with stencil
  auto map_co = make_map_transform(zip_co, map_operation<mul_op>());
  // reduce map result to a single value
  auto reduce_co = make_map_transform(map_co, reduce_operation<add_op>());
  // consumer function
  std::vector<int> output = Evaluate<std::vector, int>(reduce_co);
}
```

5.3 Matrix Multiplication

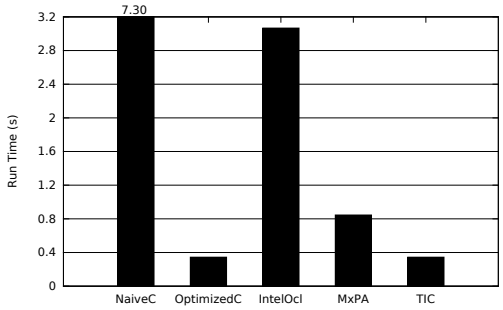
Matrix multiplication is a key benchmark to measure the peak performance of the system. The matrix multiplication can be done in the TIC framework using row and col traversal plus permutation. The implementation of a basic matrix multiplication is shown in Listing 5.2.



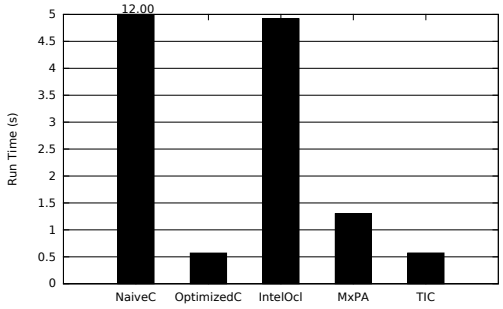
(a) 3x3 kernel



(b) 5x5 kernel



(c) 7x7 kernel



(d) 9x9 kernel

Figure 5.2: The speedup of convolution compared to C implementations (input image size is 3180x2160). (a) Convolution with a 3x3 kernel. (b) Convolution with a 5x5 kernel. (c) Convolution with a 7x7 kernel. (d) Convolution with a 9x9 kernel.

Listing 5.2: Matrix multiplication user code

```
1  void mm(int m, int n, int k,
2          float * A, float* B, float *C)
3  {
4    using namespace TIC;
5    auto A_co = make_matrix<float, 2>(A, m, k);
6    auto B_co = make_matrix<float, 2>(B, k, n);
7
8    auto A_row = make_row_traversal(A);
9    auto B_col = make_col_traversal(B);
10   auto P = make_permutation(A_row, B_col);
11
12   auto map_co = make_map_transform(P, map_operation<mul_op>());
13   auto reduce_co = make_map_transform(map_co, reduce_operation<add_op>());
14
15   Evaluate<float*>(C, reduce_co);
16 }
```

For the performance comparison, a generalized matrix multiplication (SGEMM) is measured. The performance of SGEMM in the TIC framework is compared with that of C, OpenCL and Intel MKL library (Figure 5.3). Using the TIC framework, the SGEMM benchmark can achieve more than 300x speedup over the naive C version, which does very poorly in terms of cache performance and computation utilization. In the meantime, the TIC framework achieves better performance than the OpenCL version using Intel OpenCL compiler and similar performance compared to that of MxPA. However, the TIC framework is 6x slower than Intel MKL. The reason why the TIC framework cannot achieve the peak performance in SGEMM is because the TIC framework is only capable of high-level optimization. For example, memory tiling is considered in the TIC framework but not register tiling. In order to achieve reliable register tiling, either assembly generation or compiler backend support needs to be added into the framework.

5.4 Histogram Equalization

Histogram Equalization is another benchmark from the PERFECT project. Histogram Equalization is a benchmark composed of three stages. The kernel first takes the image input and does a histogram from the input pixels. Then a

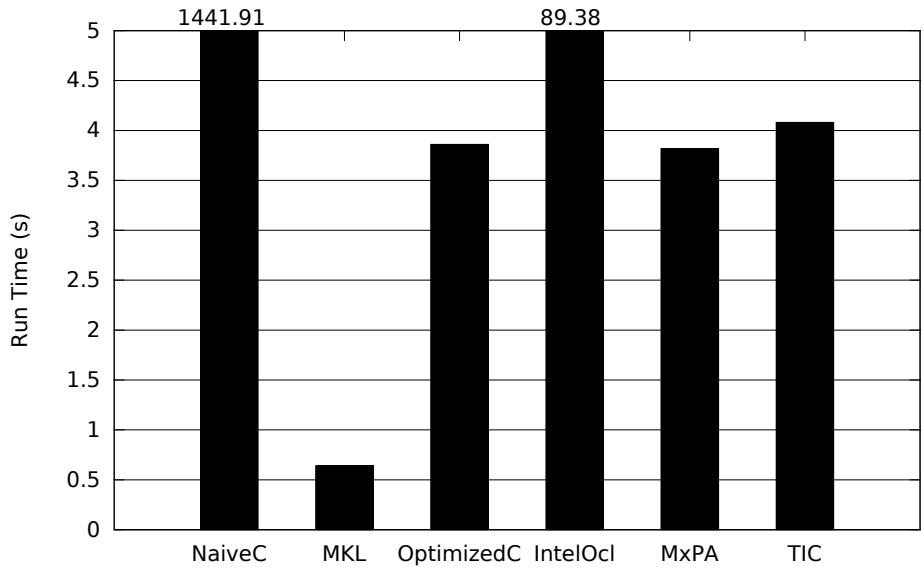


Figure 5.3: The performance of SGEMM in C, OpenCL and the TIC framework.

scan and a reduce is performed on the histogram output to generate a lookup table. Finally, the output is generated from the input using the lookup table from the previous stages. Two observations can be made from this specific benchmark. First, there are full dependencies across three stages. The stage cannot start till all previous stages are finished completely. Second, the level of parallelism changes from stage to stage. In Stage 1 and 3, there is parallelism across all the pixels while the Stage 2 has parallelism across all the histogram bins. These two features from the benchmark makes it interesting and challenging. Barriers need to be automatically generated across stages, and the work needs to be redistributed.

The performance of histogram equalization is shown in Figure 5.4. The kernel written in the TIC framework achieves 3x speedup compared to that of naive C. Through detailed analysis, the runtime of this benchmark is dominated by Stage 1 and Stage 3. In Stage 1, the performance improvement is coming from a build-in parallel histogram using privatization. In Stage 3, the better performance is from the parallel vector load.

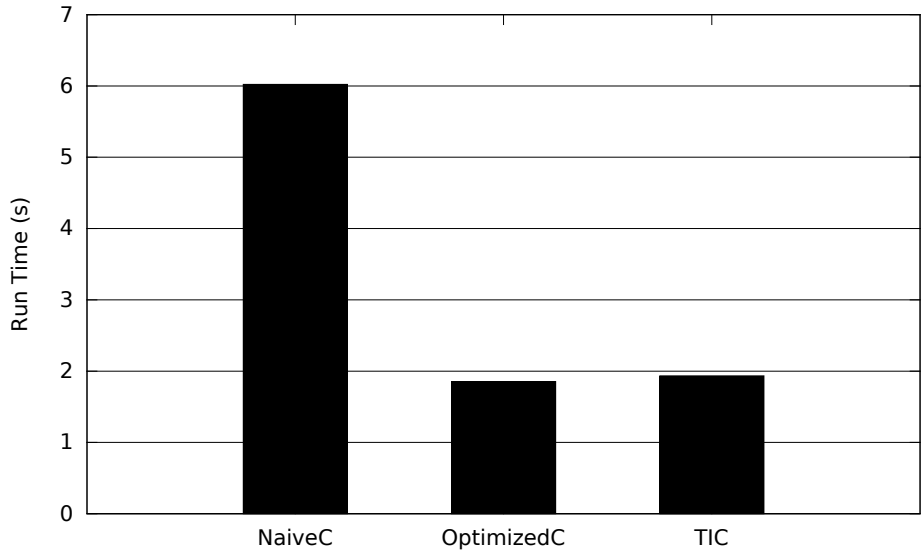


Figure 5.4: The performance of histogram equalization in naive C, optimized C and the TIC framework.

5.5 Programmability

In terms of programmability, the TIC framework also shows an advantage over both C and OpenCL. Table 5.2 shows the size of the kernel in the benchmarks written in different languages and frameworks. The size is measured by counting the lines of code excluding the write space. For the C version, only the computation code is counted. The variable declarations are not included because the baseline code is written in C89 standard, thus all variables are declared in the beginning of the functions and reused across multiple stages. For the OpenCL version, only the device code is included. The initialization and setup code on the host side is not included in the line count.

From the table, the kernel written in the TIC framework has an even smaller size than naive C implementation. On the other hand, the concise code in the TIC framework can achieve the performance of an optimized C version which is an order of magnitude longer. The results show that the TIC framework is the most efficient way to develop the high-performance kernel among the three choices.

Table 5.2: A comparison of the code length in different frameworks and languages

Framework & language	Scan	Map+Scan	Conv	SGEMM	HistoEq
Naive C	3	5	37	11	38
Optimized C	54	60	61	98	120
OpenCL	-	-	37	16	-
TIC	2	4	8	10	14

CHAPTER 6

FUTURE WORK

For future work, improvements can be made in the following directions:

- Auto-tuning should be supported. Currently all the optimization parameters are tuned and set manually. It is possible to set up a set of benchmarks to time the performance across different optimization decisions. The optimization parameters for the best performance will be stored for the static optimization.

- A source-level compiler can further improve the usability of the framework. Some information described in the previous chapters is hard to figure out only by types and templates. For example, the shared generator problem in Section 4.2 can be easily solved by a source-level flow analysis and a simple source-to-source transformation.

- More backend can be added to target the same source code to more parallel hardware. The next stage will be adding GPU support. The same computation object can be mapped to Nvida CUDA or Nvidia CUB to support GPUs. It will also be interesting to add support for heterogeneous computing.

CHAPTER 7

CONCLUSION

From the benchmark results, the TIC framework demonstrates that it is a feasible solution for programming a parallel machine. The TIC framework proves that:

- Common benchmark kernels can be constructed from our generalized computation patterns.
- Programmers can be highly productive in writing kernels with the TIC framework.
- Template-base static optimization can be used to switch high-level optimizations effectively.
- The kernels written in the TIC framework can achieve performance close to that of the hand-optimized C version.

However, there are limitations to the approach described in this thesis. Three of the most important limitations are:

- Lack of flow-analysis information from the C++ template limits the flow-sensitive optimization that can be automatically applied in the framework.
- Lack of control over low-level code generation causes the framework to fail to achieve peak performance on benchmarks which require clever register tiling.
- Lack of compiler support for automatic lambda expression lining might hurt the performance of more general computation benchmarks.

For benchmarks that do not require low-level optimizations and complicated operators, the kernels written in the TIC framework show a combination of high performance, good productivity and some potential of portability.

REFERENCES

[1] "Hadoop: Open source implementation of MapReduce." [Online]. Available: http://lucene.apache.org/hadoop/

[2] "The Phoenix system for MapReduce programming." [Online]. Available: http://csl.stanford.edu/~christos/sw/phoenix/

[3] J. Hoberock and N. Bell, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems, Jade Edition*, pp. 359–372, 2011.

[4] A. Munshi et al., "The OpenCL specification," *Khronos OpenCL Working Group*, vol. 1, pp. l1–15, 2009.

[5] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere, "An experimental study on performance portability of OpenCL kernels," in *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[6] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, and G. Newby, "Productivity of GPUs under different programming paradigms," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 2, pp. 179–191, 2012.

[7] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an embedded data parallel language," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 47–56.

[8] C. Rodrigues, T. Jablin, A. Dakkak, and W.-M. Hwu, "Triolet: A programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2014, pp. 247–258.

[9] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*. Springer, 1996, pp. 79–103.

[10] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *Compiler Construction*. Springer, 2010, pp. 244–263.

[11] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 51–61.

[12] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the automatic parallelization of the Perfect Benchmarks (R)," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 1, pp. 5–23, 1998.

[13] D. Novillo, "Openmp and automatic parallelization in GCC," in *Proceedings of the GCC Developers*, 2006.

[14] J. Reinders, *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[15] G. Mainland, R. Leshchinskiy, and S. Peyton Jones, "Exploiting vector instructions with generalized stream fusion," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2013, pp. 37–48.

[16] H.-S. Kim, I. El Hajj, J. A. Stratton, and W.-M. W. Hwu, "Multi-tier dynamic vectorization for translating GPU optimizations into CPU performance," *Center for Reliable and High-Performance Computing*, 2014.

[17] M. Garland, S. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing in CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[18] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," NVIDIA, Tech. Rep., 2008.

[19] N. Zhang, "A novel parallel scan for multicore processors and its application in sparse matrix-vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 3, pp. 397–404, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2011.174

[20] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[21] T. L. Veldhuizen, "Arrays in Blitz++," in *Computing in Object-Oriented Parallel Environments*. Springer, 1998, pp. 223–230.

[22] J. G. Siek and A. Lumsdaine, "The matrix template library: A generic programming approach to high performance numerical linear algebra," in *Computing in Object-Oriented Parallel Environments*. Springer, 1998, pp. 59–70.

[23] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Pearson Education, 2004.

[24] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional GPU programs," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ACM, 2013, pp. 49–60.

[25] J. A. Stratton, S. S. Stone, and W. H. Wen-mei, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 16–30.

[26] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013.