A Template Metaprogramming Approach to Support Parallel Programs for Multicores

Xin Liu, Minyi Guo, Daqiang Zhang, Jingyu Zhou, Yao Shen
Department of Computer Science
Shanghai Jiao Tong University
No. 800, Dongchuan Road, Shanghai, P.R.China
navyliu@sjtu.edu.cn

***Abstract*—In advent of multicore era, plain C/C++ programming languages can not fully reflect computer architectures. Source-to-source transformations help tailor programs close to contemporary hardwares. In this paper, we propose template-based approach to perform transformations for programs with rich static information. We present C++ template metaprogramming techniques to conduct parallelization for specific multicores. Parallel patterns and executions are provided in the form of template classes and oraganized as library. We implement a prototype template library – Libvina, to demonstrate the idea. It enables programmers to utilize new architectural features and add parallelization strategies by extending template library. Finally, we evaluate our template library on commodity x86 and GPU platforms by a variety of typical procedures in multimedia and scientific fields. In experiments, we show that our approach is flexible to support multiple parallel models and capable of transforming sequential codes to parallel equivalences according to specific multicore architectures. Moreover, the cost of programmability using our approach to adapt to more than one multicore is manageable.**

I. Introduction

Modern computer architectures rely on parallelism and memory hierarchy to improve performance. Both duplicated processors and elaborated storage-on-chip require programmers to be aware of underlying machines when they write programs. Even worse, multicore technologies have brought many architectural features for different implementations. Thus, it is challenging to develop efficient applications which can take advantage of various multicores.

In essence, it is because plain C/C++ programming languages can not reflect contemporary architectures [1]. Traditionally, programmers describe algorithms in sequential logics, and then resort to compiler and hardware optimization to deliver modest performance relative to their machines. In multicore era, this classic programming model gains little. It is desirable to develop alternatives to utilize horsepower of multicores while hiding architectural features.

Although researches on revolutionary programming models have obtained fruitful achievements, they are limited in specific domains [2], [3], [4]. One critical issue hinders them from applying in general programming field is that one programming model can only benefit a small group of users. It is still unclear what general purpose programming model is. Besides, the cost hardware usually weights a small part in a computer system relative to software and personnel. The ratio lowers with time. Therefore, vendors are reluctant to adopt fundamental changes of software stacks for multicore evolvement.

An acceptable tradeoff is to extend traditional programming languages to utilize effective parallel patterns. Appearently, the advantage of this approach is that it can exploit multicores progressively. Thus the knowledges and experiences of traditional programmers are still useful, and investment of legacy software is saved. In industry, OpenMP [5] and TBB [6] are successful cases. OpenMP provides parallel programming API in the form of compiler directives. TBB is a C++ library, consisting of concurrent containers and iterators. CUDA [7] extends C programming languate to describe groups of threads for GPU. The limitation of preceded approaches are platform or vendor dependent. In academia, Sequoia [8] attempts to programming for memory hierarchy. It achieves parallelization by divide a task into subtasks hierarchically and then map subtasks on nodes of machines. Merge [9] implements map/reduce programming model for heterogeneous multicores. Streamit [10] compiler supports stream/kernel model for streaming computation. Its run-time schedules kernels for specific architectures. The shortcoming of academical approaches is that each one is capable of one type of parallel patterns. In a word, existing solutions lack uniform method to express multiple parallel patterns across various multicores.

Observably, except TBB is a pure library-based solution, aforementioned approaches need compilers to facilite their programming models. It is the ad-hoc approaches embedded into compilers restrict flexibility and extensibility. Therefore, we propose a template-based programming model to support parallel programs for multicores. We exploit C++ metaprogramming techniques to perform source-to-source transformation in the unit of functions. We use *tasks* to abstract computation-intensive and side-effect free functions, which are candidates for transformations. We extend the meaning of template specialization [11]. Our approach specializes a task for target's architectures. Through applying template classes, a task is transformed into many subtasks according to different parallel patterns, and then subtasks are executed in the form of threads. Template classes are implemented for different multicore architectures. As a result, porting software from one platform to another only needs to adjust template parameters or change implementations of template classes. The difference between TBB and our approach is that we utilize C++ template metaprogramming, so the transformations complete at compile time.

Our approach is flexible and extensible. Both parallel patterns and execution models are provided as template classes, thus programmers can parallelize tasks using more than one way. In addition, template classes are organized as template library. It is possible to exploit architectural features and new parallelization strategies by extending library. We explore language features limited in ISO standard C++ [12], [13], [14], so it is applicable for platforms with standard-compilant compilers. Most platform-independent template classes can be reused. The limitation of template-based approach is that using template metaprogramming, only compile-time information is available. That includes static constant values, constant expression and type information in C++. Therefore, our approach is not a general solution and orients for programs with rich static information. Fortunately, it is not uncommon that this restriction is satisfied in the fields like embedded applications and scientific computation. Because the runtime of those programs with fixed parameters are significantly longer than compile time even the time to write programs, it will pay off if can resolve transformations at compile time. Besides, it is possible to utilize external tuning framework [15] to adjust parameters of static programs.

In summary, we proposed a template-based programming model, which tailors programs to multicores. Programers apply template classes to transform functions into the parallel equivalences on source-level, and then map them on specific multicores to run simultaneously.

The remaining parts of this paper are structured as follows. Section II presents our programming model. Section **??** introduces libvina – a prototype library to facilitate template-based programming model. Section III is how programers adapt their source codes to libvina. Section IV gives details of implemetation of our library. Section V evaluates performance on both CPU and GPU using our approach. Section VI summarizes related work to support parallel programs for multicores. Section VII is disscussion and future work.

II. Template-based Programming model

We propose a template metaprogramming approach to support parallel programms running on different multi-core systems. Fig. 1 gives an overview of our approach: a side-effect free function is abstracted as a *task* and is wrapped in a *function wrapper*. We provide wrappers to support different parallel patterns:

- **Hierarchy pattern.** A task is recursively divided into subtasks and subtasks can execute on multiple cores in parallel.
- **Pipeline pattern.** A task is divided into a series of processing subtasks, where the output of a previous subtask is directly used as the input of the next subtask and each subtask execute on a core.

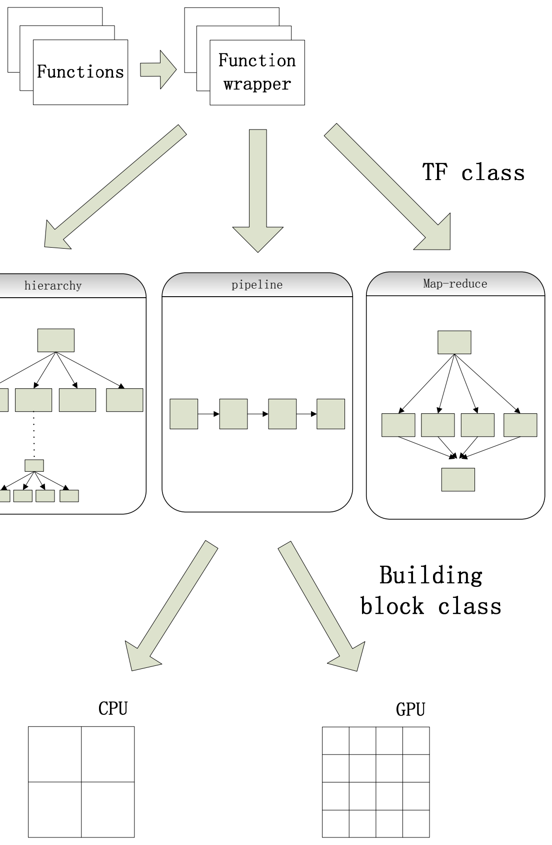


Fig. 1. Overview of template-based programming model: Programmers write side-effect free functions in C/C++, then encapsulate them in function wrappers. Template library regards a function wrapper as a task, which is automatically transformed into a group of subtasks based on appropriate parallel patterns. Finally, subtasks directly run on physical multicores.

- **Map-Reduce pattern.** A task is divided into a map phase and a reduce phase, where subtasks in each phase can execute in parallel on multiple cores. The input of reduce phase is the output of the map phase.

The transformation from a function wrapper to parallel patterns is a source-to-source conversion with C++ templates, which are called *TF class* in this paper (Section II-A). The converted code calls architecture-specific building block classes (Section II-C) so that when compiled, a task can execute on different multi-core systems in parallel at run time.

The rest of this section first describes three class types of our approach, i.e., TF, view, and building block classes. To illustrate the usage of our template library, we then describe two examples using these three types of classes.

A. TF Class

A *TF class* (short name for *TransFormation class*) is a template class representing a parallel pattern which transforms a task to a group of subtasks in isomorphism. In other words, the transformed task has the same interface while owns a call graph inside to complete the original computation by a group of subtasks. Specifically, the following two classes are used in this paper:

- **TF_hierarchy**. This template class recursively divides a task into subtasks until certain predicate is evaluated as true. In fact, TF_hierarchy can implement a programming model similar to Sequoia and we use TF_hierarchy to implement both hierarchy and Map-Reduce pattern.
- **TF_pipeline**. This template class synthesizes a call chain of an arbirary number of functions into a pipeline. This is a common pattern for stream kernel programming model.

In this paper, we consider a class of computation-intensive functions (or tasks) that are self-contained, i.e., external data references are limited and calling graphs of them are simple. For these functions, it's possible to decouple a task into a cluster of subtasks. These subtasks are identical except for arguments and we can distribute subtasks on multi-core to execute simultaneously.

B. View class

A *View* is a class representing a subset of container's data. For example, a matrix type could have views that contains a subset of elements of a matrix. There are two kinds of views, *ReadView* and *WriteView*. A ReadView is read-only, while a WriteView allows write operations on its data (by providing interfaces to write).

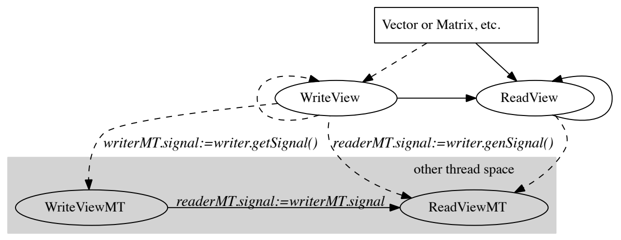
To ensure multi-thread safety, *ReadViewMT* and *WriteViewMT* are defined. For these two types, each object contains a signal that is copied across multiple threads. All operations of ReadViewMT or WriteViewMT are blocked until the current thread is signaled by other threads.

Fig. 2 depicts the relationship of views. Concrete lines means that a type cast from a source node to a destination node is legal, i.e., an implicit conversion in C++. Dashed lines means a source node can generate objects of the type of the destination node. Labels on the dashed line signifies how signals are created or copied. Shadow region is another thread space.

The design of view class has two purposes. 1) The classes are type-safed. Because template instantiation is not visible for programmers, our source transformations by templates could introduce subtle errors. We expect compilers complain explicitly when unintentional transformations happen. 2) View classes hide communication details. Implementations have choice to optimize data movement according to architectures. Shared memory systems [16] and communication-exposed multicores [17], [18] usually have different strategies to perform the operations.

C. Buiding block class

A building block class is a high-level abstraction of execution patterns that hide architecture-specific details. Programmers can use building blocks to execute tasks on different multi-core platforms. Table I lists building blocks in libvina. For tasks that can executed in SPMD (Single-Program-Multiple-Data) fasion, `par` class spawns multiple threads to execute subtasks in parallel. `seq` class supports pipeline



Vector or Matrix, etc.
WriteView
ReadView
writerMT.signal:=writer.getSignal()
readerMT.signal:=writer.genSignal()
other thread space
WriteViewMT
readerMT.signal:=writerMT.signal
ReadViewMT

Fig. 2. The relationship of view classes. A concrete line represents a valid type cast, while a dashed line represents a source node can generate objects of the type pointed to. Labels specify how signals are created or copied across views.

TABLE I
BUILD BLOCK CLASSES IN LIBVINA

Name	Semantics	Usage Example
par<T, K, F>	Iterate function F K times in parallel, implicit barrier	par<_tail, 4, F> ::apply();
seq<T, K, F>	Iterate function F K times	seq<_tail, 5, F> ::apply();
reduce<K, F>	Reduce K values using function F	reduce<8, F> ::apply(values)
mt::thread<F>	Spawn a thread to execute function F	mt::thread<F> ::apply();

patterns. Parameter T in `par` or `seq` could be nested `par`, nested `seq`, or `_tail` (meaning no further nestings). For instance, we can write the following statement

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

to build to a two-level loop, and the nested loop are executed in parallel. The equivalent OpenMP code is:

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

The `reduce` class supports the "reduce" pattern in MapReduce style tasks. Specifically, a given function F is used to reduce K input values and the final result is stored in the first value. Note that many of the $K-1$ times reduce operations are executed in parallel using multiple threads.

Finally, `mt::thread` class provides a thread interface for programmers to dynamically spawn a new thread to execute a function.

D. Example I: Hierarchy Pattern

Fig. 3 is the source code for matrix multiplication by adapting `TF_hierarchy` to implement a *Divide-and-Conquer* algorithm. Function `innner` at line 20 divides task into subtasks, while function `leaf` at line 45 performs computation. Call operator function at line 14 is the user interface for the task. Line 24∼37 is lambda expression to perform map/reduce.

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

```
1   template <class T, int M, int P, int N
2           template <class, class>
3           class PRED/*predicate*/
4           int K/*param to divide task*/>
5   struct SGEMM {
6   typedef ReadView<T, M, P>  ARG0;
7   typedef ReadView<T, P, N>  ARG1;
8   typedef WriteView<T, M, N> RESULT;
9
10  typedef SGEMM<T, M, P, N, PRED, K> SELF;
11  typedef TF_hierarchy<SELF, PRED> TF;
12
13  void //interface for programmer
14  operator()(const Matrix<T, M, P>& A,
15             const Matrix<T, P, N>& B,
16             Matrix<T, M, N>& C)
17  {
18     TF::doit(A, B, C.subViewW());
19  }
20
21  static void //static entry for TF
22  inner(ARG0 A, ARG1 B, RESULT C) {
23     //define SubTask here
24     typedef SGEMM<T, M/K, P/K, N/K, PRED, K>
25     SubTask;
26     typedef typename SubTask::TF SubTF;
27     typedef Matrix<T, M/K, N/K> SubMatrix;
28
29     //lambda for iteration
30     auto subtask = [&](int i, int j)
31     {
32       SubMatrix tmps[K];
33       //lambda for division
34       auto m = [&](int k) {
35         tmps[k].zero(); //initialize
36         SubTF::doit(
37            A.template SubViewR<M/K,P/K>(i, k),
38            B.template SubViewR<P/K,N/K>(k, j),
39            tmps[k].SubViewW(i, j));
40       };
41       //perform k operations
42       par<par_tail, K, decltype(m)&>
43       ::apply(m);
44       //sum up temporaries to submatrix of C.
45       reduce<K, plus<SubMatrix>>
46       (tmps, C.template<M/K, N/K>SubViewW(i, j));
47    };
48     //calculate each submatrix in parallel
49     typedef decltype(subtask)& closure_t;
50     par< par<par_tail, K>, K, closure_t>
51     ::apply(par_lv_handler2(subtask));
52   }/*end func*/
53
54   static void //static entry for TF
55   leaf(ARG0 A, ARG1 B, RESULT C)
56   {
57     // compute matrix product directly
58     for (int i=0; i<M; ++i)
59       for (int j=0; j<N; ++j)
60         for (int k=0; k<P; ++k)
61           C[i][j]+=A[i][k]*B[k][j];
62   }
63  };
```

Fig. 3. Source code of matrix multiplication (class SGEMM) using hierarchy pattern.

Matrix class cantains 3 template paramters: type, the number of row, the number of column. A definition of Matrix is at line 26 of Fig. 3.

Line 30∼32 of Fig. 3 generate subviews by calling functions.

Fig. 4 illustrates ...

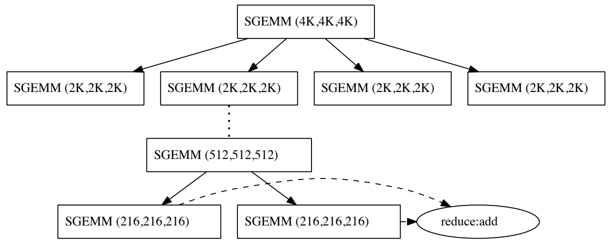


Fig. 4. An illustration of recursively dividing the task of matrix multiplication into subtasks. The original task is automatically divided into a number of subtasks at compile time. The division is implemented in Fig. 3. Here the parameter K is set to 2. As a result, each task is divided into 4 subtasks.

```
1   //template full specialization
2   template<>
3   struct TF_pipeline<>
4   {
5     //last stage defintions
6     //T* is the type of input
7     template<class T>
8     static void impl(T* in)
9     {
10      //omit...
11    }
12    template<class T>
13    static void
14    doit(T * in)
15    {
16      std::tr1::function<void (T*)>
17        func(&(impl<T>));
18
19      mt::thread_t thr(func, in);
20   }
21  };
22
23  //customize pipeline TF class
24  typedef TF_pipeline<
25    translate<Eng2Frn>,
26    translate<Frn2Spn>,
27    translate<Spn2Itn>,
28    translate<Itn2Chn>
29  > MYPIPE;
30
31  MYPIPE::doit(&input);
```

Fig. 5. Source code of a pipeline pattern (langpipe).

E. Example II: Pipeline Pattern

Fig. 10 gives pipeline processing example, which is similar to Streamit. It implements language translation pipeline by synthesizing a pipeline of four standalone functions. `TF_pipeline` is a TF class representing time-multiplex parallelism. As shown in examples, the parallel patterns and execution models are dramatically different, however, our approach can describing them well in uniform language constructs.

Our programming model facilitates the separation of roles in software development. Algorithm-centric programmers are only concerned of algorithm in conventional C/C++ form, as at line 45 of List 1 and line 8 of List 2. On the other side, system programmers knowing underlying architectures are in charge of developing and applying template classes to specialize tasks for the specific targets. This separation not only simplifies the difficulties of writing and tuning parallel programs, but also facilitates to develop effecient and portable programs for various multicores.

III. Adaption for Libvina

Programmers who apply our approach need to customize their source code to utilize libvina. Technically speaking, we provide a group of *concepts* in libvina to support transformations and expect programmrs to *model* our template classes [19].

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. Thus programmers have to bind their template functions to entries of classes. Either static function or call operator fuctions is approachable, but there is tradeoff to consider. Static function need to predefine naming convention. For example, `TF_hierarchy` use names `inner` and `leaf` to call back. Call operator has unique form to invoke, so we leave it as user interface, at expense of runtime cost [1]. Line 14 of Fig. 3 is the case.

B. Adaption for TF_hierarchy

Line 6∼10 of Fig. 3 is adaption for `TF_hierarchy`. Line 10 defines the type of task for `SGEMM`. It is used as the template parameter `TASK` for `TF_hierarchy` class. PRED template parameter at line 11 is a predicate and `TF_hierarchy` class will evaluate it using `ARG0` and `ARG1`. Line 18 calls customized TF class after dividing task. According to template argument, TF class determines whether reenter the entry inner at line 22 or terminate at leaf at line 45. Function leaf performs computation. Fig. 6 illustrates instantiation process of `TF_hierarchy` and Fig. 4 is execution after transformation. The figure depicts the case K is 2.

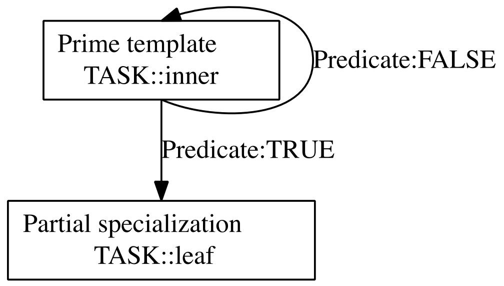


Fig. 6. Instantiation process of `TF_hierarchy`. The predicate is a template class, which is evaluted using `TASK`'s parameters.

[1]C++ does not allow overload call operator using static function, therefore we have to generate an object to call it.

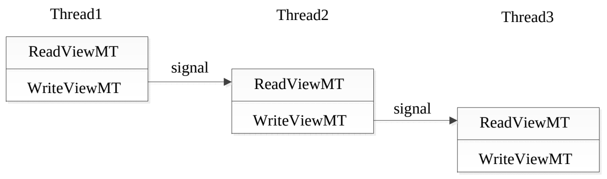


Fig. 7. Pipeline processing using ViewMTs. Access to a `ReadViewMT` is blocking until it is signaled. A stage sets its signal of `WriteViewMT` after data processing is complete.

C. Adaption for TF_pipeline

To leverage `TF_pipeline`, programmers have to provide a full specialization template class. This is because `TF_pipeline` only synthesizes functions and executes them in order, but does not know how to process the output. **TODO: A full specialization of TF_pipeline defines this behavior and is called at last. For example in Fig. 10, line 2∼21 is the case.** Static entry at line 13 serves `TF_pipeline` class. We spawn a thread to handle with the output from the previous stage. Line 24∼31 is a usage of `TF_pipeline` with 4 standalone functions. All the stages including our customized one are threads. It is noteworthy that each immediate stage, e.g., `translate<Frn2Spn>`, has to follow type interfaces and define dependences. In *lang_pipe* case, we use `ReadViewMT` and `WriteViewMT` to synchronize two adjacent stages. As Fig. 7 showing, `WriteViewMT` of a previous stage signals the `ReadViewMT` of the next stage that data is ready.

IV. Implementation

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 [20]. In order to utilize nested buiding blocks, lambda expression can generate closure objects in a concise form, e.g., line 24∼37 of Fig. 3.

A. Buiding Block Classes

Implemenation of building blocks are straightforward. We use recursive calls to support nesting patterns. `seq` and `par` are interoperable because we chose proper nested class before calling function `apply`. Note that building blocks do not know the nesting levels during the execution. To solve this problem, each function object or cloure object is decorated with a loop-variable counter. **TODO: Because some callable objects in C++ such as clousure object do not provide default constructors, we pass their references in those cases. Consequently, some call sites of building blocks are different from Table I.**

For CPU, building blocks are implemented by embedding OpenMP directives. On GPU, we bind building block classes to functions of OpenCL [21], an open standard API for heterogenous mulitcores.

B. TF class

`TF_hierarchy` has two template class definitions. The prime template calls back task's inner function, while the partial specialization calls leaf. We utilize predicate similar to Merge [9] to generate subtasks recursively. The main difference from Merge is that our predicate is a *metafunction* and is evaluated in place, e.g., line 3 below.

```
template <class TASK,
  template<class , class>  class PRED,
  bool SENTINEL = PRED<ARG0, ARG1>::value>
struct TF_hierarchy{...}

template <class TASK,
  template<class , class>  class PRED>
struct TF_hierarchy<TASK, true>
{...};
```

The `TF_pipeline` class using variadic template [22]. The simplified implementation is listed as follows, which supports an arbitrary number of functions. The only limitation is the maximal level of template recursions of a compiler.

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

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

V. Experiment

A. Methodology

We implement our library in ISO C++. Theoretically, any standard-compliant C++ compiler should process our classes without trouble. New C++ standard (a.k.a C++0x[14]) adds many language features to ease metaprogramming[2]. Compilers without C++0x support need some workarounds to pass compilation though, they do not hurt expressiveness. We develop the library and test using GCC 4.5 beta. The first implementation of OpenCL is shipped by Mac OSX 10.6, where we collect the date of GPU performance.

A couple of procedures are evaluated for our template approach. They are typical in multimedia and scientific fields. In addition, we implement a pseudo 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 product. 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.

[2]When we conducted this work, C++0x was close to finish. Iimplementing C++0x were in progress for many compilers

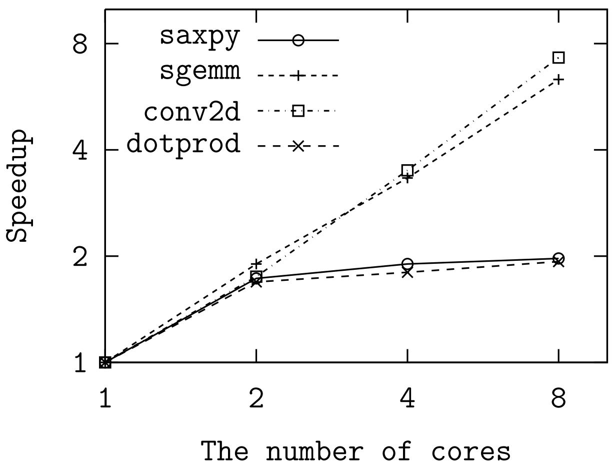


Fig. 8. Speedup of hierarchical transformation on Harpertown: We utilize TF_hierarchy class to divide tasks until they meet LLC.

- *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. On harpertown, we link Intel Math kernel to perform BLAS procedures if they are available. On macbookpro, we implement all the procedures on our own.

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

B. Evaluation

1) Speedup of Hierarchical transformation on CPU: Fig. 8 shows the speedup on harpertown. The blade server contains two quad-core Xeon processors. We experiment hierarchical transformation for algorithms. All predicates are set to cater to CPU's last level cache(LLC).

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. Note that we observe almost two-fold speedup from sequence to dual core case. But the speedup degrates to 3.3 times when the number of core continuously doubles. Harpertown consists of two quad-core processors, while Linux can not guarantee that 4 subtasks are distributed in a physical

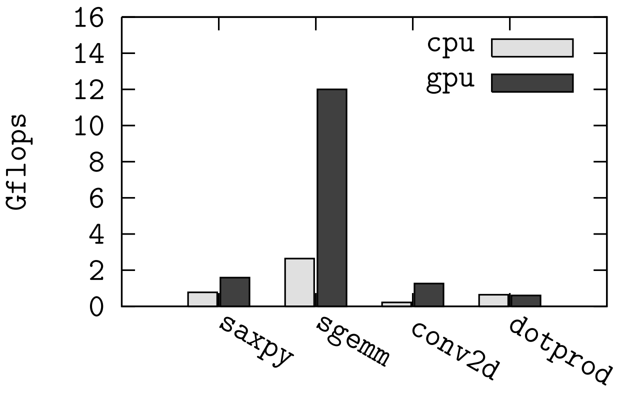


Fig. 9. Speedup Comparing GPU with CPU: We exploit the same set of template classes to transform tasks for different mulitcores

processor. Therefore, the cost of memory accesses and synchronization increases from 2-core to 4-core.

dotprod and *saxpy* show low speedups 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 our SMP system, even though we fully parallelize those algorithms by our template library.

2) Speedup of SPMD transformation on GPU: Fig. 9 shows SPMD transformation results for GPU on macbookpro. GPU's memory model has significantly different from GPU. Because TF_hierarchy makes little sense for GPU, we directly use building block par to translate iterations into OpenCL's NDRangeKernel function. Programs running on host CPU in sequence are set as baseline. Embedded GPU on motherboard contains 2 SMs[3]. Porting from CPU to GPU, developers only need to change template classes while keeping algorithms same [4]. 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 [23]. 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.

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.[5] 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 [24] can significantly ease vector programming. Programmer can implement algorithm in plain C and then replies on template transformation for GPU. Adapting to GPU only need tens of lines code efforts. Like GPU template, we apply building blocks directly to parallelize *sgemm* procedure for CPU. We observe 95.6 Gflops and about 75% effectiveness on harpertown server.

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

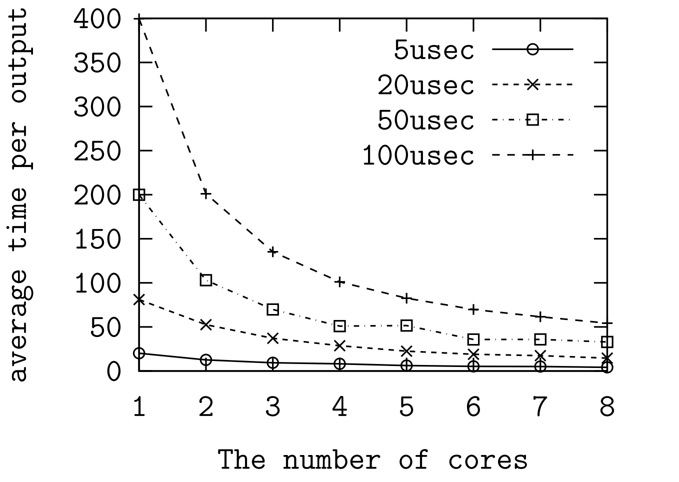


Fig. 10. Pipeline Processing for Psuedo Language Translation: improvement of 4-stage pipeline on CPU.

4) Pipeline Transformation for CPU: Fig. 10 demonstrates pipeline processing using our template library. As described before, *langpipe* simulates a multilingual scenario. We apply template TF_pipeline listed in List 2. In our case, the program consists of 4 stages, which can transitively translate English

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

[4]Because GPU code needs special qualifiers, we did modify kernel functions a little manually. Most algorithms are kept except for sgemm because it is not easy to work out sgemm for our laptop. We add blocking and SIMD instruments for CPU.

[5]$17.6Gflops = 1.1Ghz * 2(SM) * 8(SP)$. 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.

to Chinese[6]. Only its preceding stage completes, the thread is waked up and proceeds. The executing scenario is similar to Fig. 7. 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 ($20\mu s$, $50\mu s$, $100\mu s$), 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 fine-grained tasks contend CPU resources in out of the order, so the operations presumably incur extra overhead. Many cores scenario help alleviate the situation and render regular pipeline processing.

VI. Related work

Programming models to support parallel programs for muticores can be broadly categorized into directions:

1) providing library to support programming for parallelism.
2) extending language constructs to extend parallel semantics.

Library is a common method to extend language capability without modifying grammars. Pthread library is a *de facto* standard for multi-threading on POSIX-compatible systems. The relationship between pthread and native thread is straightforward. Therefore, abstractions of pthread are far away from expressing parallelism naturally. The same problem occurs on OpenCL or other vender-dependent libraries for GPUs. Libvina is a metaprogramming library instead of system library. We provides high-level parallel patterns and executions as template classes. Implementations take responsibility for binding tasks to threads on specific platforms. C++ community intends to develop parallel libraries while bearing generic programming in mind. TBB has a plenty of containers and building blocks to support loop-parallism and task-level parallelism. Inspired by TBB's approach, we enable the same effects in static domain. We aim at utilizing static information to perform source transformations for different architectures. Besides, template-based approach we propose is orthogonal to runtime parallel library TBB. We only explore parallelism which can be resolved by compilers, developers feel free to deploy TBB to farther improve programs.

The second direction for language community is to extend language constructs by modifying compiler. They add language constructs for compiler to express parallellism. OpenMP [5] compilers transform sequential code blocks into multi-threaded equivalences based on directives. OpenMP is *de facto* standard for shared memory though, the programming model does not fit heterogeneous mulitcores. Sequoia [8], [25] supports programming memory hierarchy. In order to achieve portability for parallel programs, a source-to-source compiler transforms a *task* into a cluster of *variants*, and then maps variants on tree-style virtual machines, which are described by external configuration files. We derive the same idea to choose implementations at compile time for different architectures. Merge [9] is a map/reduce programming framework for heterogeneous multicore systems in the forms of task and variant. It relies on hierarchical division of task and predicate-based dispatch system to assign subtasks on matched multicore target at runtime. Each approach mentioned above can complete one kind of parallel pattern. We demonstrate our template-based approach can achieve the same functionalities using template metaprogramming if parameters are available at compile time.

We intend to fuse the advantages of pure library approach and specialized parallel programming languages. Extending languages to express parallelism usually needs to modify compilers. We think it is this process hardwires fixed parallel patterns into the compilers. Therefore, we explore the powerness of metaprogramming to transform sources for parallelism, which can support mulitple parallel programming models while maintain portability for mulitcores.

VII. Discussion and Future work

We present a template metaprogramming approach to perform source-to-source transformations for programs with rich information. All functionalities are achieved within ISO C++ and organized as template library. The library is flexible enough to apply more than one parallel pattern and execution model. In addition, programmers can extend library to facilitate appropriate parallel patterns or new architectural features because template metaprogramming is intimate for C++ developers. Experiments show that our template approach can transform algorithms into SPMD threads with competitive performance. These transformations are available for both CPU and GPU, while the cost of migration is manageable. Besides, we can apply hierarchical division for programs on CPU. We also transform a group of standalone functions into a pipeline using our template library. It demonstrates that template metaprogramming is powerful enough to support more than one way to parallelize for multicore.

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.

Currently, kernel functions in GPUs prohibit recursion. We believe that it would be beneficial to introduce template recursion for GPUs. In addition, it is attractive for us to explore source transformations for strip-mined memory accesses in metaprogramming, because modern GPUs provide memory coalescence to optimize memory.

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

[6]follow the route: English $\rightarrow$ French $\rightarrow$ Spanish $\rightarrow$ Italian $\rightarrow$ Chinese

explores new TF classes to facilitate transforming source code close to target architectures using the static information.

References

[1] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, "A common machine language for grid-based architectures," *SIGARCH Comput. Archit. News*, vol. 30, no. 3, pp. 13–14, 2002.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] J. Armstrong, "The development of erlang," in *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1997, pp. 196–203.

[4] S. Peyton Jones, A. Gordon, and S. Finne, "Concurrent haskell," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 295–308.

[5] O. A. R. Board, "Openmp specificaiton version 3.0," 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[6] Intel. Intel thread building blocks reference manual. [Online]. Available: http://www.threadingbuildingblocks.org/documentation.php

[7] Nvidia. Cuda. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[8] 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, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 83.

[9] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008, pp. 287–296.

[10] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 179–196.

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

[12] "So/iec (1998). iso/iec 14882:1998(e): Programming languages - c++," 2003.

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

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

[15] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. J. Dally, "A tuning framework for software-managed memory hierarchies," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 280–291.

[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] 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.

[18] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," *Computer Design, International Conference on*, vol. 0, p. 282, 2002.

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

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

[21] A. Munshi, "The opencl specification version 1.0," 2009.

[22] 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.

[23] Nvidia. Nvidia opencl programming guide for the cuda. [Online]. Available: http://www.nvidia.com/content/cudazone/download/OpenCL/

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

[25] 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 '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2007, pp. 226–236.