The High Performance Embedded Computing Software Initiative: C++ and Parallelism Extensions to the Vector, Signal, and Image Processing Library Standard

Dan Campbell
Georgia Tech Research Institute, Sensors and Electromagnetics Applications Laboratory, Smyrna, GA
dan.campbell@gtrigatech.edu

Abstract

The High Performance Embedded Computing Software Initiative (HPEC-SI) program is developing a unified computation and communication Application Programming Interface (API) and framework for high performance signal processing tasks on parallel computers. The goal of the program is to address the high cost of software in Department of Defense (DoD) systems by improving the portability and productivity of signal processing application development threefold, while improving performance by one half compared to current practices. This paper describes the motivation for the HPEC-SI program, its goals and approaches, and progress of the HPEC-SI Working Groups in extending the Vector, Signal, and Image Processing Library (VSIPL) standard to C++ and transparent operation in parallel computing systems. The C++ extension to VSIPL is described, and highlights of its advantages are considered. This paper also examines results from the Demonstration Working Group, and describes requirements and plans developed by the Applied Research Working Group for data parallel extensions to VSIPL and describes Development Working Group progress so far in developing parallel VSIPL.

1. Introduction

The High Performance Embedded Computing Software Initiative (HPEC-SI) program is developing a unified computation and communication API and software development framework for high performance signal processing applications. The focus of the program is to develop an API that will reduce the number of lines of code needed to write applications by two-thirds, reduce the number of lines of code required to port an application from one platform to another, or to another configuration of the same platform by two-thirds, and to improve the performance of applications by half. The HPEC-SI program is leveraging the success of other highly successful and widely adopted software standards. The program is developing an API that maintains as much as possible of the functionality and general methodology of existing standards, which mitigates the cost of adoption for developers, and helps ensure that the API developed by the program can be implemented efficiently.

The HPEC-SI Program is comprised of three Working Groups, each with a specific distinct goal. The Applied Research Working Group analyzes a portion of the HPEC-SI Program goals, considers existing research and solutions for the goal, and designs a general approach for a standardizable solution to accomplish the goal. The Development Working Group implements the solutions formed by the Applied Research Working Group. This includes converting a broadly described software approach into a specific, detailed, and complete specification, creating a prototype reference implementation of the specification, developing small example applications, and developing whatever testing and compliance suites are necessary. The Demonstration Working Group re-implements large scale, deployed defense applications using the defined standards to show that the APIs and frameworks developed meet the HPEC-SI program goals for improved performance, portability, and development productivity. The HPEC-SI program is divided into phases, during which each Working Group focuses on a specific aspect of the overall program goals, and releases its results to the next Working Group, or to appropriate standards bodies on completion. The program is currently in its second phase, with plans for further phases under development.

2. Phase One Results

The HPEC-SI Program has selected VSIPL as the basis for computational functionality. VSIPL is an industry standard Application Programming Interface (API) for portable vector and matrix linear algebra and

Proceedings of the Users Group Conference (DOD-UGC'05)
0-7695-2496-6/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on January 11, 2010 at 14:53 from IEEE Xplore. Restrictions apply.

signal processing which was formed under initial direction from DARPA, but which grew to include significant contributions from a wide variety of academic, commercial, and government groups. Since publication of the VSIPL specification, VSIPL has enjoyed tremendous success, and wide adoption among new development efforts in the signal-processing domain. VSIPL has had significant impact on signal processing application portability by providing a C interface to the operations most often accelerated through proprietary libraries on embedded platforms. This has allowed high performance portability between a wide variety of platforms with minimal rewrite. VSIPL has improved productivity by raising the abstraction level of application software from a need to hand optimize performance critical portions to an ability to leave software written in a language that more closely resembles the mathematics expressed by the algorithm.

During Phase One of the HPEC-SI program, the Demonstration Working Group ported an existing, deployed defense application to use VSIPL, the Message Passing Interface (MPI), and the Data Reorganization Interface (DRI); the Development Working Group developed C++ extensions to single-processor VSIPL; and the Applied Research Working Group determined requirements for parallel C++ VSIPL.

The Phase One Demonstration Working Group effort was the use of existing middleware standards (VSIPL, Data Reorganization Interface (DRI), and the Message Passing Interface (MPI) to port an existing Common Imagery Processor (CIP) application. The HPEC-SI program goals were measured and lessons learned from the porting effort were considered and presented to the Development Working Group. The porting effort achieved approximately six-fold productivity increase, ten-fold portability increase, and three-fold performance increase over the original implementation. The Demonstration Working Group produced several recommendations for improvements to middleware standards that should be brought forward, including improved cohesion of the middleware standards, increased functional coverage in basic profiles, and improved performance predictability.

VSIPL has been extremely successful, and provides an excellent basis for HPEC-SI development, but it has shortcomings that make it insufficient to meet the HPEC-SI program goals in its original form. The HPEC-SI Development Working Group during the first phase of the program focused primarily on meeting these needs. One of the first priorities of the HPEC-SI was the development of object-oriented extensions to the Vector, Signal and Image Processing Library (VSIPL) standard. This version of the VSIPL standard addressed the shortcomings discussed below. The HPEC-SI Development Working Group includes strong representation from long-time members of the VSIPL Forum, as well as leaders in signal processing and high performance computing research from industry, government labs, and academia. After consideration of past efforts to port VSIPL-like functionality to C++, the HPEC-SI Development Working Group was able to establish the specific requirements for a successful C++ VSIPL meeting the goals of the HPEC-SI program. Previous standardization efforts have demonstrated the value of a high quality, widely available reference implementation of the specification.

An effective reference implementation allows potential users an easy means of experimenting with the API, as well as facilitating development on a large number of platforms. It also provides a less costly entry path for commercial vendors. A potential commercial vendor need only build the reference implementation for his platform in order to provide basic functionality, and can improve performance as desired by customizing whichever subsets of the standard are of most interest to customers. A reference implementation of the VSIPL C++ specification was developed and maintained by program participants from CodeSourcery, LLC. The Development Working Group has shown that a VSIPL-like API can be extended to C++ with the corresponding expected improvements in development efficiency, maintainability, and performance. A proposed VSIPL C++ specification was presented to the VSIPL Forum by the HPEC-SI program in November 2003. The VSIPL Forum reviewed the C++ extensions to VSIPL and formally adopted the specification August 2004.

The performance goal of VSIPL was to deliver the portability and productivity gains without significantly diminishing performance compared to proprietary interfaces. While this goal was met, it is not sufficient to meet the HPEC-SI performance goals. Existing products, such as the Portable Expression Template Engine (PETE) have demonstrated dramatic performance improvements in compound vector operations over standard C++ operator overloading. By making use of template engines and operator overloading, it is possible to evaluate compound vector operations at an expression level at compile time, outperforming C API based approaches that handle each atomic vector operation individually. As an example, consider the vector operation:

$$A = B*C+D$$

Using a C-based atomic vector operation API (e.g., VSIPL, Core-Lite profile), this would be expressed in a manner resembling below:

```
vsip_vmul_f (b, c, temp);
vsip_vadd_f (temp, d, a);
```

Since the interface to the math library functions is at the operation level, rather than the expression level, opportunities for optimization are lost. Expression templating allows the above expression to be fused into

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on January 11, 2010 at 14:53 from IEEE Xplore. Restrictions apply.

one operation per vector element on architectures supporting a fused multiply-add operation, allows improved cache coherency by reducing the operation to a single pass through the vector, and eliminates the need for temporary storage. While this is a simple example, the same types of optimizations are available to many expressions. Figure 1 demonstrates the experimental performance gains that were achieved using these techniques for several simple expressions.

In addition, VSIPL focused primarily on single processor computation. The specification did not address multiprocessor communication functionality at all, leaving these aspects to other libraries and standards, such as the MPI. Most implementations of VSIPL and MPI are in separately compiled libraries, so the same types of optimizations discussed above for VSIPL code are lost for VSIPL/MPI combination code. Also, because the two APIs are separate, message-passing functionality is normally explicitly written into applications, usually in a very configuration specific manner. Implicit and explicit configuration (number of processors, communication topology, etc) information is so deeply embedded in high performance embedded signal processing applications, that as much as 80% software rewrite is common for a configuration change as simple as changing the number of processors in a system. Since defense acquisition cycles are typically ten to fifteen years, and computing platform generations occur every one to two years, this imposes a heavy load on defense systems. System developers using current technology and methods must either face heavy software rewrite loads, sacrificing productivity, or forego the efficiency advantages of several generations of computing hardware, sacrificing performance.

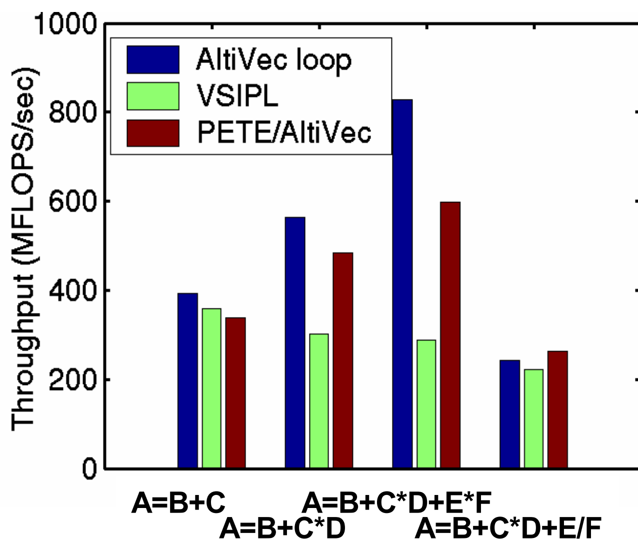

Figure 1. Experimental performance impact of Expression Templating for several simple expressions

Because VSIPL is specified in C, which does not support templates and function and operator overloading, the API requires separate function signatures for each of the accepted data types for each of the specified functions. This has resulted in a specification with many hundreds of function names and data types were as few as one tenth as many would otherwise suffice. Reducing these groups of functions to one, intuitively named function significantly simplifies the software development process. In addition, operator overloading allows much more intuitive code to be written. For example, each of the function signatures in Table 1 expresses the basic operation of

```
A = B + C
```

Table 1. VSIPL element-wise addition functions

Function	Arguments
vsip_vadd_f	(A,B,C)
vsip_vadd_I	(A,B,C)
vsip_cvadd_f	(A,B,C)
vsip_rcvadd_f	(A,B,C)
vsip_madd_f	(A,B,C)
vsip_madd_I	(A,B,C)
vsip_rcmadd_f	(A,B,C)
Vsip_cmadd_f	(A,B,C)

Note: _f represents one of at least four different specific floating point precision specifiers, and _i represents one of approximately 40 specific integer precision specifiers.

This adds complexity to the API. The inclusion of precision-templated object types for data elements such as scalars, vectors, matrices, and tensors further reduces complexity and improves code readability and maintainability.

C++ VSIPL is similar in many ways to the C VSIPL, but has several important differences. The Development Working Group recognized that one of the most significant outcomes of the VSIPL Forum effort was the establishment of a base of functionality that effectively supports high performance signal processing applications, and that can be efficiently implemented on a broad array of computing platforms. The C++ VSIPL specification therefore mimics C-VSIPL functionality and underlying abstractions as closely as possible, adding functionality or making changes when required to provide opportunities for improved productivity, portability, or performance.

C++ VSIPL maintains and augments the notions of blocks and views established in VSIPL. Blocks are the fundamental memory abstraction of VSIPL. VSIPL blocks are logically contiguous, randomly accessible one-dimensional abstract data storage. VSIPL views represent the mathematical objects on which functions operate, such as scalars, vectors, matrices, and tensors. Each VSIPL view is bound to a specific block, which provides the underlying data storage for the view. View elements map

0-7695-2496-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on January 11, 2010 at 14:53 from IEEE Xplore. Restrictions apply.

to specific block elements by means of a starting offset, and a per-dimension-element stride value. More than one view can be bound to the same block, with independent mapping parameters, providing a wide variety of methods to express a given set of data. C++ VSIPL introduces additional block properties. The block is no longer a set of specifically, strongly typed abstract data objects, but an interface, which can be implemented, and added to. User programs can implement the C++ VSIPL block interface with any class that supports the functionality required by blocks. C++ VSIPL blocks also explicitly support N-dimensional access, which simplifies data reorganization required by applications, and improves performance of multidimensional data access. In C-VSIPL, data that represents multi-dimensional information must be organized in a manner that is consistent with the VSIPL interface, which may be inconsistent with the organization both of the data source, and the preferred internal data structure organization of the VSIPL implementation. The inclusion of higher dimensional blocks allows data to be organized most efficiently for the interface between the source and the VSIPL implementation, rather than by an arbitrary organization schema. The VSIPL++ specification provides a definition for three required classes: a dense block class, a vector view class, and a matrix view class, corresponding to the VSIPL block, vector view, and matrix view objects. The dense block is templated by the data type it contains, as well as its dimensionality. Matrix and vector view classes, (and their constructors) are templated by data type, as well as block type. Data type template values default to a platform-specific default data type. Block type template values default to the appropriately dimensioned dense block. Copy constructors, and sub-view copy constructors are specified for the vector and matrix view classes, as well as constructors that arbitrarily map views to an existing block.

All C-VSIPL basic scalar, vector, and matrix mathematical and logical operators are replicated in C++ VSIPL. By using templated specifications, C++ VSIPL is able to reduce the number of and simplify the names of distinct functions required. All forms of a given VSIPL math operation share a name in C++ VSIPL. Data type, and dimensionality are templates of the function. For example, the various vector and matrix element-wise addition operations shown in Table 1, are represented by the general function signature:

```
view<data_type,block_type>add
 (view<data_type,block_type>,
 view<data_type, block_type>)
```

which is overloaded for each of the dimension of view types available (currently scalars, vectors, and matrices). Each operation that can operate on scalars is extended to operate element-wise on views. Generic programming allows this extension without the explosive complexity of function names and implementation complexity that would have been required to do the same extensions in C-VSIPL. Function names are stripped of the `vsip_` prefix, dimensionality prefixes, and precision suffixes, leaving only the base function name. Several C++ operators are overloaded to provide synonyms for element-wise functions in C++ VSIPL. A list of overloaded operators is shown in Table 2.

Table 2. C++ VSIPL Overloaded Operators

<table>
<tr><th>C++ Operator</th><th>C++ VSIPL Function</th></tr>
<tr><td>+</td><td>Add</td></tr>
<tr><td>/</td><td>Div</td></tr>
<tr><td>*</td><td>Mul</td></tr>
<tr><td>Unary -</td><td>Neg</td></tr>
<tr><td>Binary -</td><td>Sub</td></tr>
<tr><td>==</td><td>Eq</td></tr>
<tr><td>>=</td><td>Ge</td></tr>
<tr><td>></td><td>Gt</td></tr>
<tr><td><</td><td>Lt</td></tr>
<tr><td><=</td><td>Le</td></tr>
<tr><td>!=</td><td>Ne</td></tr>
<tr><td>&&</td><td>boolean and</td></tr>
<tr><td>&</td><td>And</td></tr>
<tr><td>!</td><td>boolean not</td></tr>
<tr><td>~</td><td>Not</td></tr>
<tr><td>||</td><td>boolean or</td></tr>
<tr><td>|</td><td>Or</td></tr>
<tr><td>^</td><td>Xor</td></tr>
</table>

Special vector and matrix operations in VSIPL are replicated similarly in VSIPL++, with the same data-type and block type templating, but without the uniformity of dimensionality extensions.

Signal processing operations (FFT, Convolutions, Correlations, Window functions FIR Filters, IIR Filters, and Histogram functions) are supported similarly to C-VSIPL. An operation object is created by the user that establishes the heavyweight construction and early binding operations of the function. In C-VSIPL, early binding operation options are implemented as separate object creation functions. In C++ VSIPL, options that can be known at compile time are implemented as template parameters, while runtime arguments are passed via overloaded function names. For example, for the FFT object, the sixteen VSIPL FFT object creation functions are replaced by a single templated FFT object constructor. The other signal processing objects present in the VSIPL standard are templated and overloaded similarly, creating a great simplification of the API.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on January 11, 2010 at 14:53 from IEEE Xplore. Restrictions apply.

During phase one, the applied research working group focused on developing requirements for including parallelism in C++ VSIPL. Both task- and data-parallelism were considered. Single Instruction Multiple Data (SIMD) and Single Program Multiple Data (SPMD) parallelism using MPI are generally accomplished by splitting large vector operands into smaller pieces and localizing calculations. The process of splitting vectors and matrices into pieces, distributing the pieces, redistributing, and combining the pieces during and after communication are mostly static and well established processes. Nevertheless, these steps typically require a large number of instructions to fully specify, and typically embed a large amount of information about the platform configuration directly in the source code of the software.

Use of middleware standards and libraries such as MPI and the Data Reorganization (DRI) have mitigated this problem, but the communication and organization of parallel data remains a large source of lines of code that are not directly related to algorithm specification, and are heavily platform configuration dependent. These problems significantly hinder the productivity and portability of parallel software. The working draft proposals for parallel C++ VSIPL address these problems by adding a data distribution map argument to the constructor of blocks and views. Initial data distribution will then be instantiated by the data objects, and mathematical operators will be responsible for data collection and communication to and from data objects.

The preferred method of specifying the mapping argument will be via reference to an external distribution declaration that can be read at run time. This approach addresses some of the problems with current methods of data parallel programming. Data distribution configuration information will be collected into one location per data object, and decoupled from algorithmic specification; and communication functionality will be abstracted away from the algorithmic specification of the application, and encapsulated by the math operators.

These improvements improve the productivity of signal processing application development by reducing the number of lines of code required to achieve common tasks, and improve the portability of applications by significantly reducing the amount of rewrite required in order to deploy an application to a new platform configuration. Requirement specifications were determined by the Applied Research Working Group during phase one, and delivered to the Development Working Group.

3. Phase Two Progress

During phase one, the Demonstration Working Group ported an existing DoD application to make use of existing middleware standards, and used the results of that effort to contribute to the Development Working Group; the Development Working Group extended VSIPL to C++ in order to simplify the interface and make use of several powerful productivity and performance improvements available in C++; and the Applied Research Working Group determined requirements for task and data parallel C++ VSIPL. During phase two of the HPEC-SI program, the results of each Working Group were delivered to the next Working Group in the process. The Demonstration Working Group ported existing applications to use C++ VSIPL, the Development Working Group has implemented data parallel C++ VSIPL, and the Applied Research Working Group is developing the requirements for supporting fault-tolerant parallel C++ VSIPL, as well as C++ VSIPL on heterogeneous and hybrid architectures.

The Demonstration Working Group ported several existing DoD applications to use C++ VSIPL, including a portion of the Deployable Autonomous Distributed System (DADS), by the SPAWAR Systems Center San Diego. The porting efforts implemented the existing applications in C++ VSIPL, and used the reference implementation of the standard to verify and validate the rewritten applications. The goals of the porting efforts were to evaluate the reference implementation, and to evaluate the ported program relative to the HPEC-SI program metrics. Specifically, number of software lines of code changed, and performance changes were measured and reported. Specifically, the porting effort achieved a 32% reduction in the number of lines of code, and a 49% reduction in total binary size. The performance of the ported application varied by platform with a best-case of a 30% performance improvement and a worst-case of a 10% performance loss. Also, as with the phase one demonstration, the phase two demonstrations helped to find and illustrate shortcomings in the C++ VSIPL specification that should be addressed in further revisions of the specification, as well as revealing issues that must be addressed in future phases of the HPEC-SI program.

The Development Working Group implemented parallel C++ VSIPL as determined by the Applied Research Working Group during Phase One. This phase of parallel C++ VSIPL development focused primarily on meeting the needs of data parallel application development. A first draft parallel C++ VSIPL specification was presented to the Working Group in March 2003. In this specification, block and view objects are created with a map object as a constructor argument. The map argument is itself an object that defines the data distribution of the object. Math, copy, and assignment operators are all overloaded to encapsulate implicit communication as needed, in addition to the expected behavior. Map types are defined that describe block,

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on January 11, 2010 at 14:53 from IEEE Xplore. Restrictions apply.

cyclic, and block-cyclic distributions of data objects over a set of specified nodes, with specifiable degrees of distribution. In addition, several utility functions related to parallel processing are defined, for example functions that return the number of processors available, or the rank of a particular processor. The Development Working Group considered recommendations of, and solicited feedback from developers of parallel DoD applications for desirable augmentations to the proposed specification, and implemented a draft reference implementation of the proposed parallel VSIPL++ specification. Since the initial draft was presented, the development working group has considered modifications and additions to the draft that are required, focusing most heavily on support and control functions, which require greater consideration in a parallel environment than in a single processor environment.

The Applied Research Working Group has considered fault-tolerant parallel C++ VSIPL during phase two. During the early stages of this phase, the Working Group has focused primarily on considering the methods and systems for fault tolerance that are currently in use in DoD and other signal processing applications. Common elements of the methods used, and the middleware required to support those methods is currently under consideration, and draft requirements for fault tolerant middleware are under development. As in the previous phase effort, the Applied Research Working Group will determine what elements of middleware appear to be ready for standardization and encapsulate them as a requirements set for the Development Working Group for phase three.

4. Summary

The HPEC-SI program has successfully developed a C++ extension to VSIPL that is currently in use to rewrite portions of deployed DoD applications, has demonstrated the use of middleware standards to deliver measurable improvement in productivity, performance, and portability, and is currently developing further extensions to C++ VSIPL to encapsulate data parallelism and fault tolerance. A port of a CIP application to use existing middleware standards VSIPL, MPI, and DRI achieved approximately six-fold productivity increase, ten-fold portability increase, and three-fold performance increase over the original implementation. The C++ specification was approved by the VSIPL Forum in August 2004. A standard for a unified communication and computation extension to C++ VSIPL is under development by the program, and a specification is expected to be presented to the VSIPL Forum by late 2005. Requirements for fault tolerance extensions to parallel C++ VSIPL are currently under study by the Applied Research Working Group of the program.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on January 11, 2010 at 14:53 from IEEE Xplore. Restrictions apply.