# 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. 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 [1]. 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; investment of legacy software is saved. In industry, OpenMP [2] and TBB [3] 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 [4] extends C programming languate to describe groups of threads for GPU. The limitation of preceded approaches are platform or vendor dependent. In academia, Sequoia [5] 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 [6] implements map/reduce programming model for heterogeneous multicores. Streamit [7] 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 [8]. 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++ [9], [10], [11], 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 [12] 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. III introduces libvina – a prototype library to facilitate template-based programming model. Section. IV is how programers adapt their source codes to libvina. Section. V gives details of implemetation of our library. Section. VI evaluates performance on both CPU and GPU using our approach. Section. VII summarizes related work to support parallel programs for multicores. Section. VIII is disscussion and future work.

## II. Template-based Programming model

We use template metaprogramming to implement a parallel programming model. Essentially, our approach utilizes C++ template mechanism to perform source-to-source transformations for multicores. Side-effect free functions are abstracted as *tasks*. A task is wrapped in the form of template class, named *function wrapper*. A *TF class* is a template class, which is capable of transforming a task into a group of subtasks based on a parallel pattern. Tasks apply TF classes according to their appropriate parallel patterns. This process is called ast *adaption*. Finally, we use *building block classes* to define executions of tasks on specific architectures. Both TF classes and building blocks are organized as a library – **libvina**. Fig. 1 depicts the diagram of template library-based programming model. Conventional functions are encapsuated into function wrappers. After transformation at compile time, they are executed on different multicore architectures at run time.

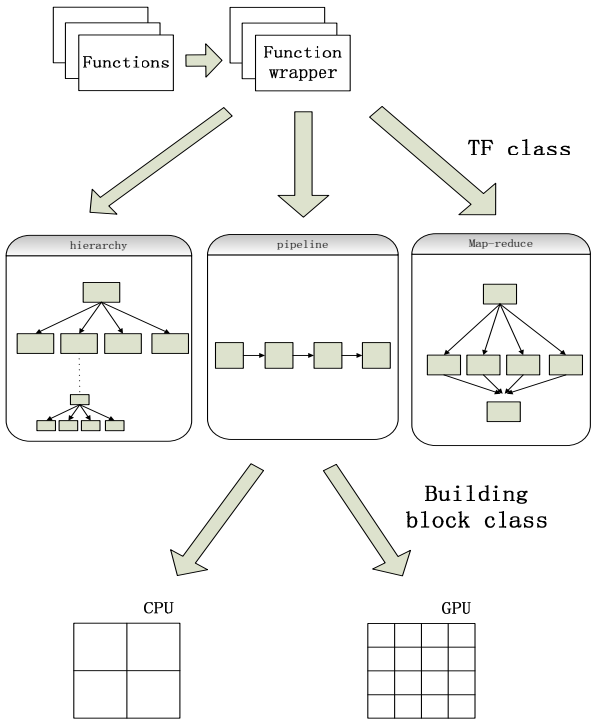


Fig. 1. Overview of template-based programming model: Programers write side-effect free functions in C/C++, then encapsulate them into function wrappers. Template library regards a function wrapper as a task. Tasks are transformed into a group of subtasks based on appropriate parallel patterns, finally map tasks on physical multicores.

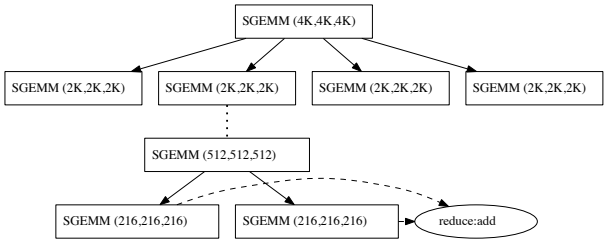


Fig. 2. Matrix-multiplication(sgemm) division:Divide matrix-multiplication task into smaller subtasks. The division process is implemented by List. 1. Triple in figure represents task parameters (M, P, N), which means A[M][P] * B[P][N]. The figure is the result of parameterizing K = 2.

```
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  tyspedef 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       //lambda for iteration
24       auto subtask = [&](int i, int j)
25       {
26         Matrix<T, M/K, N/K> tmps[K];
27         //lambda for map
28         auto m = [&](int k) {
29           TF::doit(
30              A.SubViewR<M/K,P/K>(i, k),
31              B.SubViewR<P/K,N/K>(k, j),
32              tmps[k].SubViewW(i, j));
33         };
34         par<par_tail, K, decltype(m)&>
35         ::apply(m);
36         reduce<K>(tmps, C[i][j]);
37      };
38
39       typedef decltype(subtask)& closure_t;
40       par< par<par_tail, K>, K, closure_t>
41       ::apply(par_lv_handler2(subtask));
42     }/*end func*/
43
44     static void //static entry for TF
45     leaf(ARG0 A, ARG1 B, RESULT C)
46     {
47       // compute matrix product directly
48       for (int i=0; i<M; ++i)
49         for (int j=0; j<N; ++j)
50           for (int k=0; k<P; ++k)
51             C[i][j]+=A[i][k]*B[k][j];
52     }
53    };
```

List. 1 Example code of sgemm: SGEMM class adapts TF_hierarchy to implement *Divide-and-Conquer* algorithm of matrix multiplication. 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, corresponding to SGEMM(512, 512, 512) node in Fig. 2

```
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);
```

List. 2 Example code of langpipe: A translation is a standalone function wrapper. TF class synthesizes a pipeline.

Programmers using our template-based programming model are free to choose ways to parallelize tasks. An example applying Sequoia's programming model is shown in Fig. 2. *sgemm* is a task to perform matrix-mulitplication. We can apply a TF class dedicated to hierarchical division. List. 1 illustrates the adaption. As a reuslt, we implement the straightforward *Divide-and-Conquer* algorithm for sgemm, which divides a matrix into K*K submatrices, computes them recursively, and reduces the results for each division. The control flow of source transformation is programmed using template metaprogramming inside of the TF class. To demonstrate the more than one way of parallelization can be achieved in our programming model, List. 2 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. LIBVINA: A TEMPLATE LIBRARY

We implement a prototype template library, libvina, to demonstrate our approach. Libvina consists of 3 components: (1) View class, a representation of underlying containers such as vector or matrix. (2) Building block class, provide basic executions of tasks on multicores (3) TF class, each one represents a parallel pattern.

### A. View class

To leverage static information, libvina need to associate template parameters with ADTs' parameters. For example, Matrix class cantains 3 template paramters: type, the number of row, the number of column. A definition of Matrix is at line.26 of List. 1. A *View* is a class representing the subset of containers' data. There are two kinds of views: ReadView and WriteView. Variants like ViewMTs serve for multithreaded programs. A ReadView is read-only. A WriteView has interfaces to write as well. ViewMTs contains signals, which are copied across

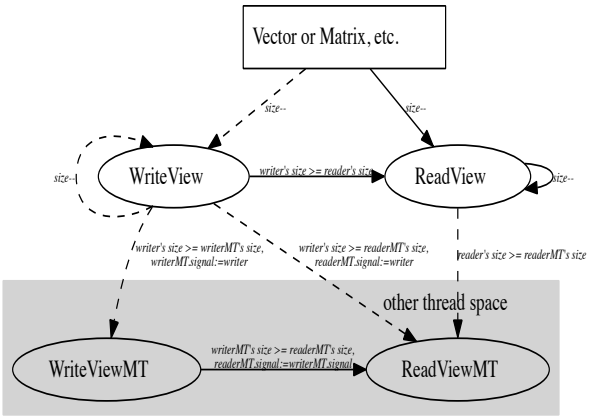


Fig. 3. View classes in libvina: A view represents subset of underlying constrainers. A ReadView is read-only. A WriteView can write back as well. Edges are conversions. A concrete line is implicit conversion from head to tail, while a dashed line represents a conversion needs explicit function call. ViewMTs are used for multithreaded programs. A signal in ViewMT is a handler of dependence.

multiple threads. All operations of ViewMTs are blocking until signals are sent by other views.

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 for conversions. Line.30∼32 of List. 1 generate subviews by calling functions. 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 [13] and communication-exposed multicores [14], [15] usually have different strategies to perform the operations.

### B. Buiding block class

A building block class is a high-level abstraction of execution. Programmers utilize building blocks to execute tasks on multicores. Table. I lists building blocks we implement in libvina. To parallelize programs, we expect most tasks are executed in SPMD (Single-Program-Multiple-Data). However, if it is not the case, we have to deal with dependences carefully using *seq* and *reduce*. At last, we provide thread interface using mt::thread class. Programmers can exploit it to bind thread directly (*e.g.* line.19 of List. 2) or develop other customized building blocks.

Like traditional programming languages, our building blocks of iterations support nesting definition. In addition, both *seq* and *par* are interoperable. *i.e.* we can write statement like

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

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

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

The first template parameter T of iterations is used to support nest. It could be either a par or a seq. Special classes *par_tail* and *seq_tail* are symbols to indicate the end of nest.

TABLE I
BUILD BLOCKS IN LIBVINA

| Name | Semantics | Example |
|---|---|---|
| **seq<T, K, F>** | Iterate function *F* *K* times | seq<seq_tail, 5, F> ::apply(); |
| **par<T, K, F>** | Iterate function *F* *K* times in parallel, implicit barrier | par<par_tail, 4, F> ::apply(); |
| **reduce<K, F>** | Reduce *K* values using function *F* | reduce<8, F> ::apply(values) |
| **mt::thread<F>** | Eexecute function *F* in a thread | mt::thread<F> ::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. As a rule of thumb, computation-intensive functions are usually self-contained, *i.e.* external data references are limited and calling graphs of them 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 subtasks on multicore to execute simultaneously. Another approach is to divide a complicated task into finer stages and run in pipeline manner to respect data locality and bandwidth. Two examples mentioned before follow the two patterns respectively. A *TF class* is a template class representing a parallel pattern which transforms a task to a group of subtasks in isomorphism. *i.e.* the transformed task has the same interface while owns a call graph inside to complete the original computation by a group of subtasks.

We implement two TF classes in libvina though, it is not necessary to use TF classes to perform source transformations. We encourage to do so because it has engineering advantages, which reduces effects of system programmers.

- TF_hierarchy It will recursively divide task into subtasks until predicate is evaluated as true. As Fig. 2 depicted, we use TF_hierarchy to implement programming model like Sequoia.
- TF_pipeline Inputing an arbirary number of functions, the template class can synthesize a call chain. This is a common pattern for stream/kernel programming model.

## IV. 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 [16].

### 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 though, there is tradeoff to consider. Static function need to predefine naming convention. *e.g.* 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 List. 1 is the case.

### B. Adaption for TF_hierarchy

Line.6∼10 of List. 1 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. 4 illustrates instantiation process of TF_hierarchy and Fig. 2 is execution after transformation. The figure depicts the case K is 2.

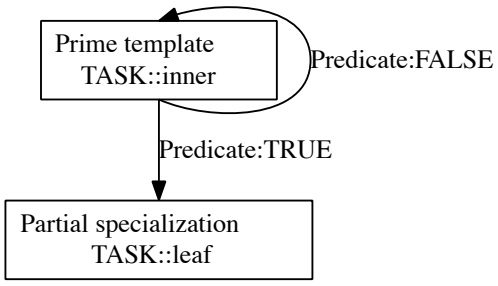


Fig. 4. Instantiation process of TF_hierarchy: The predicate is a template class, which is evaluted using TASK' parameters.

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

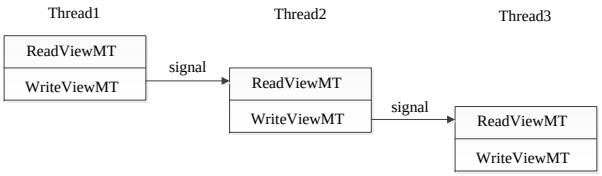


Fig. 5. Pipeline process using ViewMTs: Access of a ViewMT is blocking until it is signaled. A stage sets its signal of WriteViewMT after processing.

### C. Adaption for TF_pipeline

To leverage TF_pipeline, programmers have to provide a full specialization template class for it. This is because TF_pipeline only synthesizes functions and executes them in order. 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 *langpipe* example, line.2∼21 is the case. Static entry at line.13 serves TF_pipeline class. We spawn a thread to handle with the output of precious last 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 utilize our ViewMT despicted in Fig. 3. Asynchronous signals in ViewMTs provokes waiting stages and are used to mimic data-flow diagram.. Fig. 5 illustrates the scenario contains three 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 [17]. In order to utilize nested buiding blocks, lambda expression can generate closure objects in a concise form. *e.g.* line.24∼37 of List. 1.

### A. buiding block

Implemenation of building blocks are trivial. We use recursive calls to support nest. *seq* and *par* are interoperable because we chose proper nested class before calling function *apply*. Note that building blocks are level-free in terms of iteration, Thus function objects or cloure objects need to be decorated by loop-variable handlers. The handlers take responsibility for calculating loop variables in normalized form. It is only desirable for nested loop forms, *e.g.* line.41 of List. 1. Due to the fact that some callable objects in C++ such as clousure object do not provide default constructors, we pass their references in those cases. Consequential, some callsites of building blocks are different from Table. I. In terms of implementation, building blocks on CPU embed OpenMP directive to run in parallel. On GPU, we bind them to function of OpenCL [18], which is a open standard API for heterogenous muliticores.

### B. TF class

*1) TF_hierarchy:* 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 [6] to generate subtasks recursively. The major difference from merge is that our predicate is *metafunction* and is evaluated at place (*e.g.* line.3 below).

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

*2) TF_pipeline:* We implement the TF class using variadic template [19]. The simplified implementation is listed as follows. It supports an arbitrary number of functions, only limited by compiler's the maximal level of template recursion.

```
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. 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[11]) 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

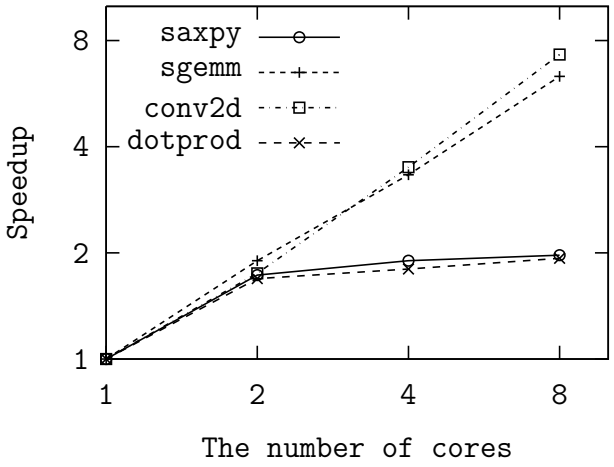


Fig. 6. Speedup of hierarchical transformation on Harpertown: We utilize TF_hierarchy class to divide tasks until they meet LLC.

- *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<br>GPU 9400m 1.1Ghz | 2G<br>256M | Mac OSX Snowleopard |

### 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 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

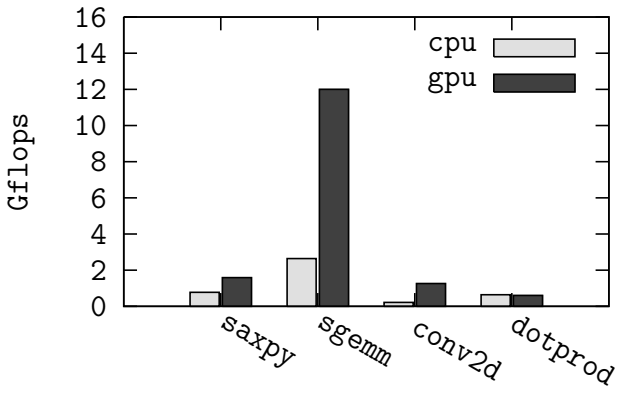


Fig. 7. Speedup Comparing GPU with CPU: We exploit the same set of template classes to transform tasks for different mulitcores

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. 7 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. 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 [20] 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 |

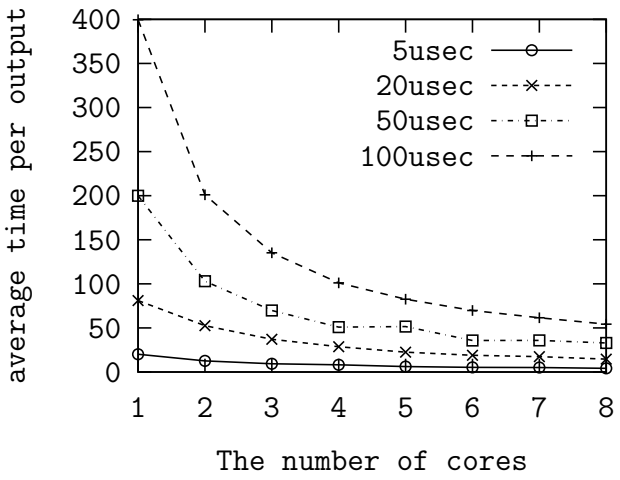


Fig. 8. Pipeline Processing for Psuedo Language Translation: improvement of 4-stage pipeline on CPU.

*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 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. 5. We use bogus loop to consume $t$ $\mu s$ 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 $t$ $\mu s$. The speedup is easy to maintain when granularity is big. 100 $\mu s$ case ends up 54 $\mu s$ for each instance for 8 cores. 50 $\mu s$ case bumps at 5 cores and then improves slowly along core increment. 20 $\mu s$ case also holds the trend of first two cases. 5 $\mu s$ case is particular. We can not observe ideal pipelining until all 8 cores are available. Our Linux kernel scheduler's granularity is 80 $\mu s$ 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.

## VII. 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 [2] 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 [5], [21] 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 [6] 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.

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

## VIII. 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

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

## REFERENCES


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

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

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

[4] NVidia. Cuda. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

[21] 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.