Libvina: A Template Approach to Specialize sources for Multicore Architectures

Xin Liu

***Abstract*—We introduce libvina – a template library designed to specialize programs for different architectures. A source-to-source compilation is helpful to change programs more closer to contemporary multicore hardwares. We proposed a new approach to achieve this target for problems with rich static information. In libvina, parallelization and optimization of memory hierarchy are conducted by a groups of C++ template classes. It is flexible and extensible. Programmers with decent knowledge of template meta-programming are capable to extend library. Because template library effects in compiler time, minimal runtime cost is incurred from our approach. Finally, We evaluate the effects of specialization of some typical applications in multimedia and scientific computation on commodity x86 and GPU platforms.**

***Keywords*-static analysis; Compiler optimization; parallelization**

I. Introduction

In multicore period, hardware architects rely on parallelism and memory hierarchy to enhance performance. Both cloned processors and elaborated storage-on-chip require programmer to restructure their source code and keep tuning binaries for a specific target.Therefore, writing a high-performance application requires non-trivial knowledge of underlying machine's architecture. The gap between hardware vendors and software developers extends development cycle, which increases marketing cost and risks for innovative multicore architectures in silicon industry.

Algorithm experts usually focus on their specific domains and have limited insights on diverging computer systems. They expect hardware and optimized compiler to guarantee decent performance for their programs. The expectation was roughly held until massive parallelization was introduced to computer community. Since frequency of general purpose processor stops growing faster, it's hard for legacy software to obtain performance enhancement from hardware's refinement freely. Plain C/C++ can not fully reflect contemplate architecture any more. Researchers have admitted that it is tremendously challenging for optimizing sequential code by a compiler, so the answer to bridge programmer to parallel hardware relies on language and library to express concurrency richer than ever.

Designing new parallel programming is possible. Many functional languages [**?**] with inherent concurrency supports have emerged to the horizon of computer. However, a conservative programmer may turn them down because there are still lack of convincing evidences to demonstrate programmability and efficiency comparing to traditional programming languages. Another reason is software cost. Considering the time span which a large computer system serves, hardwares are cheap and become cheaper with time passing; software and personnel with well-trained programming knowledge are expensive. Even numerous legacy systems designed without consideration of multi-threaded environment at that time, vendors usually prefer to update and maintain software systems for current and future hardwares rather than restarting from scratch.

One side, software developers insist on classic programming diagrams and are reluctant to rewrite all the existing sources. On the other side, utilizing the powerhouse of modern processors to boost performance for existing and new systems is an important issue for them. Therefore, it is desirable to extend traditional programming languages to honor the trade-off.

Many programming languages extending popular programming languages had been proposed for multicore. However, most of existing solutions aimed at specific architectures or computer systems. *e.g.* CUDA only works on vender-dependent GPU architecture; UPC and OpenMP are designed for shared memory system [**?**]. Sequoia [**?**] is an attempt to customize code-generation rule by XML configuration, however, it follows the similar restriction because it enforces programmer to model target as a tree of abstract machines.

Our approach is similar to Sequoia and OpenMP to perform source-to-source transformation by compiler. We shared the same idea of cite1 to generate a cluster of subprocedures for a task recursively. Instead of modifying compiler and introducing new language constructs, we exploit the capability of C++ template mechanism to achieve translation. All transformation rules are programmed in C++ meta-programming [**?**]. In libvina, a primary limitation is that only type and static constant value are available in compiler time. Therefore, it only works for problems which have rich static information. Fortunately, applications with this attributions are numerous in multimedia, digital processing, and scientific computations

Because libvina effects in compiler time, it is possible to avoid from deploying runtime system, which means that it can incur minimal runtime cost if applications allow. Another advantage of template approach is that it imposes less restricts comparing with other static approaches:

- Thread-independent: It is possible to generate all kind of threads for target, including pthread, native LWP provided by OS, even vendor-dependent thread.
- Execution-independent: Our approach does not bind to

any execution model. Libvina can change program to SPMD threads to exploit parallelism from multicore; link many subroutines to build a pipeline to hide latency of memory; or separate our program into blocks to improve locality.

- Extendability: It is flexible to develop new template class to utilize new architectural features. Template meta-programming is intimate for C++ experts. It's easy to extend new execution rules and parallel patterns. Another approaches have to ratify languages and then modify compiler to add features.
- Portability: Template mechanism are standardized in ISO [?], [?]. Our template library conforms to standards and is guaranteed to be portable for all mainstreaming C++ compilers.

The remaining parts of this paper are structured as follows. Section 2 summarizes some related works on static transformation and other library-based solutions. Section 3 introduces fundamentals of libvina. Then Section 4 represents some typical transformations in libvina. Experiments and results are in Section 5 to evaluate effect of transformations. Final section is conclusion and future works.

II. Related work

As mentioned before, it is desirable to extend conventional programming languages to reflects the essence of newer hardware. Researches in the field have two major directions:

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

First, library is a common method to extend language capability without introducing new language constructs or modifying grammar. POSIX thread library is a de facto standard interface to utilize multi-thread for applications on UNIX-like systems. The relationship between pthread and native thread provided by OS is straightforward. Therefore, abstraction of pthread is far away from naturally expressing parallelism and concurrency. Furthermore, the implementation of thread on hardware is unspecific in the standard, so it can not guarantee performance or even correctness on some architectures [?], [?]

C++ community intends to develop parallel library while bearing generic programming in mind. TBB(Thread Block Builder) has a plenty of containers and execution rules. Entities including partitioner and scheduler in TBB are created in runtime. 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 massive ALUs are exposed by hardware. Solution we proposed here is orthogonal to runtime parallel libraries. We only exploit parallelism that can be determined in static time, developers feel free to resort to other ways to speedup the programs in runtime, such as TBB.

Second, language communities extend their semantics by modifying compiler. They could add directive or annotation to help compiler transform source code. OpenMP is designed for shared memory and has shipped in almost every C/Fortran compilers. OMP compilers transform sequential code into multi-threaded programs with OMP runtime. Although OpenMP is simple and portable, the performance is not optimal in most cases. A handful of directives leave small room for further enhancing program and scaling up to larger systems. Hybrid OpenMP with MPI is possible, but difficulty surges.

Another approach needing to modify compiler is to add new language constructs. Sequoia is a source-to-source compiler to support 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 terms a computation-intensive function as *task*. New constructs apply on *task*. Sequoia compiler transforms a *task* into a cluster of subprocedures based on machines hierarchy. The target machines are described in XML files. Based on this idea, Sequoia actually can specialize task for target architectures. [?] reports Sequoia can successfully transform codes for CellBE, SMP, cluster while keeping very competitive performance. However, Sequoia as a language extension is restrictive. The basic data structure is array, which obviously shows the preference of scientific computation. The language constructs do not cover all the common parallel patterns such as pipeline and task queue. Libvina derived the idea of Sequoia to transform source code recursively. However, template meta-programming utilizes existing mechanism in C++ compiler and is capable to express all the semantics in Sequoia programming language. Moreover, Sequoia can not leverage type system to specialize code. *e.g.* Many modern processors usually provide SIMD instructions. Libvina could generate corresponding source based on predefined vector types. Sequoia compiler ignores this important information and simply relies on following compiler.

With the astonishing pace of core proliferation and architecture refinement, GPU has evolved to a pioneer of supercomputing. OpenCL [?] programming language is strongly affected by vender-dependent predecessor CUDA. It provides a unified computation platform. The downside of OpenCL is the obvious bias toward Streaming architectures. Because OpenCL has to assume its simplest computation unit for target, recursion in kernel function is forbidden in the specification. It is regarded as a simple and neat way to divide problems. Libvina can cooperate with OpenCL to transform code for GPU. It recursively generates tasks and finally emit OpenCL kernel programs while hiding details to user programming.

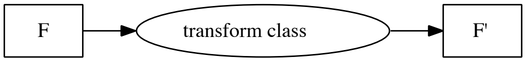

Figure 1. transform class

III. MECHANISM OF LIBVINA

A. template meta-programming approach

Originally, C++ template mechanism is invented to supersede C preprocessor. It is type-safe and could facility generic programming. People found the potential of template computation by chance. [**?**] later proved template itself is Turing completeness.

Template meta-programming is similar to functional programming language except it effects in compiler time. It only relies on static information to determine control flow and perform computation. Beside the job it meant to do, template has successfully applied to many innovative purposes in modern C++ programming practices [**?**]. MPL Library [**?**] provides control statement and STL-like data structures, which greatly eases programming in static realm.

B. Concepts

In libvina, we assume C++ compiler front-end with template mechanism as a code generator. It actually practices source-to-source transformation in the guidance of meta-programming.

Libvina focuses on computation-intensive functions which are commonly referred to *kernel* or *filter*. Mathematically, a function is single-target binary relation. Kernel functions are usually self-contained, *i.e.* external data references are limited and calling graphs are simple. It's possible for a kernel function to decouple into a group of subprocedures. Each subprocedure may be exactly the same as kernel and spread on multicore running in parallel. Another approach is to divide a kernel into finer stages and run in streaming manner to respect data locality and bandwidth. In libvina, a *transform class (TF class)* is a template class which transforms a function to a cluster of subprocedures in isomorphism. As shown in 1, the transformed function on right side has the same interface while owns a call graph to complete the original computation by a cluster of subprocedures. Execution of the call graph can be programmed by in the library to scrutinise target's architecture.

Borrowed from lisp concept, *predicate* represents a indicator of some conditions. In libvina, it is a template class with static fields initialized by constant expressions consisting of template parameters and constants. These fields are automatically evaluated when template classes are instantiated. III-B is an example to determine whether the problem size is fitting to last level cache.

```
template <class T, int SIZE_A
     , int SIZE_B, int SIZE_C>
struct p_lt_cache_ll {
 enum {CACHE_LL_SIZE = 4096*1024};
 const static bool value =
     ((SIZE_A * SIZE_B
   + SIZE_A * SIZE_C + SIZE_B * SIZE_C)
   * sizeof(T) ) <= CACHE_LL_SIZE;
};
```

List - 1 : predicate lt_cache_ll

Sentinels in libvina are non-type template parameters of *TF class*, with a *predicate* as default initializer. When a template class is instantiating, sentinels are evaluated. A *predicate* determines whether a specific requirement has been satisfied. Sentinel is responsible for changing generation strategy according to the result. C++ compiler chooses different versions of a template class to instantiates basing on the values or types of template arguments. This mechanism is known as *specialization.*

Template meta-programing can only manipulate static information. As a result, ADTs in libvina need to carry information such as dimensionality as template parameters. Only Vector and Matrix are implemented in our library, because they cover a wide spectrum of application in multimedia area. Users require more versatile data structures could resort to [?]. As 2 depicted, View is a concept to abstract dataset in libvina. It is divisible and type-safe. Shadow region is the other thread space. The only approach to communicate with other thread is through ViewMT. It guarantees synchronization by signal mechanism internally. We only provide synchronizations for data set to encourage block operations.

IV. PROGRAM SPECIALIZATION

A. Multi-threaded

Multi-threading is dominant approach to utilize duplicated computation units. There are numerous kernel functions in multimedia applications and scientific computations which can easily exploit data parallelism by dividing task into smaller and independent subtasks. This feature naturally matches libvina's transformation. We implemented *mappar* and *mapreduce* language constructs in [?] as template classes.

 IV-A is the definition of *mappar* TF class. Template parameter *Instance* is computation task containing kernel function wrapper and arguments. The last template parameter *__SENTINEL__* determines control flow when internalization occurs. **??** is a specialization to generate concrete thread to practice computation. It is noteworthy that the last two arguments are *true*, which means that this class is multi-threaded and leaf node version.

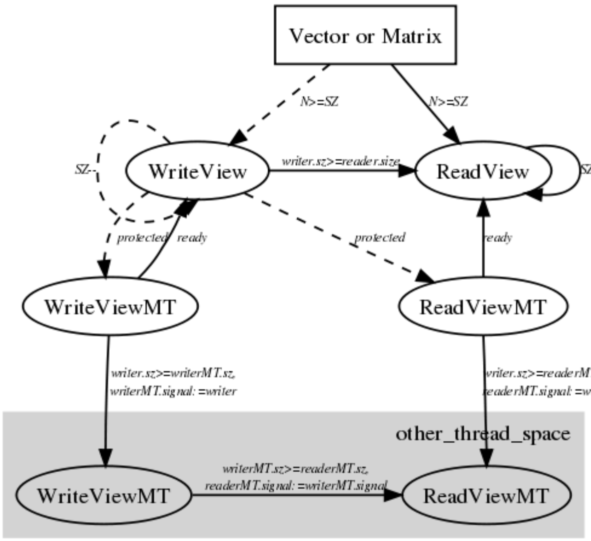

Figure 2. View

```
template <class Instance, int  _K,
  bool _IsMT,  bool __SENTINEL__>
struct mappar {
typedef mappar<typename Instance::SubTask,
        _K, _IsMT,
Instance::SubTask::_pred> _Tail;
// ...
};
```

```
template <class Instance, int  _K>
struct mappar <Instance, _K, true, true>
{
// ...
 static void
 doit(const typename Instance::Arg0& arg0,
   const typename Instance::Arg1& arg1,
   typename Instance::Result& result,
   mt::barrier_t barrier)
 {
   auto compF = Instance::computationMT();
#ifndef __USEPOOL
   mt::thread_t leaf(compF, arg0, arg1,
     __aux::ref(result, result_arithm()),
     barrier);
   leaf.detach();
#else
   pool.run(bind(compF, arg0, arg1,
     __aux::ref(result, result_arithm()),
     barrier));
#endif
  }
};
```

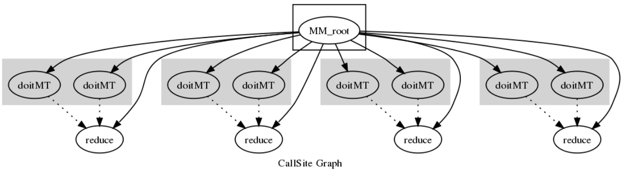

Figure 3. MM internal call graph

3 is a cluster of subprocedures generated by libvina after applying *mapreduce* to a matrix multiplication function. Template parameter _K is 2 in this case because we perform this transform for dual core x86 machine. Libvina *mapreduce* divides a matrix into 4 sub-matrices. The figure except dashed lines is actually a call graph. Shadow lines is multi-threaded environment and two subprocedures are executed in parallel. Dashed lines indicate synchronization logically.

mapseq in [?] can be trivially implemented by passing false to *_IsMT* parameter. Nested block is possible by recursively defining *TF classes*.

In libvina, programmers only care about developing efficient algorithms. Library takes responsibility for parallelization while keeping the same interface. Porting to another platforms might need to refine template parameters or apply to new template to maintain maximum performance, however, it is unnecessary for programmers to know details of algorithms beneath of the libraries.

B. Pipelining and Streaming

C. Blocking

D. Cooperation with other libraries

We implemented our library in ISO standard C++. Theoretically, any standard-compliance C++ compiler should proceed our classes without trouble. C++0X [?] added a lot of language features to ease template meta-programming. Compilers without C++0X supports need some workarounds to pass compilation. Consider the trend of C++, development libraries such as libvina should be easier in the future.

We implemented *vina::thread* based on underlying Linux PThread. A simple C++ thread pool is developed to reduce cost of thread creation. SPMD(Single program multiple data) is very important execution model. We utilize OpenCL to execute SPMD programs on GPU. OpenCL framework shipped with Mac OS 10.6 SDK is a library and be totally compatible with our approach. In order to obtain bigger performance on x86, we designed a lightweight thread library(libSPMD) based on Linux clone(2) and semaphore.

V. Experiment and evaluation

A. Methodology

C++0X is partially supported by mainstreaming compilers. Currently, we developed the library for gcc 4.4.0. In order to utilize OpenCL, we modified some codes to satisfied gcc 4.2 for apple. Modifications only increase tediousness of typing and enforce some requirements. They don't affect power of libvina or hurt performance in measurable experiments.

A couple of algorithms are evaluated for libvina. They are representatives in image processing and scientific computations.

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.

dot_prod Two vectors perform dot production.

conv2d 2-Dimensional convolution operation on image.

Three multicore systems are used to conduct experiments. On x86 platform, We linked Intel Math kernel library to perform BLAS functions. We implemented convolution 2D and other algorithms on GPU.

Table I
TABLE 1: EXPERIMENTAL MACHINES

name	ISA	topology	frequency	storage-on-chip	bandwidth
harpertown	x86 core 2	2-way quad cores	2.00Ghz	64K DCache 64K ICache 2x 6M shared L2 Cache	
nehalem	x86 nehalem	quad cores	2.93Ghz	same L1 cache; 256K L2 8M shared L3	
9400m	gpu g80	16 cores	1.10Ghz	16k local storage	3.4Gb/s

B. Results

Figures are preparing... 1. speedup
2. table: peak performance
3. comparison 3 SPMD threads.
4. cache misses reduction in blocking
5. comparison between gpu and cpu. matrix-multiplications?

VI. Conclusions

References

[1] Fatahalian, K., Knight, T., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming The Memory Hierarchy. SC2006

[2] Knight, T. J., Park, J. Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W. J., and Hanrahan, P. Compilation for Explicitly Managed Memory Hierarchies. PPoPP 2007

[3] Aho, A., Sethi, R., Ullman, J.D., Lam, M.: Compilers: Principles, Techniques, and Tools. Addison-Wesley.

[4] Boehm, H.J.: Threads Cannot Be Implemented As a Library. PLDI '05

[5] Drepper, U., Molnar, I.: The Native POSIX Thread Library for Linux. 2005

[6] Veldhuizen, T. L.: C++ Template are Turing Complete.

[7] Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., Mendelson, A.: Programming Model for Heterogeneous x86 Platform. PLDI '09.

[8] C++ Standard Committee: ISO/IEC 14882:2003(E) Programming Languages C++, 2003

[9] Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied, 2001

[10] Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, 2004

[11] Kapasi, U.J., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: The Imagine Stream Processor. Proceeding of the 2002 International Conference on Computer Design.

[12] Munshi, A., Khronos OpenCL Working Group, The OpenCL Specification ver.1.0.43, 2009

[13] Goldberg, B.: Functional Programming Language, ACM Computing Surveys, 1996

[14] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P. 2008. Larrabee: A ManyCore x86 Architecture for Visual Computing. ACM Trans. Graph. 27, 3, Article 18 (August 2008), 15 pages. DOI = 10.1145/1360612.1360617 http://doi.acm.org/10.1145/1360612.1360617.

[15] El-Ghazawi, T., Cantonnet, F., Yao, Y, Rajamony, R.: Developing an Optimized UPC Compiler for Future Architecture

[16] Gurtovoy, A., Abrahams, D.: The BOOST C++ Metaprogramming Library

[17] C++ Standard Committee: ISO/IEC DTR 19768 Doc No. 2857, 2009