A Template Approach to Transform Programs in Static

Xin Liu†, Daqiang Zhang†, Jingyu Zhou†, Minyi Guo†
Department of Computer Science
Shanghai Jiao Tong University
Dongchuan 800, Shanghai, P.R.China
{*navyliu, zhangdq*}*@sjtu.edu.cn,* {*guo-my, zhou-jy*}*@cs.sjtu.edu.cn*

***Abstract*—In advent of multicore era, plain C/C++ programming language can not fully reflects the hardware architecture any more. A source-to-source compilation assists in adapting programs close to contemporary hardwares. We proposed a template-based approach to perform the transformation for programs with rich static information. We presented template meta-programming to conduct parallelization and memory hierarchical optimization. It enables programmers to adapt 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 variety 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 run-time 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 language 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 C/C++ 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 classes 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 computation.

Because template takes effect in compile time, it is possible to avoid 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:

- Meta-programming: We proposed a way to transform source code by meta-programming. Because it can manipulate source code in compile time, our approach does not bind any execution model. It is easy to change code to Fork-Join like *OpenMP*, or perform computation as pipeline. Furthermore, we can use all kind of thread implementations, including pthread, native LWP provided by OS, even hardware thread.
- 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 models 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]. Template-based approach is applicable for every platforms with standard C++ supports. Except for platform features, template libraries are portable .

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

II. Template Metaprogramming Approach

A. Overview

We implemented a template library – libvina, to perform source transformation. One foundation of our approach is that assume C++ compiler front-end as a code generator. It actually practices source-to-source transformation in the guidance of template meta-programming. As mentioned before, template approach is limited in compile time, therefore, problems we intend to solve must have rich static information. Fortunately, applications with this characteristic are not uncommon in multimedia or digital processing fields, some of them even attract developers to implement them in hardware such as ASIC, FPGA, or DSP. To leverage static information, we provides data structures with template parameters to carry such information.

In our design philosophy, we separate two roles in software development. Domain-specific experts only care about writing efficient algorithms in conventional C/C++ form. They provide interfaces in forms of wrapper classes of functions. On the other side, system-level engineers develop C++ templates to take responsibility for parallelization and memory optimization while keeping function's type same. Optimizations for specific architectues are achieved by applying a series of template classes. Porting to other platforms might need to refine template parameters or apply to new templates, however, programmers do not need to dig into algorithms again.

B. Background

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 as *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 pipeline 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 Fig. 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 take advantage of architectural features.

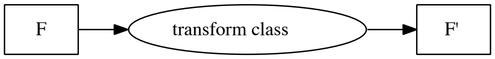

Figure 1. transform class

2) Function Interface: In programming language, a function which can applies any values of different types are parametric polymorphism. C++ has already supported 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*. Because the entry address of a template function is not available until instantiation, it is desirable to extend function polymorphism for different template argumenta at compile-time. Our approach is to wrap the template function by a template class and pass it as *template template class*. A wrapper function acts as *interface* provided by algorithm developers. Fig. 2 is a wrapper function of vector addition.

Beside it enables late-instantiation, the advantage of template class interface is to provide different entries for different execution environments. *doit_b* is used to implement synchronization in Fig. 6. Another benefit of this form is that wrapper classes give compiler a chance to select appropriate codes based on their types, which is T in our example. This method incurs an extra function call thought, it is 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,
                           result);
 }
 static void
 doit_b(const Arg0& arg0, const Arg1& arg1,
     Result& result, mt::barrier& barrier)
 {
   doit(arg0, arg1, result);
   barrier.wait();
 }

 //...
};
```

Figure 2. Function Wrapper

3) Predicate: Borrowed from lisp concept, *predicate* represents a indicator of some conditions. 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. Fig. 3 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;
};
```

Figure 3. Predicate

4) Sentinel: Sentinels in libvina are non-type template parameters of *TF class*, with a *predicate* as default initializer. 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 [16].

D. Supporting data structure

Template meta-programing can only manipulate static information. As a result, data structures need to carry such information as template parameters. Only vector and matrix are implemented in our implementation, because they cover many applications in multimedia and scientific fields. Users require more versatile data structures could resort to mature libraries such as [10].

View class is a concept to represent data set. Fig. 4 depicts relationship of Views. Concrete lines represent implicit conversion in C++, while dashed lines are explicit function calls to complete conversion. Text in edges are constraints when conversions perform. Shadow region is another thread space. The only approach to communicate with other thread is via ViewMT.

Modern multicore architectures emphasize on utilization of bandwidth and storage-on-chip, therefore we manipulate data in bulk way. Essentially, View is an abstract of *stream* and it helps programmers build streaming computation. Underneath of View class, we can perform optimizations based on architectures. *e.g*, it is not neccessary to duplicate data on shared memory, or asynchronous communication can be raised to hide latency. We define a signal class to synchronize, which mimic signal primitive on CellBE [19].

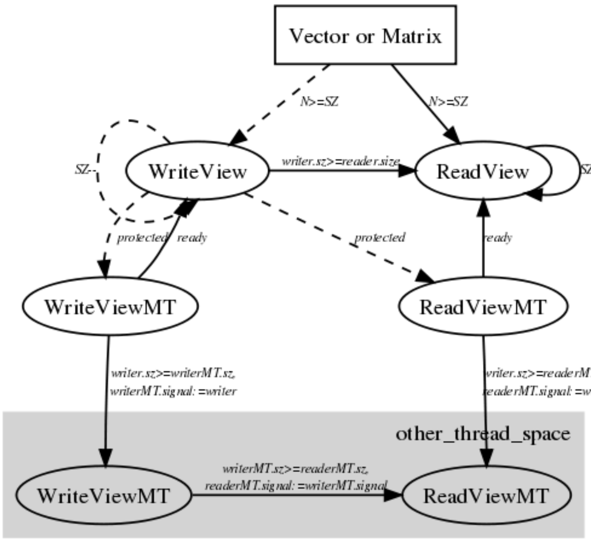

Figure 4. View concept in libvina

It is implemented by conditional variable on x86. For GPU, we use *event* of OpenCL to achieve the same semantics.

In addition, View class is type-safed. Programmers can get compilation errors if programs have potential violations of date access rules. Early errors are particularly important to prevent programmer from trapping into multi-threaded bugs.

III. Source Transformations by Template

A. SPMD

Multi-thread is dominant approach to utilize duplicated computational units. SPMD model is the most intuitive thread model. Moreover, SPMD is foundation of streaming computation. There are numerous kernel functions in multimedia applications and scientific computation which can exploit massive data parallelism by dividing task into smaller and independent subtasks. This characteristic can be naturally expressed by libvina's TF class. We implement *mappar* and *mapreduce* language constructs in [1] as template classes.

Fig. 5 is a definition of *mappar* TF class for x86. Template parameter *Instance* is computation task containing a kernel function wrapper and template arguments. The last template parameter *__SENTINEL__* determines control flow when instantiation occurs. The second class in the figure is a template partial specialization of prime template, which generates concrete thread to practice computation. It is noteworthy that the last two arguments are *true*s, which means that this class is multi-threaded and leaf node version. aux::subview is a *meta*-function to cut off Views. It could return a real subview or trivially return itself according to types of template parameters. Fig. 5 gives the definition of arg1_isomoph, which determintes whether the *arg1* is cutted off or keeps intact.

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

 typedef typename mpl::or_<mpl::bool_
   <std::tr1::is_arithmetic
     <typename Instance::Arg1>::value>
   ,mpl::bool_<
     std::tr1::is_same<typename Instance::Arg1,
              typename Instance::SubTask::Arg1>
      ::value>
   >::type
   arg1_isomorph;
 //...
 static void
 doit(const typename Instance::Arg0& arg0,
      const typename Instance::Arg1& arg1,
        typename Instance::Result& result)
 {
   for (int k=0; k < _K; ++k) {
    auto subArg0 = ...

    auto subArg1 = aux::subview<decltype(arg0),
        arg1_dim::value, arg1_isomorph::value>
            ::sub_reader(arg1, k);

    auto subResult = ...

    _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)
 {
   auto compF = Instance::computationMT();

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

Figure 5. TF class of *mappar*

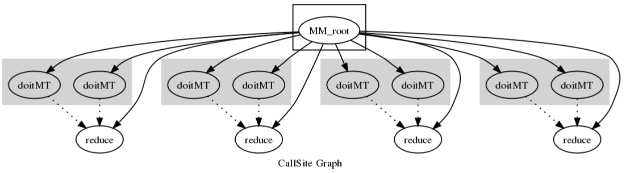

Figure 6. MM internal call graph

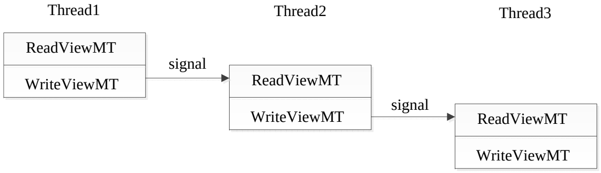

Figure 7. ViewMT in pipeling

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

Figure 8. TF class of pipeline

Fig. 6 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 intend to perform this transform for a dual core machine. *mapreduce* divides a matrix into 4 sub-matrices. The figure except dashed lines is actually a call graph. Shadow box is multi-threaded environment and two subprocedures are executed in parallel. Dashed lines indicate logical synchronization.

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 organized in pipeline way to process in turn, while keeping stream in on-chip storage. It can utilize multicore 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 and ring network [14], [19]. If specific processors are exposed by platform and communication cost is manageable, developers intend to leverage them for performance or energy advantages. Template approach has no problem to link external computational devices as long as developers provide communication layers.

Our template library provides two components to support streaming computation. First, we provides multi-threaded *ViewMT* classes depicted in Fig. 4. Fig. 7 is the scenario in pipeline processing. We leverage ViewMTs to build a chain among threads. Each stage takes charge of releasing the ownership of data set after it finishes job.

Second, we provide a TF class to build a pipeline. The simplified class is listed as follows Fig. 8. The class chains a series of stages. It is noteworthy that we dedicatedly 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 the behaviors.

C. Cooperation with other libraries

We implemented our library in ISO standard C++ [8]. 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 though, they do not hurt expressiveness. Consider the trend of C++, development of template library such as libvina should become 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. Because PThread does not have group-scheduling, we design and implement a lightweight thread library(libSPMD) based on Linux clone(2) and semaphore. For GPU,we use OpenCL to obtain platform independence. Many accelerators are scheduled or declared to implement OpenCL, which might extend our approach to new territories in the future. Although the interfaces of threads memtioned before are varying, our library can deal with them well.

IV. Experiment

A. Methodology

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

A couple of algorithms are evaluated for our template approach. They are typical algorithms in image processing and scientific field. In addition, we implemented a fake language translation scenario to illustreate pipeline processing. The programs in expriments are listed as follows:

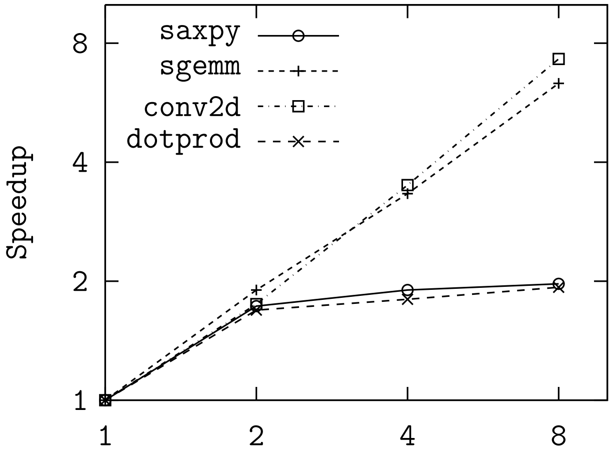

Figure 9. Speedup on Hapertown

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.

dotprod 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.

Two multicore platforms are used to conduct experiments. The hardsware envronments are summaried in the following table:

Table I
EXPERIMENTAL PLATFORMS

name	type	processors	OS
harpertown	SMP server	intel quad-core 2.0Ghz	Linux Fedora kernel 2.6.30
macbookpro	laptop	intel dual-core 2.63Ghz nvidia 9400m 1.1Ghz	Mac OSX Snowleopard

On harptown, we link Intel Math kernel to perform BLAS procedures except for conv2d. On macbookpro, we implemented the algorithms on our own.

B. Results

Fig. 9 shows the speedup on harpertown. HP Proliant blade server contains 2-way 8 cores x86 Xeon processors. We obversed good performance scalalbility for programs sgemm and conv2d. The first one use *mapreduce* pattern and the other uses *mappar*. Both of them are close to upper-limited speedup on target machines. dotprod and saxpy reveal low speedup because non-compute-intensive programs are subject to memory bandwidth. saxpy needs one load and one store for every two operations, and dotprod has similar situation. They quickly saturate memory bandwidth for SMP system.

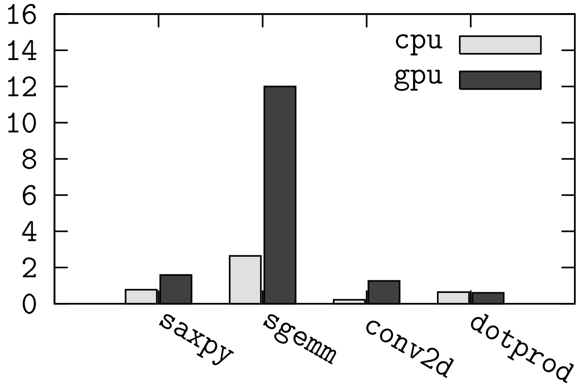

Figure 10. Speedup Comparing GPU with CPU

Fig. 10 shows transformation results for GPU on macbookpro. Programs run on host CPU in sequence, and embedded GPU on motherboard contains 2 SMs[1] Porting from CPU to GPU, developer only need a couple of lines to change templates and keep algorithm the same [2] As figure depicted, compute-intensive programs sgemm and conv2d still maintain their speedups. In addtion, we observed about 2 times performance improvement comparing with saxpy counterpart on CPU. GPU can map massive threads in group of warp(32 threads) on hardware and memory accesses are possible to be coalesed if a half of warps satisfy some specific access patterns. Memory coalesce mitigates bandwidth issue occurred on CPU. Because dotprod has fixed step to access memory, we can not obtain hardware optimization without tweaking the algorithm.

Table II
COMPARION OF SGEMM ON CPU AND GPU

	Seqential	CPU	GPU
cores	1	8	2 SMs
Gflops	2.64	95.6	12.0
utilization	12.6%	74.9%	68.2%
lines of function	63	unknown	21

Table. II details sgemm execution on CPU and GPU. Dense maxtrix multiplication is a typical computation-intensive program. Our template transforms a sequential sgemm both for CPU to embedded GPU. We choose sequential execution on macbookpro's CPU as baseline. After mapping the algorithm to GPU, we obtains over 4.5 times speedup compare with host CPU. Intel core 2 microprocessor can issue 2 SSE instructions per cycle, therefore, the

[1] Streaming Multiprocessor, each SM consists of 8 scalar processors(SP)

[2] Because GPU code needs special qualifiers, we did decorate kernel functions. We keep algorithms the same except for sgemm. It is not easy to work out sgemm for a laptop, we added blocking and SIMD instruments for CPU.

theoritical peak float performance is 21 Gflops. We obtain only 2.64 Gflops which utilization percentage is only 12.6% even develop quite complicated source. On the otherside, 12 Gflops is observe on GPU whose theorically maximal performance is roughly 17.6Gflops.[3] The implementation of sgemm on GPU is obviously easier than CPU counterpart though the algorithm are almost the same. Applying *mapreduce* template for GPU only need tens of lines code effort. Like GPU template, we apply *mapreduce* for CPU, and then obverse 95.6 peak Gflops with about 75% utilization percentage on harpertown.

V. 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(*pthread*) is a de facto standard for multi-threading on UNIX-compatible systems. The relationship between pthread and native thread provided by OS is straightforward. Therefore, abstraction of pthread is far away from expressing parallelism and concurrency naturally. 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 many 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(OMP)* [20] is designed for shared memory and has shipped in almost every C/Fortran compilers. OMP compilers transform sequential code into multi-threaded equivalence with 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 or scaling up to larger systems. Hybrid OpenMP with MPI is possible, but difficulty surges.

Sequoia [1] 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. Target machine is 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.

VI. Discussion and Future work

We present a template-based approach to perform source-to-source transoformation for programs with rich information. Because it applies metaprogramming technique, template approach does not bind to any execution model. Therefore, we can synthesize more than one models a program. In addion, our approach is flexible and extensible. Instead of modifying a compiler to add annotations or extend grammar, we implement the all functionalities by template mechanism. Template library is intimate for C++ developers so they can extend the library to adapt innovative architectures. Our approach follows C++ standards, which means that the methodology should work fine for every platform with standard-compliant C++ compilers. libvina is a prototype implementation to demonstrate the idea in this paper.

Our programming model bridges algorithm experts and diverging multcore architectures. Domain-specific experts focus on algorithms in form of conventional programming languages. They wrap functions to template classes and then pass them to *TF class* as template parameter. Template mechniasm takes responisbility to transform source code according to their targets.

[3] $17.6Gflops = 1.1Ghz * 2(SM) * 8(SP)$. nVidia declares their GPUs can perform a mad(multiply-add op) per cycle for users who concern performance over precision. However, we can not observe mad hints bring any performance improvement in OpenCL.

libvina is a prototype library, therefore some code tranformations have not be implemented yet. Blocking is mature techniques to optimize memory for single processor. We think it is also feasible to apply template transformation for locality improvement.

Streaming is an important computation model for communication-exposed parallel architectures. We paritially exploit the utilization of GPU in this paper though, it is still unclear how much efforts should be pay to develop full-blown template library to support streaming computation. Existing implementation can only deal with regular data. *Gather-and-scatter* operation is not supported by our template library. Futher work on libvina will concentrate on communication abstract of streaming computation.

General applications also contain a variety of static information to optimize even memory footprints are not regular. It is desirable explores other execution models to utilize the key hint to transform source code close to target architectures.

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/.

[21] Linderman, M. D., Collins, J. D., Wang, H., Meng, T. H.: Merge: A Programming Model for Heteogeneous Multi-core Systems. ASPLOS '08, March 1-5, Seattle, USA