A Template Approach to transform Sources for Multicore Architectures

Xin Liu

Abstract—**In advent of multicore era, plain C/C++ programming language can not fully reflects the hardware artchitecture any more. A source-to-source compilation assists in adapting programs close to contemporary multicore hardwares. We proposed a template-based approach to perform the transformation for programs with rich static information. We presented template metaprogramming to conduct parallelization and memory hierarchical optimization. It enables programmers to adapte new architectural feature or parallel computation models by extending template library. In this paper, we implemented a prototype template library – libvina to demonstrate the idea. Finally, We evaluate the performance on commodity x86 and GPU platforms by a variaty of typical applications in multimedia and scientific fields. The experiments show that our approach is flexible to support multiple computational models. In addition, the experimental results reveal that our approach incurs little runtime overhead because it takes effects in compile-time.**

Keywords-**static analysis; Compiler optimization; parallelization**

I. Introduction

In multicore period, hardware architects rely on parallelism and memory hierarchy to enhance performance. Both cloned processors and elaborated storage-on-chip require programmer to restructure their source code and keep tuning binaries for a specific target. Therefore, writing a high-performance application requires non-trivial knowledge of underlying machine's architecture. The gap between hardware vendors and software developers extends development cycle, which increases marketing cost and risks for innovative multicore architectures in silicon industry.

Algorithm experts usually focus on their specific domains and have limited insights on diverging computer systems. They expect hardware and optimized compiler to guarantee decent performance for their programs. The expectation was roughly held until parallel system was introduced to computer community. Since frequency of general purpose processor stops growing faster, it has been hard to obtain free performance enhancement from hardware's refinement any more. Essentially, plain C/C++ can not fully reflect contemplate architecture such as multicore and distributed storage-on-chip. Researchers have admitted that it is tremendously challenging for optimizing sequential code by a compiler, so the answer to bridge programmer to parallel hardware relies on language and library to express concurrency richer than ever.

Designing new parallel programming is possible. Many functional languages [13] with inherent concurrency supports have emerged to the horizon of computer. However, a conservative programmer may turn them down because there are still lack of convincing evidences to demonstrate programmability and efficiency comparing to traditional programming languages. Another reason is software cost. Considering the time span which a large computer system serves, hardwares are cheap and become cheaper with time passing; software and well-trained personnel are expensive. Even numerous legacy systems designed with little consideration of multi-threaded environment at that time, vendors usually prefer to maintain and update legacy software systems for current and future hardwares rather than rebuilding them from scratch.

One side, software developers insist on classic programming diagrams and are reluctant to rewrite existing sources. On the other side, exploiting horsepower of modern processors for existing and new systems is a moderate issue. Therefore, it is desirable to extend traditional programming languages to balance the trade-off.

Many programming languages extending popular programming languages had been proposed for multicore. However, most of existing solutions aimed at specific architectures or computation models. UPC and OpenMP are designed for shared memory system [15]. CUDA works on vender-dependent GPU architecture for streaming computation; Sequoia [1] is an attempt to customize code-generation rules by XML configuration, however, it follows the similar restriction by enforce programs execute on a tree of abstract machines.

Our approach performs source-to-source transformation by compiler like Sequoia and OpenMP. We shared the same idea of [1] to generate a cluster of subprocedures for a task recursively. Instead of modifying compiler and introducing new language constructs, we exploit the capability of C++ template mechanism to achieve translation. All transformation rules are programmed in C++ meta-programming [10] and are conducted by a group of template class when that are instantiated. The primary limitation is that only type and static constant value are available in compile time. Therefore, it only works for programs which own rich static information. Fortunately, applications with this characteristic are pervasive in multimedia, digital processing, and scientific computations

Because libvina takes effect in compile time, it is possible to avoid from deploying runtime system, which means that it can incur minimal runtime cost. Besides that, our template-based approach imposes few restricts comparing with other static approaches:

- Thread-independent: It is possible to generate all kind of threads for target, including pthread, native LWP provided by OS, even vendor-dependent thread.
- Execution-independent: Our approach does not bind to any execution model. Utilizing template transform, we can change program into SPMD threads to exploit parallelism from multicore; chain many subroutines to build a pipeline to hide latency of memory; and separate program into blocks to improve locality.
- Extendability: It is flexible to develop new template class to utilize new architectural features. Template meta-programming is intimate for C++ programmers. It is easy to extend new execution rules and parallel patterns. Other approaches have to ratify languages and then modify compiler to complete features.
- Portability: Template mechanism are standardized in ISO [8], [17]. Our template library conforms to standards and is guaranteed to be portable for all mainstreaming C++ compilers.

The remaining parts of this paper are structured as follows. Section 2 summarizes some related works on static transformation and other library-based solutions. Section 3 show techniques to perform transforms by template metaprogramming. Audiences with C++ template programming experiences or functional programming concepts are helpful but not prerequisites. Then Section 4 presents some typical transforms by our template library. Experiments are in Section 5 to evaluate effect of specialization. Final section is conclusion and future works.

II. Related work

As mentioned before, it is desirable to extend conventional programming languages to reflects the essence of newer hardware. Researches in the field have two major directions:

1) providing new library to support programming in concurrency
2) extending language constructs to extend parallel semantics

First, library is a common method to extend language capability without introducing new language constructs or modifying grammar. POSIX thread library is a de facto standard to utilize multi-thread for applications on UNIX-compatible systems. The relationship between pthread and native thread provided by OS is straightforward. Therefore, abstraction of pthread is far away from naturally expressing parallelism and concurrency. Furthermore, the implementation of thread on hardware is unspecific in the standard, so it can not guarantee performance or even correctness on some architectures [4], [5].

C++ community intends to develop parallel library while bearing generic programming in mind. TBB(Thread Block Builder) has a plenty of containers and execution rules. Entities including partitioner and scheduler in TBB are created in runtime. In that case, key data structures have to be thread-safe. Although TBB exploits task parallelism or other sophisticated concurrency on general purpose processors, the runtime overhead is relative high in data parallel programs, especially in the scenario that massive ALUs are exposed by hardware. Solution we proposed here is orthogonal to runtime parallel libraries. We only explore parallelism that can be determined in compile time, developers feel free to deploy other ways to speedup the programs in runtime, such as TBB.

Second choice for language communities is to extend semantics by modifying compiler. They add directive or annotation to help compiler transform source code. OpenMP is designed for shared memory and has shipped in almost every C/Fortran compilers. OMP compilers transform sequential code into multi-threaded programs with OMP runtime. Although OpenMP is simple and portable, the performance is not optimal in most cases. A handful of directives leave little room for further enhancement and scaling up to larger systems. Hybrid OpenMP with MPI is possible, but difficulty surges.

Another approach is to add new language constructs. Sequoia is a source-to-source compiler to support programming memory hierarchy. First of all, It targets execution environment as a tree of machines, which an individual machine owns its storage and computation unit. Second, It terms a computation-intensive function as *task*. New constructs apply on *task*. Sequoia compiler transforms a *task* into a cluster of subprocedures based on machines hierarchy. The target machines are described in XML files. Based on this idea, Sequoia actually can specialize task for target architectures. [2] reports Sequoia can successfully transform codes for CellBE, SMP, cluster while keeping very competitive performance. However, Sequoia as a language extension is restrictive. The basic data structure is array, which obviously shows the preference of scientific computation. The language constructs do not cover all the common parallel patterns such as pipeline and task queue. Libvina derived the idea of Sequoia to transform source code recursively. However, template meta-programming utilizes existing mechanism in C++ compiler and is capable to express all the semantics in Sequoia programming language. Moreover, Sequoia can not leverage type system to specialize code. *e.g.* Many modern processors usually provide SIMD instructions. Libvina could generate corresponding source based on predefined vector types. Sequoia compiler ignores this important information and simply relies on native compilers.

III. A Template library

A. Overview

We implemented a template library, libvina, to perform source specialization. One fundation of our approach is that assume C++ compiler front-end as a code generator. It actually practices source-to-source transforms in the guidance of template meta-programming. Defintely, to ulitize static information, we provides a couple of data structures with template parameters to carry such information. When template classes are instantiated , compiler recursively generates codes until specific conditions are statisfied.

In our design philosophy, programmers only care about developing efficient algorithms. Template library takes responsibility for parallelization and optimizstion of memory while keeping the same interface. Porting to another platforms might need to refine template parameters or apply to new templates to obtain maximal performance, however, it is unnecessary for programmers to modify even know details of algorithms.

B. Prerequisites

Originally, C++ template mechanism is invented to supersede C preprocessor. It is type-safe and could facility generic programming. People found the potential of template computation by chance. [6] later proved template itself is Turing completeness. Beside the job it meant to do, template has successfully applied to many innovative purposes in modern C++ programming practices [9].

A powerful feature of C++'s templates is template specialization. This allows alternative implementations to be provided based on certain characteristics of the parameterized type that is being instantiated. Template specialization has two purposes: to allow certain forms of optimization, and to reduce code bloat [18].

Template meta-programming is similar to functional programming language except it takes effect in compile time. It only relies on static information to determine control flow and perform computation. MPL Library [16] provides control statement and STL-like data structures, which greatly eases programming in static realm.

C. Concepts

1) TF class: Computation-intensive functions are commonly referred to *kernel* or *filter*. Mathematically, a function is single-target binary relation. Kernel functions are usually self-contained, *i.e.* external data references are limited and calling graphs are simple. It's possible for a kernel function to decouple into a group of subprocedures. Each subprocedure may be exactly the same as kernel and spread on multicore running in parallel. Another approach is to divide a kernel into finer stages and run in streaming manner to respect data locality and bandwidth. In libvina, a *transform class (TF class)* is a template class which transforms a function to a cluster of subprocedures in isomorphism. As shown in 1, the transformed function on right side has the same interface while owns a call graph to complete the original computation by a cluster of subprocedures. Execution of the call graph can be programmed by in the library to scrutinise target's architecture.

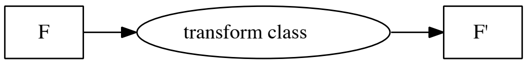

Figure 1. transform class

2) Polymorphism in compile-time: In programming language, a function which can applies any values of differnt types are parameteric polymophism. C++ has already supportted this language feature by template function. Our library need to manipulate template functions and instantiate them on demand, which we call it *late-instantiation* inspired of *late-binding*. However, the entry address of a template function is not available until it is instantiated. Therefore, it is desirable to extend function polymorphism in compile-time. Our approach is to wrap the template function by a template class and pass class as template template class. This is the only way to implement late-instantiation. It incurs an extra function call and will be hopefully eliminated by compiler's optimization.

```
template<class Result, class Arg0, class Arg1>
struct vecAddWrapper {
 //...
 static void
 doit(const Arg0& arg0, const Arg1& arg1,
     Result& result)
 {
    vecArithImpl<T, DIM_N>::add (arg0, arg1, resul
 }
};
```

3) Predicate: Borrowed from lisp concept, *predicate* represents a indicator of some conditions. In libvina, it is a template class with static fields initialized by constant expressions consisting of template parameters and constants. These fields are automatically evaluated when template classes are instantiated. **??** is an example to determine whether the problem size is fitting to last level cache.

```
template <class T, int SIZE_A
     , int SIZE_B, int SIZE_C>
struct p_lt_cache_ll {
 enum {CACHE_LL_SIZE = 4096*1024};
 const static bool value =
     ((SIZE_A * SIZE_B
   + SIZE_A * SIZE_C + SIZE_B * SIZE_C)
   * sizeof(T) ) <= CACHE_LL_SIZE;
};
```

4) Sentinel: Sentinels in libvina are non-type template parameters of *TF class*, with a *predicate* as default initializer.

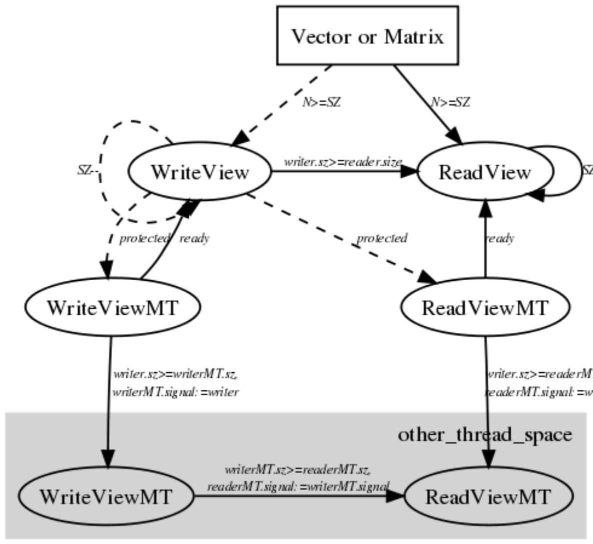

Figure 2. View

When a template class is instantiating, sentinels are evaluated. A *predicate* determines whether a specific requirement has been satisfied. Sentinel is responsible for changing generation strategy according to the result. Using *template specialization*, C++ compiler chooses different versions of a class to instantiate basing on the values or types of template arguments. The most important application of *sentinel* is terminate code generation. More general flow control such as branch is feassible in [**?**].

D. Supporting data structure

Template meta-programing can only manipulate static information. As a result, ADTs in libvina need to carry such information as template parameters. Only Vector and Matrix are implemented in our library, because they cover a wide spectrum of application in multimedia area. Users require more versatile data structures could resort to mature library such as [10]. As 2 depicted, View is a concept to abstract data set. It is not neccessary to duplicate concrete data in shared memory. View classes are divisible and type-safe.

Shadow region is the other thread space. The only approach to communicate with other thread is through ViewMT. We only provide synchronization for ViewMT to encourage bulk operations. ViewMTs can copy in non-blocking, however inducing from ViewMT to normal View is a blocking operation. By this approach, libvina guarantees synchronization by internal signal. We defined a signal class with semphore sematics. It is copyable to support transmitting data set to multiple receivers. In addition, many contemporary multicore architectures [20] have explicit facilities to perform such kind of operation so we can implement it effectively.

IV. Program specialization

A. SPMD

Multi-threading is dominant approach to utilize duplicated computation units. SPMD model is the most intuitive manner to describe parallelism. Furthermore, high performance of SPMD is fundation of streaming model. There are numerous kernel functions in multimedia applications and scientific computations which can easily exploit data parallelism by dividing task into smaller and independent subtasks. This feature naturally matches libvina's transformation. We implemented *mappar* and *mapreduce* language constructs in [1] as template classes. aux::subview is a meta-functions to deal with cutting off data set. It could return a real subview or itself according to types of template parameters. IV-A gives the definition of arg1_isomoph, which determintes whether the *arg1* is isomorphic.

IV-A is the definition of *mappar TF* class. Template parameter *Instance* is computation task containing a kernel function wrapper and arguments. The last template parameter *__SENTINEL__* determines control flow when instantiation occurs. **??** is a template partial specialization to generate concrete thread to practice computation. It is noteworthy that the last two arguments are *true*, which means that this class is multi-threaded and leaf node version.

```
template <class Instance, int  _K,
  bool _IsMT,  bool __SENTINEL__>
struct mappar {
  typedef mappar<typename Instance::SubTask,
        _K, _IsMT,
Instance::SubTask::_pred> _Tail;

  typedef typename mpl::or_<mpl::bool_<std::tr1::i
    mpl::bool_<std::tr1::is_same<typename Instance
 typename Instance::SubTask::Arg1>
 ::value>
      >::type
  arg1_isomorph;
  //...
  static void doit(const typename Instance::Arg0&
   const typename Instance::Arg1& arg1,
     typename Instance::Result& result)
  {
     for (int k=0; k < _K; ++k) {
auto subArg0   = aux::subview<typename Instance::A
    arg0_dim::value, arg0_isomorph::value>::sub_re

auto subArg1   = aux::subview<typename Instance::A
    arg1_dim::value, arg1_isomorph::value>::sub_re

auto subResult = aux::subview<typename Instance::R
    ret_dim::value, ret_isomorph::value>::sub_writ
```

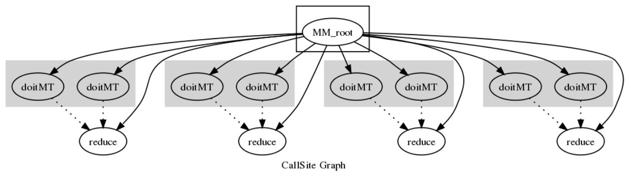

Figure 3. MM internal call graph

```
_Tail::doit(*subArg0, *subArg1, *subResult);
    }//end for
 }
};
```

```
template <class Instance, int  _K>
struct mappar <Instance, _K, true, true>
{
// ...
 static void
 Doit(const typename Instance::Arg0& arg0,
   const typename Instance::Arg1& arg1,
   typename Instance::Result& result,
   mt::barrier_t barrier)
 {
   auto compF = Instance::computationMT();

   mt::thread_t leaf(compF, arg0, arg1,
     __aux::ref(result, result_arithm()),
     barrier);
 }
};
```

3 is a cluster of subprocedures generated by libvina after applying *mapreduce* to a matrix multiplication function. Template parameter _K is 2 in this case because we perform this transform for dual core machine. Libvina *mapreduce* divides a matrix into 4 sub-matrices. The figure except dashed lines is actually a call graph. Shadow lines is multi-threaded environment and two subprocedures are executed in parallel. Dashed lines indicate logical synchronization. We implemented it by semphore or higher level barrier based on different thread libraries.

mapseq in [1] can be trivially implemented by passing false to *_IsMT* parameter. Nested block is possible by recursively defining *TF classes*.

B. Streaming and pipelining

Streaming computation is a computer paradigm to perform massive parallelled computation. It models data set as a *stream*. Operations are usually orangized in pipeline way to process in turn, while keeping stream in on-chip storage. It can utlize duplicated ALUs array to perform computation in parallel and reduce external bandwidth.

More general streaming computation does not restrict to keep data stationary. Pipeline processing inherently support heterogeneous architecture or ring network [14], [19]. If specific processors are exposed by platform and communition cost is manageable, developers intend to ultize them for throughput or energy advantages. Our template approach has no problem to link external computation as long developers provides communication layers.

Our template library provides two components to support streaming compuation. First of all, we provides a multi-threated *ViewMT* classes depicted in 2. Accessing elements from ViewMT is blocking and that thread is supposed to sleep until concerned event arrives. *e.g.* upstreaming thread completes job and release the ownership, this event will wake up sleeping threads and proceeding with data. On shared memory system, we implemented it by plain pointer and semphore. For external accelerator, we implemented by memory map and event provided by OpenCL [12]. Secondly, we provide a TF class to build a pipeline. The simplifed class is enlisted as follows **??**. The class chains a serie of stages. It is noteworthy that we dedicatedly undefine the end of recursion because framework has no idea how to deal with the output of pipeline. It is user's responsibility to add final stage to clarify behaviors.

graph to depict multithreaded structure on x86.

```
template <typename... Stages>
struct pipeline;

template <class P, typename... Tail>
struct pipeline<P, Tail...> {
  typedef typename P::input_type in_t;
  typedef typename P::output_type out_t;

 static out_t doit(in_t in)
  {
     pipeline<Tail...>::doit(  P::doit(in)  );
  }
};
```

C. Blocking

D. Cooperation with other libraries

We implemented our library in ISO standard C++. Theoretically, any standard-compliance C++ compiler should process our classes without trouble. C++0X [17] added a lot of language features to ease template meta-programming. Compilers without C++0X supports need some workarounds to pass compilation. Consider the trend of C++, development of template libraries such as libvina should became easier and smoother in the future.

We implemented *mt::thread* based on underlying Linux PThread. A simple C++ thread pool is developed to reduce cost of thread creation. SPMD(Single program multiple data) is very important execution model. In order to obtain similiar SPMD execution model on x86, we designed a lightweight thread library(libSPMD) based on Linux clone(2) and semaphore. In order to obtain platform independence, we utilize GPU by OpenCL API. Many Accelerates are scheduled to implemented OpenCL, which might be extend our approach to new territories in the future.

V. Experiment and evaluation

A. Methodology

C++0X is partially supported by mainstreaming compilers. Currently, we developed the library for gcc 4.4.0. GPU performance was measured on Mac OSX 10.6.

A couple of algorithms are evaluated for libvina. They are typical programs in image processing and scientific computations. In addition, we implemented a language translation scenario to illustreate pipeline processing. On x86 platform, We linked Intel Math kernel library to perform BLAS functions. Other procedures are implemented on our own.

saxpy	Procedure in BLAS level 1. A scalar multiplies to a single precision vector, which contains 32 million elements.
sgemm	Procedure in BLAS level 3. Two 4096X4096 dense matrices multiply.
dot_prod	Two vectors perform dot production.
conv2d	2-Dimensional convolution operation on image.
lang_pipe	Pseudo-Multi-language translation. A word is translated from one language A to language B, and then new function will translate it from language B to language C, etc.

Three multicore systems are used to conduct experiments.

Table I
Table 1: Experimental machines

name	ISA	topology	frequency	storage-on-chip	bandwidth
htn	x86 harpertown	2-way quadcores	2.00Ghz	6M shared L2 Cache X2	
nhm	x86 nehalem	quad cores	2.93Ghz	256K L2 8M shared L3	
9400m	gpu g80	2 MultiProcessors	1.10Ghz	16k local storage	3.4Gb/s

B. Results

Figures are preparing...

1. speedup on x86
2. speedup on GPU.
3. table: peak performance for x86 and gpu
4. comparison 3 SPMD threads.
5. comparison between gpu and cpu. matrix-multiplications?

VI. Conclusions and Future work

References

[1] Fatahalian, K., Knight, T., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming The Memory Hierarchy. SC2006

[2] Knight, T. J., Park, J. Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W. J., and Hanrahan, P. Compilation for Explicitly Managed Memory Hierarchies. PPoPP 2007

[3] Aho, A., Sethi, R., Ullman, J.D., Lam, M.: Compilers: Principles, Techniques, and Tools. Addison-Wesley.

[4] Boehm, H.J.: Threads Cannot Be Implemented As a Library. PLDI '05

[5] Drepper, U., Molnar, I.: The Native POSIX Thread Library for Linux. 2005

[6] Veldhuizen, T. L.: C++ Template are Turing Complete.

[7] Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., Mendelson, A.: Programming Model for Heterogeneous x86 Platform. PLDI '09.

[8] C++ Standard Committee: ISO/IEC 14882:2003(E) Programming Languages C++, 2003

[9] Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied, 2001

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

[11] Kapasi, U.J., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: The Imagine Stream Processor. Proceeding of the 2002 International Conference on Computer Design.

[12] Munshi, A., Khronos OpenCL Working Group, The OpenCL Specification ver.1.0.43, 2009

[13] Goldberg, B.: Functional Programming Language, ACM Computing Surveys, 1996

[14] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P. 2008. Larrabee: A ManyCore x86 Architecture for Visual Computing. ACM Trans. Graph. 27, 3, Article 18 (August 2008), 15 pages. DOI = 10.1145/1360612.1360617 http://doi.acm.org/10.1145/1360612.1360617.

[15] El-Ghazawi, T., Cantonnet, F., Yao, Y, Rajamony, R.: Developing an Optimized UPC Compiler for Future Architecture

[16] Gurtovoy, A., Abrahams, D.: The BOOST C++ Metaprogramming Library

[17] C++ Standard Committee: ISO/IEC DTR 19768 Doc No. 2857, 2009

[18] Stroustrup, B.: The C++ Programming Language (Spcial Edition) Addison Wesley. Reading Mass. USA. 2000. ISBN 0-201-70073-5

[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy: Introduction to the Cell Multiprocessor, IBM Journal of Research and Development 49, No. 4/5, 589-604, 2005

[20] Official OpenMP Specifications, OpenMP Architecture Review Board, 2002, http://www.openmp.org/specs/.