# 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 template-based approach to perform transformation for programs with rich static information. We present C++ template metaprogramming techniques to conduct parallelization for specific multicores. Parallel pattern and execution model 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 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 to more than one multicore platform 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. More worse, multicore technologies have brought various 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 language 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 gain 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 to obstacle them applying to general programming field is that one programming model can only benefits a small group of users. It is still unclear what general purpose programming model is. Besides, hardware cost usually weights small in the whole 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 softwares are 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. 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 are lack of 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 library-based programming model to support parallel programs for multicores. We exploit C++ metaprogramming techniques to perform source-to-source transformation in the unit of function. We use *task* to abstract computation-intensive and side-effect free function, which is a candidate for transformation. We extend the meaning of template specialization [8], which specializes 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 need to adjust template parameters or change implementation 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 sour approach is that using template metaprogramming, only compile-time information are 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 time of writing programs, it will pay off

../overview.pdf

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. Programers ulitize template library to transform a task into a group of subtasks, map task on physical multicores.

../mmexample.pdf

Fig. 2. Matrix-multiplication(sgemm) division:Divide matrix-multiplication task into smaller subtasks. The division process is implemented by source code 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.

if can resolve transformation 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 a function into parallel equivalence on source-level, and then

The remaining parts of this paper are structured as follows. Section 2 presents our programming model. Section 3 introduces libvina – a prototype library to facilitate template-based programming model, and how user adapt their code to libvina. Section 4 gives details of implemetation of our library. Audiences with C++ template programming experiences and functional programming language concepts are helpful but not prerequisites for these sections. Section 4 evaluates performance on both CPU and GPU using our approach. Section 5 summarizes some related works to support parallel programs for multicores. Section 6 is disscussion and conclusion.

## 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 transformation for multicores. Side-effect free functions are abstracted as *tasks*. A task is wrapped in the form of template class, named *function wrapper*. Tasks apply *TF classes* according to appropriate parallel patterns. This process is called as adaption. *TF classes* are able to manipulate tasks, which take responsibility for transforming a tasks into a group of subtasks. Each TF class represents a parallel pattern. Finally, we use *building blocks* to define execution models for specific multicore architectures. Both TF classes and Building blocks are template classes and 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, they are mapped on different multicore architectures to run in parallel.

```
template <class T, int M, int P, int N
        template <class, class>
        class PRED/*predicate*/
        int K,/*param to divide task*/>
struct SGEMM {
typedef ReadView<T, M, P>  ARG0;
typedef ReadView<T, P, N>  ARG1;
typedef WriteView<T, M, N> RESULT;

tyspedef SGEMM<T, M, P, N, PRED, K> SELF;
typedef TF_hierarchy<SELF, PRED> TF;

void //interface for programmer
operator()(const Matrix<T, M, P>& A,
           const Matrix<T, P, N>& B,
           Matrix<T, M, N>& C)
{
   TF::doit(A, B, C.SubViewW());
}

static void //static entry for TF
inner(ARG0 A, ARG1 B, RESULT C) {
   //lambda for iteration
   auto subtask = [&](int i, int j)
   {
     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 class to implement matrix-multiplication(sgemm task). *innner* at Line. 20 divides task into subtasks, while *leaf* at Line.45 performs computation. Call operator function at Line 14 is the user interface for the task. Line.24∼37 is lambda to perform map/reduce, corresponding to SGEMM(512, 512, 512) node in Fig. 2

```
//template full specialization
template<>
```

```
struct TF_pipeline<>
{
  //last stage defintions
  //T* is the type of input
  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);
```

---

List. 2. Example code of langpipe: translation<AtoB> is template function, which is capable of translating string from language A to language B. TF class transforms standalone functions into a pipeline.

Using our template-based programming model are free to choose ways to parallelize. An example using 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 illustrate the adaption at Line.11. Beside, we building blocks *par* and *reduce* to express execution. As a result, we implement the straightforward *Divide-and-Conquer* algorithm for matrix-multiplicaiton. which divides matrix into K*K submatrices to compute them recursively, and then reduces the results. The decomposition rule and termination of recursion 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 similar to Streamit. It implements language translation pipeline by synthesizing 4 functions. TF_pipeline is a TF class representing this time-multiplex parallelism. As shown in examples, the parallel patterns and execution models are dramatically differently, however, our approach can describing them well in uniform language constructs.

Our programming model facilitates to separate roles in software development. Algorithm-centric programmers only concern 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 known underlying architecture are in charge of developing and applying template class to specialize tasks for the concrete target. This separation not only solves the difficulties of writing and tuning parallel programs, but also facilitates programming model to develop effecient and portable parallel programs for multicores

## III. Libvina: A template library

We implement a prototype 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, provide basic iterations to execute tasks (3) TF class, each one represents a parallel pattern.

### *A. Data Structure*

To leverage static information, libvina need to associate template parameters with ADTs' parameters. For example, for Matrix class, it cantains 3 template paramters: type, the number of row, the number of column. A definition of Matrix is at Line.31 of List. 1.

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. Line.30$\sim$32 of List. 1 generate subviews by calling functions. 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 classes is to hide communication. Implementations have chance to optimize data movement according to architectures. *i.e.* shared memory systems [13] and communication-exposed multicores [14], [15] usually have different strategies. In addition, a view class is type-safed. Programmers can get compilation errors if they violate data access rules. Early errors are critical to prevent programmers from trapping into multithreaded bugs.

../view_concept.png

Fig. 3. View classes in libvina

### B. *Buiding Block*

Table. I lists a group of building blocks to execute tasks in libvina. To parallelize programs, we expect most tasks are executed in SPMD (Single-Program-Multiple-Thread). Therefore we define SPMD programming model as a building block. However, if it is not the case, we have to deal with dependences carefully to guarantee correctness.

Similar to constructs in traditional programming languages, our building blocks support nesting definition. Both *seq* and *par* is interoperable. *i.e.* we can define a block like

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

to build to level-2 loop, and the nested loop are executed in parallel. Its equivalence in OpenMP is 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);
}//implicit barrier
```

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, explicit barrier | par<par_tail, 4, F> ::apply(); |
| **reduce<K, F>** | reduce values using function $F$ | reduce<8, F> ::apply(values) |

### *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. It is not necessary to use TF classes to perform source transformations. We encourage to do so because it has engineering advantage, 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 similar to Sequoia.
- TF_pipeline Input arbirary number of functions, the template class can synthesize a call chain. Template parameter of class determines whether bind functions to threads. This is 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 programming 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. So programmers have to bind their template functions to entries of classes. Either static function or call operator fuctions is okay though, there is tradeoff to consider. Static function need to predefine naming convention. *e.g.* TF_hierarchy use names *inner* and *leaf*. Call operator has unique name to call, 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 are adaption for TF_hierarchy. The codes define the type of subtask for SGEMM at Line.10. It is used as TASK template parameter 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. leaf function 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.

../algo.pdf

Fig. 4. Instantiation process of TF_hierarchy

### C. Adaption for TF_pipeline

To leverage TF_pipeline, programmers have to provide a full specialization template class for it. It is because that TF_pipeline can only synthesize functions and execute them 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 *langpipe* example, Line.2∼21 is the case. Static entry at Line.13 is served for TF_pipeline template. 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 separated threads, as a result, the pipeline is nonblocking. 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 and WriteViewMT. The

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

../viewmt.pdf

Fig. 5. Pipelining functions using ViewMTs

first case represents initialization, while the second builds dependence transparently when type conversion occurs. We use signal mechanism to provoke downstreaming stages. Fig. 5 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 [17]. To utilize nested buiding blocks, lambda expression *e.g.* Line.24∼37 of List. 1 generates closure object [18] at current enviroment. If labmda is not available, implementations require non-straightforward codes using function object and bind.

### 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 noting that building blocks are level-free in terms of nest. The function object or cloure object need to be wrapped by loop-variable handlers. The handlers take responsibility for calculating loop variables in normalized form. It is only desirable for nest loop forms, e.g. Line.54 of List. 1. Because some function object such as clousure does not provide default constructor, we pass their references or right-value references. So the callsite is slight different from Table. I. Building block *par* embeds OpenMP directive to run in parallel on CPU. On GPU, we use OpenCL API [19].

### B. TF class

*1) TF_hierarchy:* We utilize predicate similar to merge [6] to generate subtask hierarchically. The major difference 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
```

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

*2) TF_pipeline:* We implement the TF class using variadic template [20]. 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.

```
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) );
  }
};
```

## VI. Experiments and evalution

### A. Methodology

We implement our library in ISO C++. Theoretically, any standard-compliance C++ compiler should process our classes without trouble. New C++ standard (a.k.a C++0x)[11] adds a lot of language features to ease metaprogramming[2]. Compilers without C++0x support 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. We developed the library and tested using GCC 4.5beta. 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 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 production. Each vector comprises 32 million elements.

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


Fig. 6. Speedup on Harpertown

*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<br>server | x86 quad-core<br>2-way 2.0Ghz | 4G | Linux Fedora<br>kernel 2.6.30 |
| macbookpro | laptop | x86 dual-core<br>2.63Ghz<br>GPU 9400m<br>1.1Ghz | 2G<br><br>256M | Mac OSX<br>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.

### *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

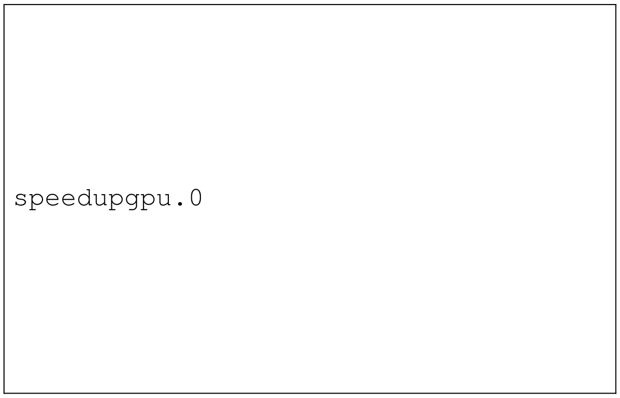


Fig. 7. Speedup Comparing GPU with CPU

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. 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 subtasks are executed within a physical processor. Therefore, the cost of memory accesses and synchronization increases from 2-core to 4-core platform.

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

*2) Speedup of SPMD transformation on GPU:* Fig. 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, developer only need a couple of lines to change templates 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

[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. 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.

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.[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 [21] 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.

*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[6]. 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 $t$ $\mu s$ 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

[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.

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

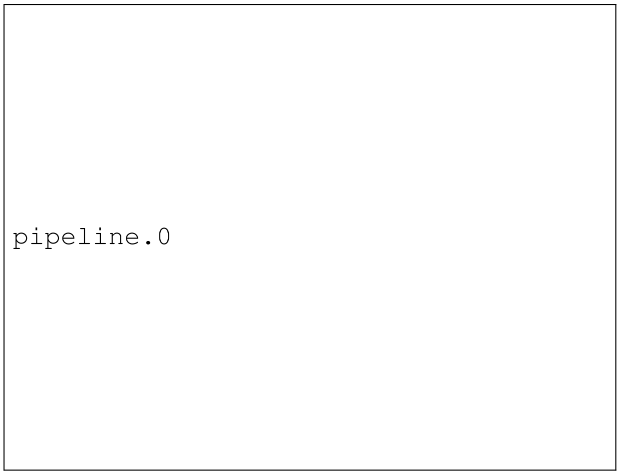


Fig. 8. Pipeline Processing for Psuedo Language Translation

scheduler's granularity is 80 $\mu s$ in default. We think that the very fine granular tasks contend CPU resources in out of the order. The runtime behavior presumably incurs extra overhead. Many cores scenario helps alleviate the situation and render regular pipeline processing.

## 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 [22]. C++ community intend to develop parallel library while bearing generic programming in mind. TBB has a plenty of containers and building blocks similar to the components in libvina. 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 [2] 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 little room for further improving performance or scaling up to larger systems. Hybrid OpenMP with MPI is possible though, difficulties surge. Sequoia [23] 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. [**?**] reports 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. Merge [6] features a uniform runtime environment for heterogeneous multicore systems in 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. However, Merge only supports *map-reduce* programming model. Methods mentioned before all need non-trivial efforts to modify compilers. As discussed in [5], 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 programmers and computer system developers. Conventional C/C++ programming language can not reflect hardware. Existing ad-hoc techniques or platform-dependent programming language pose issues of generality and portability.

We present a template metaprogramming approach to perform source-to-source transformation for programs with rich information. All functionalities are achieved within ISO C++ and organized as as template library. The library is flexible enough to apply more than one parallel pattern and execution model. In addition, our approach is extensible. Template metaprogramming is intimate for C++ programmers so they can extend the library to facilitate appropriate parallel patterns and new architectural features. 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. 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.

Streaming is an important computation model for innovative multicore architectures [15], [14], [13], [4]. We partially exploit GPU functionality in this paper, however, the transformations for GPU are quite straightforward. It is still unclear how many efforts need to pay for a full-blown template library, which supports streaming computation.

Currently, kernel functions in GPU prohibit recursion. We believe that it would be beneficial to introduce template recursion for GPU. 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] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] (2008) Openmp specificaiton version 3.0.

[3] Intel thread building blocks reference manual.

[4] Nvidia opencl programming guide for the cuda.

[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, "Memory - sequoia: programming the memory hierarchy," in *SC*. ACM Press, 2006, p. 83.

[6] 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.

[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," in *ICCD*. IEEE Computer Society, 2002, pp. 282–288.

[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] D. Vandevoorde, "So/iec n2927, new wording for c++0x lambdas," 2009.

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

[20] 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.

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

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

[23] 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.