A Template Metaprogramming Approach to Support Parallel Programs for Multicores

Xin Liu, Daqiang Zhang, Jingyu Zhou, Minyi Guo, Yao Shen
Department of Computer Science
Shanghai Jiao Tong University
No. 800, Dongchuan Road, Shanghai, P.R.China
{navyliu, zhangdq}@sjtu.edu.cn, {guo-my, zhou-jy, shen_yao}@cs.sjtu.edu.cn

***Abstract*—In advent of multicore era, plain C/C++ programming language can not fully reflect computer architectures any more. Source-to-source transformation helps tailor programs close to contemporary hardwares. We propose a template-based approach to perform transformation for programs with rich static information. The template metaprogramming techniques we present can conduct parallelization and memory hierarchical optimization for specific multicores. They enable programmers to utilize new architectural features and parallel patterns by extending template library. In this paper, we implement a prototype template library – libvina to demonstrate the idea. Finally, We evaluate our template library on commodity x86 and GPU platforms by a variety of typical applications in multimedia and scientific fields. In experiments, we show that our approach is flexible to support multiple parallel models and capable of transforming sequential code to parallel equivalence according to specific multicore architectures. Moreover, the cost of programmability using our approach to adapt more than one multicore platform is manageable.**

***Index Terms*—static analysis; source-to-souce transformation; parallelization; multicore; metaprogramming**

I. Introduction

Multicores rely on parallelism and memory hierarchy to improve performance. Both duplicated processors and elaborated storage-on-chip require programmers to restructure their source code and keep tuning binaries for a specific target. Therefore, non-trivial knowledge of underlying machine's architectures is necessary to write high-performance applications. More worse, various implementations of multicores bring many architectural features to programmers, which in fact further enlarge the gap between software programmers and hardware vendors.

Traditionally, algorithm experts usually focus on their specific domains and have limited insights on diverging computer systems. Writing algorithms in sequence, they expect hardwares and compilers to guarantee decent performance for their programs. The expectation was roughly held until explicit parallel system was introduced to computer community. Since the frequency of microprocessor increases slowly, it has been difficult to obtain free performance improvement from hardware's refinement. Workloads of application developers surge for parallel computer systems. In essence, it is because plain C/C++ can not fully reflect contemporary parallel architectures. It is desirable to develop methods to adapt to diverging multicore architectures.

Although extensive researches on non-traditional programming models obtain fruitful achievements, mainstream software development still stay at imperative programming languages and multi-threading. We think the primary reason is software cost. Considering the time span which a large computer system serves, hardwares are cheap and become cheaper with time elapsing, while software and well-trained personnel are expensive. Because numerous legacy software systems were designed and developed in conventional programming model, vendors usually prefer to maintain and update them rather than rebuilding from scratch. Nevertheless, exploiting horsepower of multicore processors for legacy and new systems is a moderate issue.

Source-to-source transformation can help tailor specific architectures. In particular, some transformations can extend traditional programming language to support multicore architectures. OpenMP [1] and Sequoia [2], [3] are typical examples. One distinct attribute comparing with other fancy languages is that they support progressive parallelization from original source code, which can guarantee to keep software investment. OpenMP transform program regions into fork-join threads based on pragma directives. Sequoia attempts to map computation-intensive functions on a machine tree describing in configuration file. In those extended languages, a set of language constructs are provided to support specific parallel patterns, so they need dedicated compilers. Instead of modifying compiler and introducing new language constructs, we exploit the capability of C++ template mechanism to achieve source-to-source transformation. All transformations are programmed in C++ metaprogramming [4] and are conducted by a group of template classes when that are instantiated. The primary limitation is that only static information are available at 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 at compile time, it is possible to avoid deploying run-time for transformation, which means that it can incur minimal runtime cost. Besides, our template-based approach imposes fewer restricts comparing with other static approaches:

- Flexibility: We proposed a way to perform source-to-source transformation by metaprogramming. Because it

can manipulate source code in metaprograms, our approach does not bind any parallel models. It is easy to change transform to fork-join, or perform computation as pipeline. In addition, our approach can deploy any thread implementations to support parallelism and concurrency. We experiment pthread, low-level threads provided by OS and device driver. As far as we know, no parallel programming language declares such flexibility. Theoretically, metaprogramming is as expressive as any general-purposed programming languages, so we think it is a promising approach to explore more parallel patterns beyond this paper.

- Extendability: It is extensible to develop new template class to utilize new architectural features and parallel patterns. Template metaprogramming 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. The progress is usually a year-old compaign and can not determined by software developers alone.
- Portability: Template is part of ISO C++ [5], [6]. Template-based approach is applicable for every platforms with standard C++ compiler. Template metaprogramming is widely used in other applications in C++ community and full-blown metaprogramming libraries like MPL [10] is portable. Through good encapsulation of platform details, most of code in our template approach can be reused.

The remaining parts of this paper are structured as follows. Section 2 shows techniques to perform transforms by template metaprogramming. Audiences with C++ template programming experiences or functional programming language concepts are helpful but not prerequisites. Then Section 3 presents some typical transformations by our template library. Experiments are in Section 4 to evaluate performance on both CPU and GPU. Section 5 summarizes some related works on library-based approach and language extension to support multicore architectures. The last section is conclusion and future work.

II. Template Metaprogramming Approach

A. Background

Originally, C++ template mechanism is invented to supersede C preprocessor. It is type-safed and could facility generic programming. People found the potential of template computation by chance. [7] later proved template itself is Turing completeness. Beside the job it meant to do, template has been successfully applied to many innovative purposes in modern C++ programming practices [8].

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 [9].

Metaprogramming is the writing of computer programs that write or manipulate other programs(or themselves) as their data. *C++ Template metaprogramming* is metaprograming by template techniques in C++ programming language. It is similar to functional programming language except it takes effect at compile time. It only relies on static information to determine control flow and perform computation. MPL Library [4], [10] provides language constructs and STL-like data structures corresponding to conventional C++ program, which significantly eases programming in static realm.

B. Overview

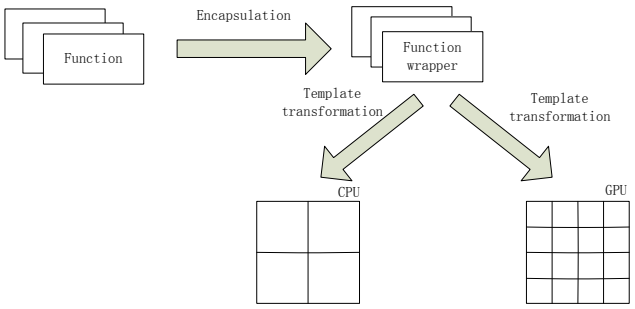

Fig. 1. template transformation

Fig. 1 is the diagram of source-to-source transformation using template metaprogramming. In our design philosophy, we separate two roles in software development. Algorithm programmers are application-specific experts, who are only concerned about efficient algorithms in conventional C/C++ form. They provide functions in the form of function wrappers. So the encapsulation process is achieved by algorithm programmers. On the other side, system programmer who know underlying computer architectures write C++ template library to take responsibility for transformation. We extend the concept of template specialization to multicore architectures. Specializations of a function wrapper for different architectures are achieved by applying a series of template classes. A regular C++ compiler can transform source codes according to deployed templates.

One foundation of our approach is to assume C++ compiler front-end as a code generator. It actually practices source-to-source transformation in the guidance of template metaprogramming. Function wrappers are parametric with respect to template arguments instead of function arguments. This enables C++ compiler manipulate functions at compile time.

We implemented a prototype template library – libvina, to perform source-to-source transformation for both CPU and GPU. On CPU, we define template classes recursively to generate variants for a function wrapper. On GPU, things became trivial because GPUs map and manage massive number of threads by hardware. Transformations are abstracted by metaprograms and are organized as a template library. Template library such as libvina enables to re-use the repetitive efforts of system programmers. Porting from one platform to another usually only need to adjust template arguments or

Fig. 2. transform class

apply another group of template classes. It is not necessary to dig into algorithms to explore parallelism again. As depicted in Fig. 1, we apply different template classes to transform the same function wrappers for CPU and GPU respectively.

C. Components

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 cluster of subprocedures. Each subprocedure may be exactly the same and spread on multicore to execute 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. 2, 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.

2) Function wrapper: In programming language, a function which can apply any values of different types are parametric polymorphism. C++ has already supported this language feature by template function. Our library needs 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 arguments 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. 3 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. 7. 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 inline optimization.

3) Predicate: Borrowed from lisp concept, *predicate* represents an indicator of some conditions. It is a template class

```
/*A function wrapper for vector addition
 */
template<class Result, class Arg0,
         class Arg1>
struct vecAddWrapper {
 //omitted...

 //entry
 static void
 doit(const Arg0& arg0, const Arg1& arg1,
     Result& result)
 {
  vecArithImpl<T, DIM_N>::add(arg0, arg1,
                              result);
 }

 //entry with a barrier to synchronize
 static void
 doit_b(const Arg0& arg0, const Arg1& arg1,
     Result& result, mt::barrier& barrier)
 {
   doit(arg0, arg1, result);
   barrier.wait();
 }

 //...
};
```

Fig. 3. Function Wrapper

with static fields initialized by constant expressions consisting of template parameters and constants. These fields are automatically evaluated when template classes are instantiated. Fig. 4 is an example to determine whether the problem size is fitting for last level cache.

```
/*determine whether the problem size is
 *less than 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;
};
```

Fig. 4. Predicate

4) Sentinel: Sentinels in libvina are non-type template parameters of *TF class*. When a template class is instantiating, sentinels are evaluated from a *predicate*. The *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 class to instantiate basing on the values or types of template arguments. The most important application of *sentinel* is to terminate recursion. More general

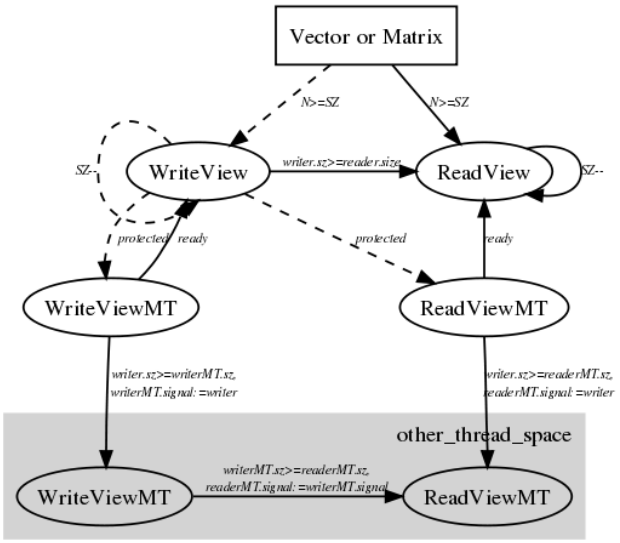

Fig. 5. View classes in libvina

flow control such as branch is available in MPL [10].

D. Supporting data structures

Template metaprograming works in compile-time. Therefore, problems we can 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 FPGA or DSP. To leverage static information, data structures need to associate template arguments with such information. Only vector and matrix are implemented in libvina, because they cover many applications in the fields mentioned previously. Users require more versatile data structures can also resort to MPL.

A *View* class is a concept to represent data set. Fig. 5 depicts relationship of views in libvina. 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 threads is through a special view class named ViewMT.

Modern multicore architectures emphasize on utilization of bandwidth and storage-on-chip, therefore we manipulate data in bulk. Essentially, a view is an abstract of *stream* and it helps programmers build streaming computation. Underneath view classes, we can perform specialization based on architectures. *e.g*, it is not necessary to duplicate data on shared memory, or we can raise asynchronous communication to memory hide latency. In addition, a 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. Runtime supports

Basically, our static transformation does not incur any runtime overhead. However, we need thread and synchronization to execute in multi-threaded environment.

We implemented *mt::thread* based on underlying 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. Other advantages of libSPMD is that it provides hook function to perform reduction and CPU binding. On GPU, we use OpenCL [11] 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 mentioned before are varying, our library can deal with them well.

In libvina, we implement barrier for both CPU and GPU. CPU's implementation uses pthread conditional variables, while GPU counterpart uses openCL's API function. GPU's barrier synchronizes *work-groups*. We still can not find a good method to generate kernel barrier statement worked on *work-items*.

We define a signal class to synchronize, which mimic signal primitive on CellBE [12]. It is implemented by conditional variable on x86. On GPU, we use *event* of OpenCL to achieve the same semantics.

IV. Source Transformations by Template

We demonstrate transformation for two kinds of parallel pattern using our template library. It is noting that we only apply recursion of template classes on CPU. GPUs has dedicated hardware to map massive number of threads on physical cores. In addition, we leverage OpenCL API on GPU, which supports data parallelism and task parallelism. So transformation for both SPMD and streaming becames on GPU are straightforward.

A. SPMD

Multithreading is dominant approach to utilize cloned computational resources. SPMD model is the most intuitive thread model. SPMD is also the foundation of streaming computation. There are numerous kernel functions in multimedia applications and scientific computation which can exploit 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 Sequoia as template classes.

Fig. 6 is a definition of *TF_mappar* for x86. Template parameter *Instance* is an adapter, which will be described in next section. The last template parameter of *TF_mappar* is a *sentinel*. It determines control flow when instantiation occurs. The second class in the figure is a template partial specialization for the prime template, which generates concrete thread to perform computation. It is noteworthy that the last two arguments are true, 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 subset of the view or trivially return itself according to the template parameters. Fig. 6 gives a definition of arg0_isomoph.

Using the same mechanism, we define *TF_mapreduce* class. Instance provides a reduction function, which will perform after the barrier of loop. Fig. 7 is a cluster of subprocedures generated by libvina after applying *TF_mapreduce* to a matrix multiplication function. Subprocedure is the same concept of *variant* in [2], [13]. Template parameter _K is 2 in this case because we intend to perform this transformation for a dual core processor. *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 subprocedures are executed simultaneously in pairwise. Dashed lines indicate logical synchronization, implemented by a barrier.

mapseq in Sequoia 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 parallel computation. It models data set as a *stream*. Operations are usually organized in pipeline way to process in turn, while keeping stream in local storage. It can utilize multicore to perform computation in parallel and reduce external bandwidth. Imagine [14] is typical architecture for streaming computation.

More general streaming computation does not restrict to keep data stationary. Pipeline processing inherently support heterogeneous architectures or built-in ring network on chip [15], [16]. If specific processors are exposed by platform and communication cost is manageable, developers intend to leverage them for performance or energy advantages. Given the capability of metaprogramming, we have no problem to link external computational devices as long as developers provide communication layers. OpenCL is a programming model to support such kind of hybrid execution for host CPU and its accelerators.

Our template library provides two components to support streaming computation. First, we provide multi-threaded *View* classes depicted in Fig. 5. A *ViewMT* class contains a signal object to safeguard the ownership of underlying data set. Second, libvina defines a TF class listed in Fig. 8 to perform pipeline transformation. It synthesizes a group of function wrappers and generates an execution chain with ViewMTs. Fig. 9 depicts the scenario of pipelining in multi-threading environment. A stage passes its ownership of data through ViewMT classes. Signal classes take responsibility for waking up following stages. It is noteworthy that we intentionally leave the tail of recursion undefined. Our template library has no idea how to deal with the output of pipeline. It is user's job to define the last stage.

V. Adaption of TF classes

TF classes manipulate instance classes. They transform the source code based on properties of instances. *e.g.*

```
/*A TF class for mappar. no dependence exists
 *in subtasks of Instance.
 */
template <class Instance,          /*problem*/
          int _K,       /*number of children*/
          bool _IsMT,      /*multi-threaded?*/
          bool __SENTINEL__ = Instance::_pred
>
struct TF_mappar {
 /*define recursively, evalue __SENTINEL__
  *using subtask's predicate
  */
 typedef TF_mappar<typename Instance::SubTask,
       _K, _IsMT,
       Instance::SubTask::_pred> _Tail;

 //determine whether Arg0 is isomorphic
 typedef typename mpl::or_<mpl::bool_
   <std::tr1::is_arithmetic
     <typename Instance::Arg0>::value>
   ,boost::mpl::bool_<
   std::tr1::is_same<typename Instance::Arg0,
             typename Instance::SubTask::Arg0>
      ::value>
   >::type
 arg0_isomorph;

 //omitted ...

 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 = aux::subview<decltype(arg0),
        arg0_dim::value, arg0_isomorph::value>
            ::sub_reader(arg0, k);
    auto subArg1 = ...
    auto subResult = ...

    //execute recursively
    _Tail::doit(*subArg0,*subArg1,*subResult);
   }
 }
};
/*A partial template specialization of
 *TF_mappar.
 */
template <class Instance, int  _K>
struct TF_mappar <Instance, _K
             ,true          /*multithreaded*/
             ,true/*predicate is statisfied*/>
{
 //omitted...
 static void
 doit(const typename Instance::Arg0& arg0,
   const typename Instance::Arg1& arg1,
   typename Instance::Result& result)
 {
   auto compF = Instance::computationMT();

   /*bind function object to a thread
    *and run it.*/
   mt::thread_t leaf(compF, arg0, arg1,
     aux::ref(result, result_arithm()));
 }
};
```

Fig. 6. TF class of *TF_mappar*

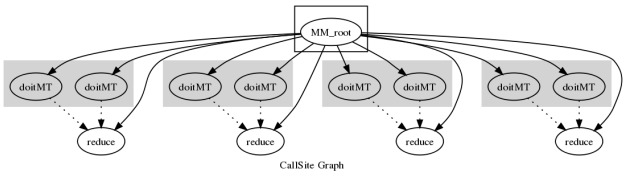

Fig. 7. MM internal call graph

```
/*A TF class for pipeline. User need to define
 *spacialization to handle the output of
 *pipeline.
 */
template <typename... Stages>
struct TF_pipeline;

template <class P, typename... Tail>
struct TF_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, omitted...
    TF_pipeline<Tail...>::doit( P::doit(in) );
  }
};
```

Fig. 8. TF class of pipeline

TF_mapreduce utilizes the fact that instance is divisible and reducible. TF_pipeline utilizes the fact that instances are combinable. User programmers are free to define their instance class to adapt libvina's TF class to their specific problems. The only requirement is that they have to define the properties using static members. Besides, TF classes need a couple of interfaces to manipulate instance, including types and function objects. We present two examples of instance classes to demonstrate how to adapt customer's codes.

Fig. 10 is definition of an instance for *saxpy*. We will describe the algorithm later. The adapter defines TF_mappar and call it in entry. The instance class recursively defines its subtask. Essentially, it defines how to divide the instance. We evaluate the predicate *Pred* in instance class and the value is used by TF class as initializer of template argument. Type interfaces include Arg0, Arg1 and Result. A static function returns a function object which instantiates function wrapper *Func* using parameters in context. Fig. 11 shows the usage of our adapter. Comparing with original call, new entry keeps funciton name and arguments intact. However, inside of TF_MT::saxpy, a cluster of subprocedures are generated and executed using threads..

Fig. 12 is an excerpt of program *langpipe*. Fig. 8 utilizes variadic template [17] to define recursion, which is one of C++0x features. The distinct of the TF_pipeline class in Fig. 12 is zero template argument. We define a full template specialization for the last stage of pipelining. The assignment of *reader* is tricky. The type of result is an regular view, while

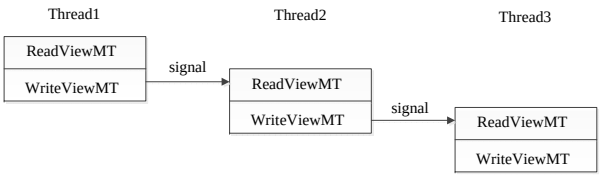

Fig. 9. ViewMT in pipelining

```
/*A adapter class for algorithm saxpy.
 */
template <class RESULT, class T,   class RHS,
    template <typename, typename> class Func,
    template <typename, int>      class Pred,
              int K = 2 , bool IsMT = false>
struct ADAPTER_saxpy{
  //trait classes for type resolution
  typedef view_trait<RHS>    trait1;
  typedef view_trait<RESULT> trait_res;

  //types interfaces used by TF class
  typedef T                                  Arg0;
  typedef typename trait1::reader_type       Arg1;
  typedef typename trait_res::writer_type
                                             Result;
  //define TF class
  typedef TF_mappar<ADAPTER_saxpy/*instance*/
                    ,K, IsMT> Map;

  //define subTask
  typedef ReadView<T, trait1::READER_SIZE / K>
    SubRView1;
  typedef WriteView<T
              ,trait_res::WRITER_SIZE / K>
    SubWView;
  typedef ADAPTER_saxpy<SubWView, T, SubRView1,
                  Func, Pred, K, IsMT>
    SubTask;

  //pre-calculate predicate for TF class
  const static bool _pred =
   Pred<T, trait1::READER_SIZE>::value;

  static void
  saxpy(const T& alpha, const Arg1& lhs,
       Result& rhs)
  {
    Map::doit(alpha, lhs, rhs);
  }

  static std::tr1::function
  <void (const T&, const Arg1&, Result&)>
  computation()
  {
    return &(Func<Result, Arg1>::doit);
  }
};
```

Fig. 10. an adapter class for saxpy

```
//regular usage
vina::saxpy<VEC_TEST_TYPE, VEC_TEST_SIZE_N>
(7, x, result);

//use TF class to perform saxpy
typedef ADAPTER_saxpy<Writer, VEC_TEST_TYPE
           ,TestVector          /*Vector type*/
           ,wrpSaxpy      /*function wrapper*/
           ,p_lt_cache_ll        /*predicate*/
           ,2/*K*/     ,true/*Multi-threaded*/
> TF_MT;
TF_MT::saxpy(7.0f, x, result);
```

Fig. 11. Call function with and without transformation

the type of argument *in* is uncertain. If it is a regular view, the assignment is trivial. If it is a ViewMT class, the operation is blocking as depicted in Fig. 9. *MYPIPE* is the synthesized class generated by TF_pipeline. Each *translate* is independent instance, which complies with the type interfaces defined in TF_pipeline. Because instances access contend data through ViewMT classes, it is combinable.

VI. Experiments and evalution

A. Methodology

We implement our library in standard C++ [5], [6]. Theoretically, any standard-compliance C++ compiler should process our classes without trouble. New C++ standard (a.k.a C++0x) [6] adds a lot of language features to ease template metaprogramming. 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 like libvina should become easier and smoother in the future. Currently, C++0x has been partially supported by some mainstreaming compilers. We developed the library and tested using GCC 4.4.0. The first implementation of OpenCL was shipped by Mac OSX 10.6. The GPU performance is collected on that platform.

A couple of algorithms are evaluated for our template approach. They are typical in image processing and scientific fields. In addition, we implemented a psuedo language translation program to illustrate pipeline processing. The programs in experiments are listed as follows:

- *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 4096*4096 dense matrices multiply.
- *dotprod* Two vectors perform dot production. Each vector comprises 32 million elements.
- *conv2d* 2-Dimensional convolution operation on image. The Image is 4094*4096 black-white format. Pixel is normalized as a single float ranging from 0.0 to 1.0.
- *langpipe* Pseudo-Multi-language translation. A word is translated from one language A to language B, and then another function will translate it from language B to language C, etc.

```
/*template full specialization for langpipe
 */
template<>
struct TF_pipeline<>
{
 static const bool _IsTail = true;
 typedef view_trait<STRING>::reader_type
   READER;

 template <class U>
 static void output(U* in)
 {
   int i;
   /*U may be a ReadViewMT, in which case
    *the assignment is blocking operation.
    */
   READER reader = *(in);
   char output[READER::VIEW_SIZE+1];

   for(i=0; i<READER::VIEW_SIZE; ++i)
     out[i] = reader[i];
   out[i] = '\0';
   std::cout << out << std::endl;
 }

 template <class T>
 static void doit(T * in)
 {
   std::tr1::function<void (T*)>
     func(&(output<T>));

   mt::thread_t thr(func, in);
 }
};

//customize pipeline TF class
typedef TF_pipeline<
   translate<Eng2Frn
            ,true/*Multi-threaded?*/
           >,
   translate<Frn2Spn, true>,
   translate<Spn2Itn, true>,
   translate<Itn2Chn, true>
> MYPIPE;

//function call
MYPIPE::doit(&input);
```

Fig. 12. Call function before and after transformation

Two multicore platforms are used to conduct experiments. The hardware platforms are summed up in Table. I

On harpertown, we link Intel Math kernels to perform BLAS procedures except for conv2d. On macbookpro, we implemented all the algorithms on our own. For CPU platform, we link libSPMD thread library to perform computation. The library binds CPUs for each SPMD thread and switch to realtime scheduler on Linux. This configuration helps eliminate the impact of OS scheduler and other processes in the system.

B. Evaluation

TABLE I
EXPERIMENTAL PLATFORMS

name	type	processors	memory	OS
harpertown	SMP server	x86 quad-core 2-way 2.0Ghz	4G	Linux Fedora kernel 2.6.30
macbookpro	laptop	x86 dual-core 2.63Ghz GPU 9400m 1.1Ghz	2G 256M	Mac OSX Snowleopard

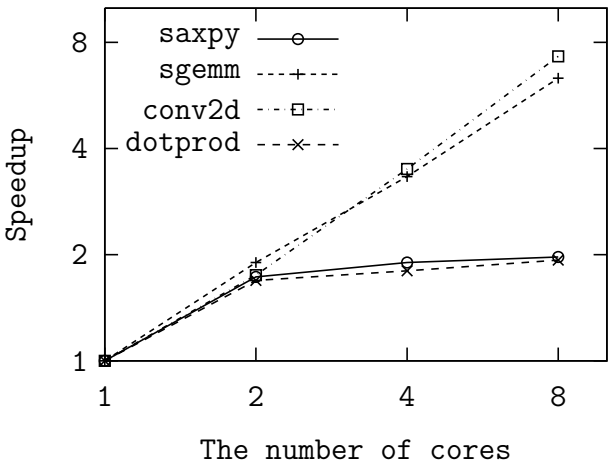

Fig. 13. Speedup on Harpertown

1) Speedup of SPMD transformation on CPU: Fig. 13 shows the speedup on harpertown. The blade server contains two quad-core Xeon processors. We experiment SPMD transformation for algorithms. *saxpy* and *conv2d* apply *TF_mappar* while *dotprod* and *sgemm* apply *TF_mapreduce*.

We obverse good performance scalability for programs *conv2d* and *sgemm*. *conv2d* does not have any dependences and it can obtain about 7.3 times speedup in our experiments. sgemm needs an extra reduction for each division operation. The final speedup is about 6.3 times when all the cores are available. It is worth noting that we observe almost two-fold speedup from sequence to dual core. However, the speedup degrates to 3.3 time when execution environment change to 4-core. Harpertown consists of 2-way quad-core processors, Linux can not guarantee that 4 subprocedures are executed within a physical processor. Therefore, the cost of memory accesses and synchronization increases from 2-core to 4-core platform.

dotprod and *saxpy* reveal low speedup because non-computation-intensive programs are subject to memory bandwidth. In average, *saxpy* needs one load and one store for every two operations. *dotprod* has similar situation. They quickly saturate memory bandwidth for SMP system and therefore perform badly. Even though we fully parallelize those algorithms by our template library.

2) Speedup of SPMD transformation on GPU: Fig. 14 shows SPMD transformation results for GPU on macbookpro. Programs run on host CPU in sequence as baseline. Embedded

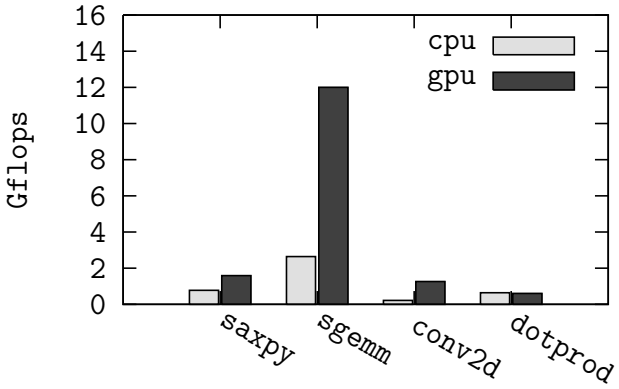

Fig. 14. Speedup Comparing GPU with CPU

GPU on motherboard contains 2 SMs[1]. Porting from CPU to GPU, developer only need a couple of lines to change templates while keeping algorithms same [2]. As figure depicted, computation-intensive programs *sgemm* and *conv2d* still maintain their speedups. 4.5 to 5 times performance boost is achieved for them by migrating to GPU. In addition, we observe about 2 times performance boost for *saxpy*. Nvidia GPUs execute threads in group of warp (32 threads) on hardware and it is possible to coalesce memory accesses if warps satisfy specific access patterns. Memory coalescence mitigates bandwidth issue occurred on CPU counterpart. Because our program of *dotprod* has fixed step to access memory which does not fit any patterns, we can not obtain hardware optimization without tweaking the algorithm.

TABLE II
COMPARISON OF SGEMM ON CPU AND GPU

	baseline	CPU	GPU
cores	1 x86(penryn)	8 x86(harpertown)	2 SMs
Gflops	2.64	95.6	12.0
effectiveness	12.6%	74.9%	68.2%
lines of function	63	unknown	21

3) Comparison between different multicores: Table. II details *sgemm* execution on CPU and GPU. Dense matrix multiplication is one of typical programs which have intensive computation. Problems with this characteristic are the most attractive candidates to apply our template-based approach. Our template library transforms the *sgemm* for both CPU and GPU. We choose sequential execution on macbookpro's CPU as baseline. After mapping the algorithm to GPU, we directly obtains over 4.5 times speedup comparing with host CPU. Theoretically, Intel Core 2 processor can issue 2 SSE instructions per cycle, therefore, the peak float performance is 21 Gflops on host CPU. We obtain 2.64 Gflops which effectiveness is only 12.6% even we employ quite complicated

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

[2]Because GPU code needs special qualifiers, we did modify kernel functions a little manually. Algorithms are kept except for sgemm. It is not easy to work out sgemm for a laptop, so we added blocking and SIMD instruments for CPU.

implementation. On the other side, 12 Gflops is observed on GPU whose maximal performance is roughly 17.6 Gflops.[3] Although both column 2 and column 4 implement SIMD algorithm for *sgemm*, GPU's version is obviously easier and effective. It is due to the dynamic SIMD and thread management from GPU hardware [18] can significantly ease vector programming. Programmer can implement algorithm in plain C and then replies on template transformation for GPU. Adapting *TF_mapreduce* template class for GPU only need tens of lines code efforts. Like GPU template, we apply *TF_mapreduce* to parallelize *sgemm* procedure in MKL for CPU. We observe 95.6 Gflops and about 75% effectiveness on harpertown server.

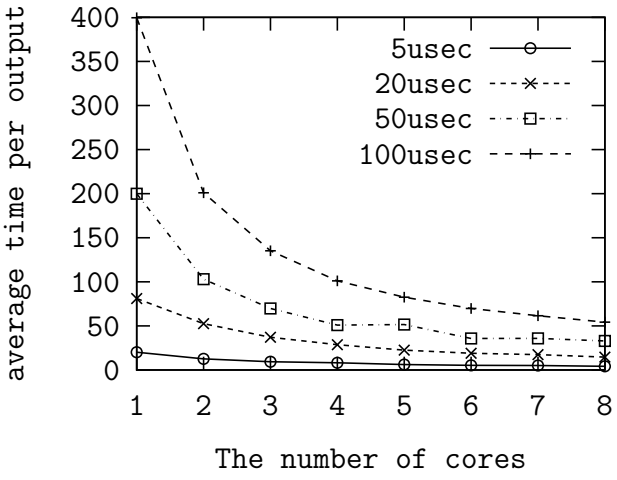

Fig. 15. Pipeline Processing for Psuedo Language Translation

4) Support streaming computation: Fig. 15 demonstrates pipeline processing using our template library. As described before, *langpipe* simulates a multilingual scenario. We apply template *TF_pipeline* listed in Fig. 8. In our case, the program consists of 4 stages, which can transitively translate English to Chinese[4]. Only the preceding stages complete, it can proceed with the next stages. The executing scenario is similar to Fig. 9. We use bogus loop to consume t μs on CPU. For each t, we iterate 500 times and then calculate the average consumptive time on harpertown. For grained-granularity cases (20μs, 50μs, 100μs), we can obtain ideal effectiveness in pipelining when 4 cores are exposed to the system. *i.e.* our program can roughly output one instance every t μs. The speedup is easy to maintain when granularity is big. 100 μs case ends up 54 μs for each instance for 8 cores. 50 μs case bumps at 5 cores and then improves slowly along core increment. 20 μs case also holds the trend of first two cases. 5 μs case is particular. We can not observe ideal pipelining until all 8 cores are available. Our Linux kernel scheduler's granularity is 80 μs in default. We think that the very fine granular tasks contend CPU resources in out of the order. The runtime behavior presumably incurs extra overhead. Many cores scenario helps alleviate the situation and render regular pipeline processing.

[3] $17.6Gflops = 1.1Ghz * 2(SM) * 8(SP)$. nVidia declared 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.

[4] follow the route: English → French → Spanish → Italian → Chinese

VII. Related work

As mentioned before, it is desirable to extend conventional programming languages to reflects new hardwares. Researches in the field have two major directions:

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

First, library is a common method to extend language capability without modifying grammar. Pthread library is a *de facto* standard for multi-threading on POSIX-compatible systems. The relationship between pthread and native thread is straightforward. Therefore, abstraction of pthread is far away from expressing parallelism and concurrency naturally. Furthermore, the implementation of thread on hardware is undefined in the standard, so it can not guarantee performance or even correctness on some architectures [19]. C++ community intend to develop parallel library while bearing generic programming in mind. TBB [20] has a plenty of containers and execution rules. Entities including partitioner and scheduler in TBB are created at run time. 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 lightweight threads are executing by hardware. Template-based approach we proposed is orthogonal to runtime parallel libraries. We only explore parallelism which can be determined at compile time, developers feel free to deploy other ways such as TBB to farther improve programs.

The second choice for language community is to extend language constructs by modifying compiler. They add directive or annotation to help compiler transform source code. OpenMP compilers transform sequential code into multi-threaded equivalence. The run-time is usually provides in the form of dynamic link library. Although it is simple and portable, the performance is not optimal in most cases. Moreover, a handful of directives in OpenMP leave small room for further improving performance or scaling up to larger systems. Hybrid OpenMP with MPI is possible though, difficulties surge. Sequoia supports 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 transforms a *task* into a cluster of *variants*. Target machine is described in XML files. [2], [3] report that Sequoia can transform programs for CellBE, cluster while keeping competitive performance. That is at expense of implementing one compiler for each platforms. The primary drawback of Sequoia is that its language constructs can not cover common parallel patterns such as pipeline or task queue. Besides, sequoia compiler ignores type information to select optimal

implementation. Merge [13] features a uniform runtime environment for heterogeneous multicore systems in forms of task and variant. However, Merge only support *map-reduce* programming model. Its run-time overhead is not negligible for fine-granularity parallelism. Methods mentioned before all need non-trivial efforts to modify compilers. As discussed in [2], the authors of the Sequoia were still not clear whether the minimal set of primitives they provided provides can sufficiently express dynamic applications. We doubt if it is worthwhile to invest a compiler given the fact that template library can also achieve the same functionalities.

VIII. Discussion and Future work

The silicon industry has chosen multicore as new direction. However, diverging multicore architectures enlarge the gap between algorithm-centric programmer and computer system developers. Conventional C/C++ programming language can not reflect hardware essence any more. Existing ad-hoc techniques or platform-dependent programming language pose issues of generality and portability. Source-to-source transformation can meet the challenge and help tailor programs to specific multicore architectures.

We present a template metaprogramming approach to perform source-to-source transformation for programs with rich information. Because it applies metaprogramming technique, template library is flexible enough to apply any parallel patterns and execution models. In addition, our approach is extensible. Instead of modifying a compiler to add annotations or language constructs, we implement the whole functionalities by template mechanism. Template metaprogramming is intimate for C++ programmers so they can extend the library to facilitate proper parallel patterns and new architectural features. Our approach follows ISO C++ standards, which mean the methodology is guaranteed to work across platforms. Experiments shows that our template approach can transform algorithms into SPMD threads with competitive performance. These transformation are available for both CPU and GPU, the cost of migration is manageable. We also transform a group of standalone function wrappers into a stream using our template library. It demonstrates that template metaprogramming is powerful enough to support more than one parallel pattern.

Streaming is an important computation model for innovative multicore architectures. We partially exploit GPU functionality in this paper though, transformation for GPU is quite straightforward. It is still unclear how many efforts need to pay for a full-blown template library, which support streaming computation. Libvina can only deal with regular data. Future work on view class will concentrate on supporting general operations like gather and scatter etc. Currently, kernel functions in GPU prohibit recursion. So we believe that it is beneficial to introduce template recursion for GPU kernel functions. TF classes which support strip-mined memory access and loop iteration transformation are particularly attractive for GPU targets because because GPUs provide memory coalescence for specific access patterns.

On CPU, source-to-source transformation should go on improving data locality of programs. We plan to explore template approach to generalize blocking and tiling techniques. It is also possible to re-structure or prefetch data using template metaprograming accompanying with runtime library.

General applications also contain a variety of static information to optimize.The problem is that their memory footprints are irregular and very hard to identify. It is desirable to explores new TF classes to facilitate transforming source code close to target architectures using the static information.

References

[1] (2008) Openmp specificaiton version 3.0.

[2] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Memory - sequoia: programming the memory hierarchy," in *SC*. ACM Press, 2006, p. 83.

[3] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *PPOPP*, K. A. Yelick and J. M. Mellor-Crummey, Eds. ACM, 2007, pp. 226–236.

[4] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[5] "So/iec (2003). iso/iec 14882:2003(e): Programming languages - c++," 2003.

[6] "So/iec n2960, standard for programming language c++, working draft," 2009.

[7] T. L. Veldhuizen, "C++ templates are turing complete."

[8] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[9] B. Stroustrup, *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Professional, February 2000.

[10] D. A. Aleksey Gurtovoy. The boost c++ meta-programming library.

[11] A. Munshi, "The opencl specification version 1.0," 2009.
[12] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.
[13] M. D. Linderman, J. D. Collins, H. W. 0003, and T. H. Y. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ASPLOS*, S. J. Eggers and J. R. Larus, Eds. ACM, 2008, pp. 287–296.
[14] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," in *ICCD*. IEEE Computer Society, 2002, pp. 282–288.
[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.
[16] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*. New York, NY, USA: ACM, 2008, pp. 1–15.
[17] D. Gregor and J. Järvi, "Variadic templates for c++," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 1101–1108.
[18] K. Fatahalian and M. Houston, "Gpus: A closer look," *Queue*, vol. 6, no. 2, pp. 18–28, 2008.
[19] H.-J. Boehm, "Threads cannot be implemented as a library," in *PLDI*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 261–268.
[20] Intel thread building blocks reference manual.