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. Source-to-source transformation helps tailor programs close to contemporary hardwares. In this paper, 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. 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.**

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 advantage 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. However, the ad-hoc approaches of source-to-souce transformation can not support general parallel patterns. It is not a trivial task to determine how many language constructs should be provided by compilers to well support the full spectrum of multicores.

We present an approach to perform source-to-souce transformation to support multicore using C++ template mechanism. We uses *tasks* to abstract computation-intensive and side-effect free function. Our primary idea is to extend C++ template specialization to task for different multicores. Template classes can transform a task into other forms according the parallel patterns and then map to threads to execute in parallel.

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++ [4], [5]. 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 [6] 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

Libvina utilizes C++ template mechanism to perform source-to-source transformation for multicores. We uses *tasks* to abstract side-effect free functions. A task is wrapped in the form of template class. *TF classes* are able to manipulate tasks, which take responsiblilty for transforming a tasks into a group of subtasks. Finally, we map subtasks into threads for specific architectures. Fig. 1 depicts the diagram of template-based programming model.

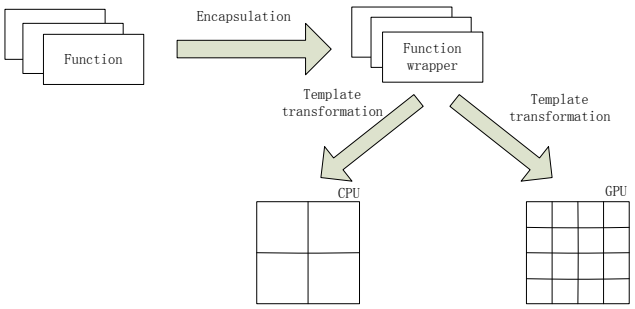

Fig. 1. template transformation

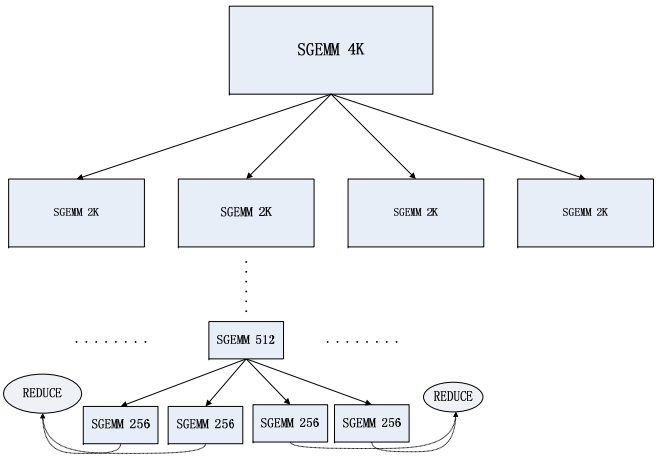

Fig. 2. Matrix-multiplication Example

```
template <class ARG0, class ARG1,
          class RESULT,
 template <class, class> class PRED/*predicate*/
        int K,   /*param to divide task*/
       >
struct SGEMM {
 typedef view2_trait<ARG0> trait_arg0
 //omit other trait classes...

 typedef typename
   ReadView2<typename trait_arg0::type,
                     trait_arg0::M / K,
                     trait_arg0::N / K>
 SubARG0;
 //omit other arguments definitions...


 typedef SGEMM<SubARG0, SubARG1, SubRESULT,
               PRED, K> SubTask;

 typedef TF_hierarchy<SubTask, SubARG0, SubARG1,
                      SubRESULT, PRED>
 TF;

```

```
static void //static entry
doit(const ARG0& arg0, const ARG1& arg1,
      RESULT& res)
{
  //lambda for iteration
  auto subtask = [&](int i, int j)
  {
     //temporaries
     Matrix<typename trait_res::type,
            SubARG0::M,
            SubARG1::N> tmp[K];

     //lambda for map
     auto m = [&](int k) {
       TF::doit(arg0.subR<SubARG0::M, SubARG0::N>(i, k),
                arg1.subR<SubARG1::M, SubARG1::N>(k, j),
                tmp[k].subW());
     };
     //lambda for reduce
     auto r = [&](int i, int j) {
          tmp[i] += tmp[j];
     }
     mapr<K, decltype(m)&, decltype(r)&>
     ::apply(m, r, res[i][j]);
  };

  typedef decltype(subtask)& closure_t;
  par< par<par_tail, K>, K, closure_t>
  ::apply(par_lv_handler2(subtask));
}/*end func*/f

void
operator() (const ARG0& arg0, const ARG1& arg1,
            RESULT& res)
{
   //algorithm of matrix-multiplication
   //omit...
}
};
```

```
//template full specialization for langpipe
template<>
struct TF_pipeline<>
{
 //last stage defintions
 //T* is the type of input parameter.
 template<class T>
 static void impl(T* in)
 {
   //omit...
 }
template<class T>
static void doit(T * in)
{
  std::tr1::function<void (T*)>
    func(&(impl<T>));

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

//customize pipeline TF class
typedef TF_pipeline<
  translate<Eng2Frn>,
  translate<Frn2Spn>,
  translate<Spn2Itn>,
  translate<Itn2Chn>
> MYPIPE;

MYPIPE::doit(&input);
```

Applications using our template-based programming model are free to choose parallel pattern. An example using hierarchical division like Sequoia is shown in Fig 2. A matrix-multiplication application can be divided into many submatrices multiplication and reduced them into the result. We can apply a TF class dedicated to this parallel pattern and it hierarchically generates a subtasks and reductions. The decomposition rule and termination of recursion is programmed in template metaprogramming.

Our programming model facilitates to separate two roles in software development. Algorithm-centric programmers only concern of algorithm in convential C/C++ form. They provide computation-intensive and side-effect free functions in the form of task. On the other side, system programmers known underlying architecture are charge in developing and applying template class to specialize tasks for the concrete target. This separation not only solve the difficulties of writing and tuning multicore, but also providing a uniform programming model to develop effecient and portable parallel programs.

The primary limitation of libvina is that we perform transformation using template metaprorgramming. Only static information, *i.e.* static constant value and types in C++, is available at compile time. For example, we utilize array dimensions(as template arguments) to estimate problem size in aforementined matrix-multiplication programs. Thus, our programming model orients to programs which contain rich static information. In embedded and scientific applictions, the runtime with fixed parameters are significantly longer than compile time even the time developing programs. Therefore, metaprogramming will pay off

III. LIBVINA: A TEMPLATE LIBRARY

We implement a template library, libvina, to demonstrate our approach. Libvina consists of 3 components: (1) Data structures, associated with static information as template parameters. (2) Building blocks, provides basic iterations to divide tasks (3) TF class, a TF class represent a parallel pattern.

A. Data Structure

To leverage static information, libvina need to associate template parameters with ADTs' parameters. For example,

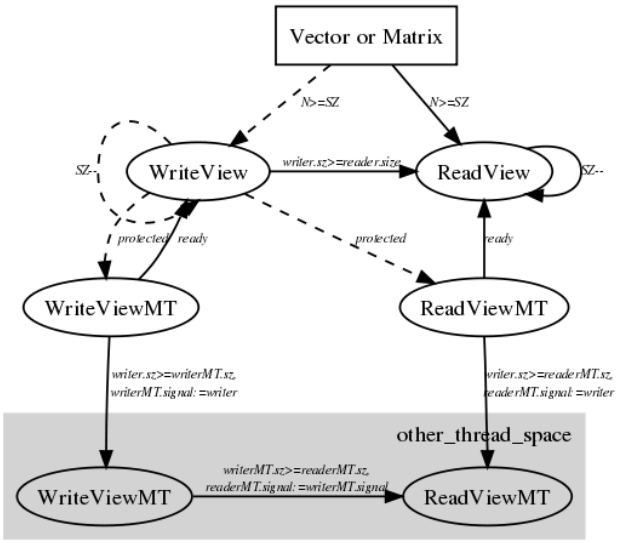

Fig. 3. View classes in libvina

for Matrix class, it cantains 3 template paramters: type, the number of row, the number of column.

A *View* class is a concept to represent data set. Fig. 3 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 kind of view called *ViewMT*.

The primary aim of view class is to hide communication. For shared memory system or communication-exposed multcore [**?**], [7], we have chance to implement data movement according to target's archictures. 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.

B. Buiding Block

Libvina provides a group of building blocks to execute subtasks. To parallelize program, we expect that most subtasks are executed in SPMD(Single-Program-Multiple-Thread). However, we needs to deal with dependences carefully to guarantee correctness.

Similar to constructs in traditional programming languages, our building block support nesting defintion. Both *seq* and *par* is interoperatable. for example, we can define a block as follows:

```
seq<par<par_tail, 4>, 3, F>::apply();
```

build to level-2 loop, and the nested loop are executed in parallel. Its equivalence in OpenMP as follows:

```
int i, j;
F f;
for (i=0; i<3; ++i)
{
  #pragma omp parallel private(j)
  for (j=0; j<4; ++j)
    f(i, j);
}
```

TABLE I
BUILD BLOCKS IN LIBVINA

Name	Semantics	Example
seq	execute *F* in squence	seq<seq_tail, 5, F>::apply();
par	execute *F* in paralllel	par<par_tail, 4, F>::apply();
mapr	execute *M* in parallel, *R* and then perform after barrier	mapr<8, F, R>::apply()

C. TF class

TF class is the short form of *TRansformation class*. A side-effect free function is referred to as *task* in libvina. Mathematically, a function is single-target binary relation. As a rule of thumb, computation-intensive functions are usually self-contained, *i.e.* external data references are limited and calling graphs are simple. Therefore, it's possible to decouple a task into a cluster of subtasks. The subtasks may be identical except for arguments and we can distribute them on multicore to execute in parallel. Another approach is to divide a complicated task into finer stages and run in pipeline manner to respect data locality and bandwidth. A *TF class* is a template class repsenting a parallel pattern which transforms a task to a cluster of subtasks in isomorphism.*i.e.* the transformed task has the same interface while owns a call graph to complete the original computation by a cluster of subtasks. TF class also takes responsibility for programming execution model.

To demonstrate more than one paralleling model, we implement two TF class as follows:

- TF_hierarchy It will recursively generate subtasks until predicate is evaluate as true. As Fig. **??** depict, We use TF_hierarchy to implement programming model similar to Sequoia.
- TF_pipeline Input arbirary number of function, the template class can synthesize a call chain. This is common pattern for streaming programming mode.

IV. USER ADAPTION FOR LIBVINA

User programmers need to customize their code to utilize Libvina. Technically speaking, we provide a group of *Concepts* to support transformation and expect user programmer *Model* our template classes.

A. Function Wrapper

Function wrapper is an idiom in libvina. Our approach needs to manipulate template functions according to their template arguments.However, a template function is unaddressable until it is instantiated. So user programmers have to bind their template functions to static entries of classes. Line 25 of List. **??** is the case. In order to cooperate with other entities, naming convention of entry is predefined. In libvina, TF classes use name *doit* and building blocks use *apply*.

B. Adaption for TF_hierarchy

Line.7 23 in Fig. **??** are adaption for TF_hierarchy. The codes define the type of SubTask for SGEMM at Line.18. It is used to define TF_hierarchy class as TASK template parameter. PRED template parameter at Line.22 is a predicate and TF_hierarchy class will evaluate it using SubARG0 and SubARG1. Line.39 calls customized TF class after dividing task. According to template argument, TF class determine whether reenter the static entry at Line. 25 or call *call* operator at Line.57 to perform computaton. Fig. **??** illustrates instantiation process of TF_hierarchy and the final result is depicted in Fig. **??**

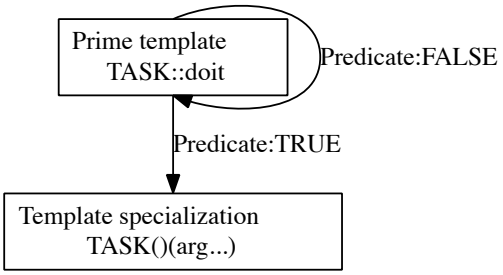

Fig. 4. Algorithm of TF_hierarchy

C. Adaption for TF_pipeline

To leverage TF_pipeline, user programmers have to provide a full specialization template class for it. It is because that TF_pipeline can only synthesize functions and execute it in sequence. It does not know how to process the output. A full specialization of TF_pipeline very defines this behavior and is called at last. For our example of *langpipe*, Line.2 20 is the case. static entry at Line.13 is served for TF_pipeline. We spawn a thread to handle with the output of precious last stage. Line.23 28 is a usage of TF_pipeline with 4 standalone functions. It is noteworthy that each function e.g *translate<Frn2Spn>* has to follow type interfaces and define dependences. In *lang_pipe* case, we utilize our ViewMT despicted in Fig. 3. ReadViewMT is only generated from WriteView or WriteViewMT. The first case represents intialization. The second builds dependence transparently when type conversion occurs. We use signal mechanism to provoke downstreaming stages. Fig. **??** illustrates the scenario contains 3 threads.

V. Implementation details

We implement all the functionalities described before using C++ template metaprogramming technique. The grand idea is to utilize template specialization and recursion to achieve control flow at compile time. Besides template mechanism, other C++ high level abstracts act important roles in our approach. Function object and bind mechnism is critical to postpone computation at proper place with proper enviroment [**?**]. To utilize nested buiding blocks, lambda expression generate closure object [**?**] at current enviroment.

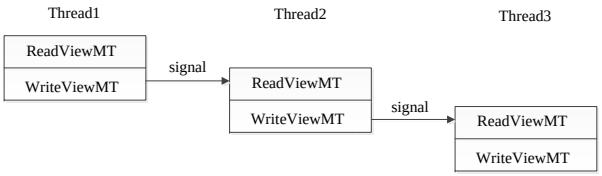

Fig. 5. ViewMT in pipelining

A. buiding block

Implemenation of building blocks are trivial. We use recursive calls to support nest. *seq* and *par* are interoperatable because we chose proper nested class before calling. It is worthy noting that building blocks are level awareless. The function object or cloure object need to be wrapped by loop-variable handlers. The handlers take responsibility for calculating loop variable in normalized form. It is only desirable for nest loop forms, e.g. Line.53 in List. **??**.*par* embeds OpenMP directive to run in parallel on CPU. We don't implement GPU counterpart because we can not see any benefits.

B. TF class

1) TF_hierarchy: We utilize predicate similar to merge [8] to generate subtask hierarchically. The major difference is that our predicate is *metafunction* and is evaluated at template parameter declaration at place(Line.4).

```
1template <class TASK
2  class ARG0, class ARG1, class RESULT,
3  template<class, class>  class PRED,
4  bool SENTINEL = PRED<ARG0, ARG1>::value>
5struct TF_hierarchy{...}
6
7template <class TASK
8  class ARG0, class ARG1, class RESULT,
9  template<class, class>  class PRED>
10struct
11TF_hierarchy<TASK,ARG0,ARG1,RESULT,true>
12{...};
```

2) TF_pipeline: We implement the TF class using variadic template in C++0x. The simplest implementation is listed as follows. It supports arbitrary number of function, only limited by compiler's the maximal level of template recursion. For C++ compilers don't support variadic template, there are workarounds to achieve the same effect, but quite tedious

```
1 template <class P, typename... Tail>
2  struct pipeline<P, Tail...> {
3    typedef typename P::input_type in_t;
4    typedef typename P::output_type out_t;
5
6    static out_t doit(in_t in)
```

```
7     {
8      pipeline<Tail...>::doit( P::doit(in) );
9     }
10  };
```

VI. Experiments and evalution

A. Methodology

We implement our library in standard C++[4], [5]. Theoretically, any standard-compliance C++ compiler should process our classes without trouble. New C++ standard (a.k.a C++0x)[5] 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.

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

TABLE II
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

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.

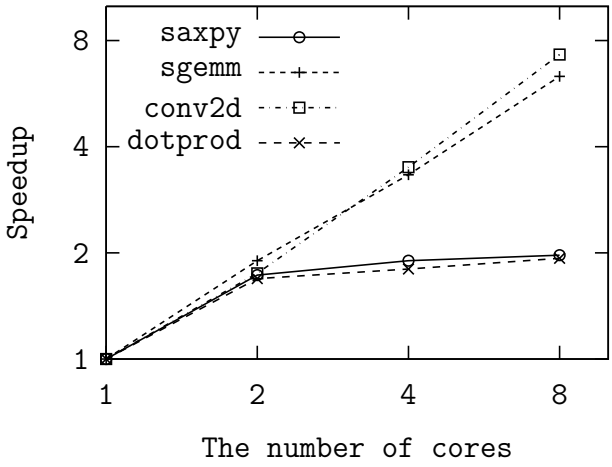

Fig. 6. Speedup on Harpertown

B. Evaluation

1) Speedup of Hierarchical transformation on CPU: Fig. 6 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 plain transformation on GPU: Fig. 7 shows SPMD transformation results for GPU on macbookpro. Programs run on host CPU in sequence as baseline. Embedded GPU on motherboard contains 2 SMs[1]. Porting from CPU to GPU, developer only need a couple of lines to change

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

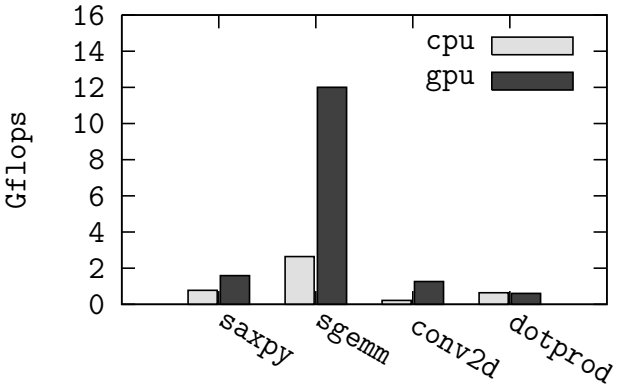

Fig. 7. Speedup Comparing GPU with CPU

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 III
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. III 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 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 [9] 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.

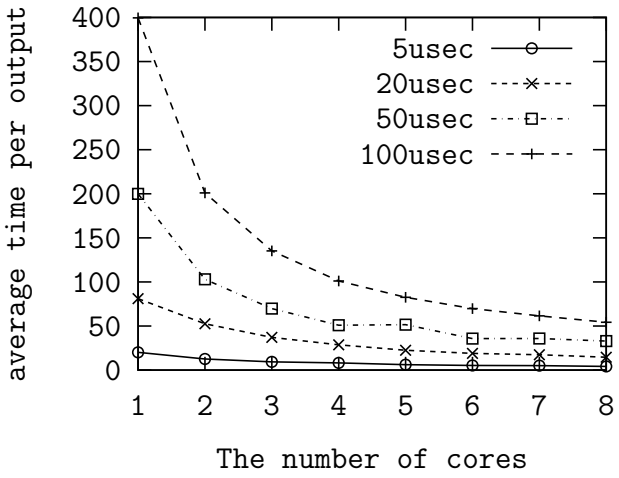

Fig. 8. Pipeline Processing for Psuedo Language Translation

4) Pipeline Transformation for CPU: Fig. 8 demonstrates pipeline processing using our template library. As described before, *langpipe* simulates a multilingual scenario. We apply template *TF_pipeline* listed in Fig. **??**. 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. 5. 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.

[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.

[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

The runtime behavior presumably incurs extra overhead. Many cores scenario helps alleviate the situation and render regular pipeline processing.

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 [10]. C++ community intend to develop parallel library while bearing generic programming in mind. TBB [11] 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 [8] 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] "So/iec (2003). iso/iec 14882:2003(e): Programming languages - c++," 2003.

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

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

[7] 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.

[8] 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.

[9] K. Fatahalian and M. Houston, "Gpus: A closer look," *Queue*, vol. 6, no. 2, pp. 18–28, 2008.

[10] H.-J. Boehm, "Threads cannot be implemented as a library," in *PLDI*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 261–268.

[11] Intel thread building blocks reference manual.