Algorithmic Skeletons within an *Embedded Domain Specific Language* for the CELL Processor

Tarik Saidani,Claude Tadonki, Lionel Lacassagne
Institut d'Electronique Fondamentale
University Paris-Sud XI, Orsay, France
saidani,tadonki,lacassagne@u-psud.fr

Joel Falcou, Daniel Etiemble
Laboratoire de Recherche en Informatique
University Paris-Sud XI, Orsay, France
joel.falcou,de@lri.fr

Abstract

Efficiently using the hardware capabilities of the Cell processor, a heterogeneous chip multiprocessor that uses several levels of parallelism to deliver high performance, and being able to reuse legacy code are real challenges for application developers. We propose to use Generative Programming and more precisely template meta-programing to design an Embedded Domain Specific Language *using algorithmic skeletons to generate applications based on a high-level mapping description. The method is easy to use by developers and delivers performance close to the performance of optimized hand-written code, as shown on various benchmarks ranging from simple BLAS kernels to image processing applications.*

KEYWORDS: *Cell processor, Generative Programming, C++ meta-programing , algorithmic skeletons,* Embedded Domain Specific Language

1. Introduction

Multi-core processors, either homogeneous or heterogeneous, are the response to the increasing demand of processing power and the energy efficiency issues. The Cell processor[19] is a good example of such an heterogeneous architecture. Thanks to both its thread level and SIMD parallelism, high peak performances are reached in various application domains[3, 23]. But using efficiently these levels of parallelism is not trivial. Even with a large collection of tools[18, 17], the development times are significantly longer than on conventional multi-core architectures. The main problem is not to fully exploit the SIMD SPE performance but to choose an optimal mapping for the considered application. In [20], several parallelization models have been investigated for the Cell: the results show that the mapping strategy greatly impact performance. Alas, some specific programming aspects, such as explicit DMA[1] transfers, make it difficult to experiment with different mapping strategies during the application design process.

Designing a tool to facilitate the programmer tasks on such architectures is not straightforward. It should provide a way to integrate arbitrary user-defined functions and an expressive way to combine them to design the application mapping. The mapping description issues can be solved using high-level models like stream processing [10], parallel design patterns [16] or algorithmic skeletons [4]. Main stroke against such methods is that high-level libraries are often designed using run-time polymorphism or other idioms which induce a large runtime overhead. To limit this overhead, Generative Programming appears as a growing trend [22]. Generative Programming is a set of techniques and tools for designing and implementing software components which are meant to be combined to generate specialized, highly optimized systems[6]. Among the various idioms related to Generative Programming, meta-programing allows the developers to write code that can analyse, transform and generate code fragments at compile-time.

An application of those idioms are the design of *Embedded Domain Specific Language* . By definition, a *domain specific language* (DSL) is a declarative languages which allow the expression of a familly of related problem as programs that are easy to understand, reason about, and maintain. On the other hand, there may be a significant overhead in creating the infrastructure needed to support a DSL. To solve this problem, methodologies were described for building embedded domain specific languages (EDSL), in which a DSL is designed within an existing general purpose programming language[14]. In this methodology, the *embedded* language is built upon the syntax of a given *host* language and extends its semantic to fit the domain needs. Meta-programing then appears as a way to boost the performance of such EDSL by leveraging most of the mashalling code at compile-time.

[1] Direct Memory Access

In this paper, we propose a C++ library named SKELL BE that provides a simple way to design application mappings based on algorithmic skeletons and seamlessly support arbitrary user-defined functions. To do so, we rely on various idioms of meta-programing in C++ like code introspection and partial evaluation to design an *Embedded Domain Specific Language* for applciation mapping with a high level of expressiveness and performance close to hand-written code. The rest of this paper is organized as follows. Section 2 presents the API provided by SKELL BE . Section 3 introduces the principles of meta-programing and the specific idioms used in our tool. Section 4 presents the experiment results for the different benchmarks.

2. The SKELL BE library

2.1. Motivation

The concept of parallel skeletons[4] is based on the observation that, in a large number of applications, parallelism is expressed in the form of a few recurring patterns of computation and communication. In general, every application domain has its own specific skeletons. In computer vision, parallel skeletons mostly involve slicing and distributing regular data while parallel exploration of tree-like structures is common in operational research applications. The main advantage of this parametrization of parallelism is that all low-level, architecture or framework dependent code is hidden from the user, who only has to write sequential code fragments and instantiate skeletons. The skeleton approach thus provides a decent level of organization.

Another interesting feature of skeletons is their ability to be nested. If we look at a skeleton as a function taking functions as arguments and producing parallel code, then any instantiated skeleton is eligible as being another skeleton's argument. Skeletons are thus seen as *higher-order functions* in the sense of functional programming. At the user's level, building parallel software using algorithmic skeletons boils down to simply combining skeletons and sequential code fragments.

When programming the Cell processor, the limited amount of communications channels and the DMA API complexity make building a proper communication schemes between PPE and SPEs or between SPEs a real challenge. In this case, the use of skeleton lessen the amount of communications pattern to write as their mostly known in advance and can be chosen so that limitations can be circumvented. Those considerations lead us to choose a proper skeleton set and programming model for the SKELL BE library.

2.2. Supported skeletons

Various Parallel Skeletons have been proposed in the literature covering both control-driven or data-driven parallelism. If there is no standard list of skeletons, a small common subset can be proposed:

- The SEQ skeleton encapsulates sequential user-defined functions to use them as parameters of skeletons;
- The PIPELINE skeleton that is functionally equivalent to the parallel composition of functions;
- The MAP skeleton models regular data parallelism in which data are sliced, sent to slave processing elements and then merged back;
- The FARM skeleton models irregular, asynchronous data parallelism based on an arbitrary load-balancing strategy.

The architectural specificities of the CELL processor induces various software design choices that we sketched in [20]. Among those, the ability to chain kernels over a single SPE and the ability to replicate a given sequence of kernels over groups of SPE have the biggest impact on the execution times of the applications. Those considerations lead to the following set of skeletons to be supported by SKELL BE :

- The SEQ and PIPELINE skeletons, defined earlier;
- The CHAIN skeleton models the sequential call of two user defined functions on the same SPE;
- PARDO skeleton models independent tasks that are all executed in parallel. This skeleton both supports *ad-hoc parallelism*[4] and provides a simple way to perform the replication of kernels over sets of SPEs.

This subset is small enough to limit the problem of complex communications schemes but large enough to support a large variety of mapping strategies.

2.3. SKELL BE programming model

In SKELL BE , we still consider the Cell processor as an asymmetric machine and propose an appropriate programming model. In this model, an application is composed of two sets of source files : one for the PPE and one for the kernel description on the SPEs. From the PPE point of view, an application may call a computation kernel that is compiled and run on the SPE as a classic stream processing application (listing 1).

Listing 1. Sample PPE kernel call

```
1  #include <skell.hpp>
2  SKELL_KERNEL(sample,(2,(float const*,float*)));
3
4  int main(int argc, char** argv)
5  {
6   float in[256], out[256];
7
8   skell::environment(argc,argv);
9   sample(in,out);
10
11  return 0;
12 }
```

A kernel is defined by the `SKELL_KERNEL` macro as a function prototype in which arguments passed by reference are considered as outputs of the kernel, while arguments passed by value or by const references are considered as inputs of the kernel. Line 9 shows how this function can be called as a normal function. At runtime, the initialization of the Cell processor made by the call to `skell::environment` starts a group of SPE threads and make them wait for an upcoming kernel call, thus reducing the overhead of threads creation. Threads termination is similarly performed at the end of the `main` function scope.

From the SPEs point of view, each SPE is part of a 8 nodes cluster supporting point-to-point communications. The applications are designed using composition of skeletons instantiated with user defined functions that access the PPE as a remote memory from which data can be asynchronously read or written through either used-defined DMA calls or one of the functions provided by SKELL BE (listing 2).

Listing 2. Sample SPE kernel definition

```
1  #include <skell.hpp>
2
3  void sqr()
4  {
5   float in[32],out[32];
6
7   pull(arg0_,in);
8   for(int i=0;i<32;++i) out[i] = in[i]*in[i];
9   push(arg1_,out);
10
11  terminate();
12 }
13
14 SKELL_KERNEL(sample,(2,(float const*,float*)))
15 {
16  run( pardo<8>( seq(sqr) );
17 }
```

This sample code illustrates various things:

- the `SKELL_KERNEL` macro generates the stub `main` function and the introspection code required by SKELL BE ;
- The `run` function that is used in the kernel function to construct an application using the `pardo` and `seq` skeleton constructors;
- The `argN_` object that provides a pervasive access to the N^{th} kernel argument stored in the PPE main memory;
- The `pull` and `push` functions that give an asynchronous access to the PPE main memory address space. Those functions actually deduces from their argument type the best way to retrieve a piece of data from the main memory and write it to a given SPE variable. By default, those functions use a static slicing of the PPE variable based on the number of SPE used by the kernel and size of the local variable used for the transfer. Overloads are provided to define more complex usage;
- The `terminate` function that triggers the completion of the kernel. It is usually called once per kernel after all the data stored in the main memory have been processed.

In term of functionality, SKELL BE kernels arguments may be of any kind of default-constructible types. However, an explicit allocation is needed for pointers, which may hinder performances. A good strategy is then to use data containers which pull data from PPE using an application specific strategy like tiling or some variation of software cache. Table 1 sums up the API of SKELL BE .

2.4. Related Work

State of the art skeleton-based parallel programing libraries are built on top of languages like C, C++, JAVA, OCaml or haskell. Considering our architectural target, the most representative works are :

- **BlockLib** [1] is a C library using preprocessor macros to generate code for the Cell processor using a small subset of skeletons and support some vectorization primitives. Benchmarks show that code written using BlockLib perform as well as hand-written code at the cost of a relatively verbose macro-based interface.
- **MUESLI**, the Münster Skeleton Library [15] is a C++ skeleton library proposed by Herbert Kuchen. MUESLI generates a process topology from the construction of various skeleton classes and to use a distributed container to handle data transmission. This polymorphic C++ skeleton library is interesting as it proposes a high level of abstraction but stays close to a language that is familiar to a large crowd of developers. Moreover, the C++ binding for higher order functions and polymorphic calls ensure that the library is type safe. The main problem is that the overhead due to dynamic polymorphism is rather high (between 20 and 110 % for rather simple applications).

Application handling	
`environement(argc,argv)`	Application start-up
`rank()`	Return the current SPE PID
`terminate()`	Signal the end of the data stream and terminate the application
`run(skeleton)`	Execute a skeleton application
Skeletons constructors	
`seq(f)`	Turn an user-function in a skeleton task

`operator,(`s_1`,`s_2`)`
`chain(`s_1`,...,`s_n`)`
`chain<N>(s)`
	Sequential composition constructor

`operator|(`s_1`,`s_2`)`
`pipeline(`s_1`,...,`s_n`)`
`pipeline<N>(s)`
	Pipeline constructors

`operator&(`s_1`,`s_2`)`
`pardo(`s_1`,...,`s_n`)`
`pardo<N>(s)`
	Pardo constructors

Data transfer	
`pull(arg`$_N$`,v,sz=0,o=0)`	Retrieves sz elements of the N^{th} data from the PPE main memory and store it in v with an offset o
`push(arg`$_N$`,v,sz=0,o=0))`	Send sz elements of the data v to the N^{th} data in PPE main memory with an offset o

Figure 1. SKELL BE User Interface

- **The ESkel Library** [4] proposed by Murray Cole represents a concrete attempt to embed the skeleton based parallel programming method into the mainstream of parallel programming. It offers various skeletal parallel programming constructs which stay similar to MPI primitives and which are directly usable in C code. However, eSkel low-level API requires to take care of internal implementation details.

Compared to those tools, SKELL BE provides a few advantages. As we'll see later, SKELL BE performances are only a few percent less than hand-written code compared to the large overhead introduced by MUESLI and its runtime polymorphism implementation. In term of interface, SKELL BE is lighter than the macro-based BlockLib and more type-safe than eSkel that relies on untyped pointers.

3. SKELL BE Implementation

Developing an efficient yet expressive skeleton-based parallel programming library is a rather complex task. Various attempts show that the trade-off between expressiveness and efficiency is a critical parameter to the success of a library. While polymorphism seems to be the tool of choice to express the relationship between skeletons and function objects, experience demonstrates[15] that the overhead induced by its runtime support may hinder the application overall performance. In the case of skeletons, the run-time polymorphism is in fact mostly unneeded. By design, an application expressed as a combination of nested skeletons has its structure entirely known at compile-time. The idea is then to find a proper way to exploit this compile-time information in an useful way.

The problem can be solved by a perspective shift. Let's consider skeleton constructors as keywords of a small, declarative domain-specific language[2]. The informations about the application to generate are then given by the operational semantic of those constructors. This strategy is classically used by parallel skeleton library like P3L[2] or Skipper[21]. In our case, the challenge is to find a way to define such a language as an extension to C++ –defining an EDSL – without having to build a new compiler variation but by relying on meta-programing .

Meta-programing is a set of techniques inherited from Generative Programming that enable the manipulation, generation and introspection of code fragment within a language. By comparison, when a function is executed at runtime to produce run-time values, a meta-function operates at compile-time on code fragment to generate new, more specialized code fragment to be compiled. The execution of a meta-programing enabled code is then done in two passes. In C++ , such a system is carried out by template classes and functions. By using the flexibility C++ operator or function overloads and the fact that C++ templates can achieve arbitrary compile-time computation, we can evaluate **at compile-time** the structure of a parallel application described as a combination of parallel skeletons constructions. To do so, the skeleton structure extracted from the application definition has to be transformed into an intermediate representation based on a network of sequential processes. For SKELL BE , the main difficulties were to **embed the skeleton constructor as language element**, **generate code over SPEs** and to perform **arguments transfer between PPE and SPEs**.

3.1. Generating code over SPEs

Implementing a *Embedded Domain Specific Language* in C++ requires to find a proper way to retrieve non-trivial informations about a given expression abstract syntax tree in an usable form. This is usually done by using a technique known as *Expression Templates* [25]. *Expression Templates* use function and operator overloads to build a light-weight representation of an expression abstract syntax tree. The tree structure itself is a complex template type structured as a linear representation of the tree. Informations about the expression terminals are stored as references in the actual

[2] as opposed to a general purpose language

AST object. This temporary object can then be passed as arguments to other functions that will analyse its type and extract the needed informations for the task at hand, thus performing **partial evaluation** [11, 26, 13].

To turn a skeleton AST into a proper code, we have to transform the tree structure into a network of process. To do so, we reuse the operational semantic defined in [9] and turn them into meta-programs [24] able to generate a static list of processes. Each SKELL BE skeleton constructor generate a stateless object which type encodes the skeleton structure. As an example, here is the actual code for the `pipe` operator (fig. 3). We notice that no computation is done at this point but that the skeleton structure is itself embedded in the return type.

Listing 3. The `pipe` operator

```
template<class LS, class RS>
expr<pipe, args<LS,RS> >
operator|( LS const&, RS const& )
{
  return expr<pipe, args<LS,RS> >();
}
```

This template type is now usable with our meta-functions. These meta-functions will generate structure representing process network. The `run` function now call meta-function to parse the template AST and generate the proper `process_network` type instantiation. Using template partial specialization as a pattern matching mechanism, we apply the proper semantic rules on each skeleton encountered in the AST. Once defined, this network is turned into code by iterating over its node and generating a sequence of SPMD code fragment in which the process instructions list is executed. This is done by building a tuple of function objects which contains the code of small scale operations (calling a function, sending or receiving data through DMA transfers) that are instantiated once per SPE. As an example of this process, let's take the following skeleton expression that builds a simple three stages pipeline:

```
run( seq(A) | seq(B) | seq(C) );
```

This expression leads to the following skeleton AST:

Listing 4. Skeleton Compile-time structure

```
expr< pipe
    , args<expr< seq, args<function<&A> > >
         ,expr< pipe
              , args<expr<seq, args<function<&B> > > >
              , args<expr<seq, args<function<&C> > > >
              >
         >
    >
```

The structure of this temporary object is rather clear. Successive calls of the pipeline operator is clearly visible and the terminal function object appears explicitly. For performance purpose, we use the fact that the address of a function is a valid compile-time constant to store it directly as a template parameter. We know convert this type into a proper process network representation. The end results is the following type:

Listing 5. Process network representation

```
network< int_<0>, int_<2>
       , list< process< int_<0>
                      , desc< pid<-1>, pid<1>
                            , instrs<Call<&A>,Send>
                            >
                       >
             , process< int_<1>
                      , desc< pid<0>, pid<2>
                            , instrs<Recv,Call<&B>,Send>
                            >
                      >
             , process< int_<2>
                      , desc< pid<1>, pid<-1>
                            , instrs<Recv,Call<&C> >
                            >
                     >
           >
>
```

The `network` template type contains all the informations that describe the Communicating Serial Process Network built from our skeletons, namely : the PID of the first network node, the PID of the last network node and a list of processes. In a same fashion, the `process` template structure holds information about its own PID and a code descriptor. This descriptor contains the PID of the process predecessor, the PID of the process successor and a list of macro-instructions that are drawn from the skeleton semantic.

Last step is now to iterate over those types and instantiate the proper SPMD code. Listing 6 shows the final look of the generated code.

Listing 6. Generated code

```
if(rank() == 0)
{
  result_of<A>::type out;
  do
  {
    call<A>(out); DMA_send(out,1);
  } while( status() );
}

if(rank() == 1)
{
  parameters<B>::type in;
  result_of<B>::type  out;
  do
  {
    DMA_recv(in,0); call<B>(in,out); DMA_send(out,1);
  } while( status() );
}

if(rank() == 2)
{
  parameters<C>::type in;
  do
  {
    DMA_recv(in,1); call<C>(in);
  } while( status() );
}
```

The SPMD structure of chained `if` statements shows the iterative nature of our code generator. Each of these blocks perform the same operations. First the input and output types are retrieved from the meta-programmed analysis of the function type. Those types are then instantiated as an unique tuple. The actual process code is then run into a loop that awaits for a termination signal. In this loop, each of the macro-instructions appearing in the process network type description generated a concrete call to either a DMA transfer function or to the function call proxy function that extracts data from tuples, feeds them to the user-defined function itself and returns a tuple of results. Most parts of this process are leveraged by Boost libraries like Proto that handles the generation of the meta-programmed semantic rules and Fusion which handles the transition between compile-time and runtime behaviour[7].

This generation process also shed a light on why SKELL BE performances are better than other C++ solutions like MUESLI. In the generated code, all the functions and skeleton dependant code have been statically resolved. All data types are concrete and all functions call are direct. No runtime polymorphism actually takes places and the compiler is able to inline more code and to perform deeper optimization than on polymorphic C++ code.

3.2. PPE/SPEs Communications

The last challenge is to provide a pervasive way to transfer the kernel arguments between the PPE that is usually in charge of I/O operations with the user and the SPEs. A classic strategy consists in building a *control block* which gathers common informations to be transferred to SPEs at the beginning of the program. Usually, this control block is an application-dependant structure that contains all the data hat the SPE kernel needs. In our case, those data are provided as arguments of the main kernel function call. Then we have to build **at compile-time** a proper control block structure. This is done by using template meta-programing to parse the function prototype to extract a list of its argument types. The control block is then defined as an array containning each SPE memory space base address and a tuple built using the following algorithm :

- Native types inputs are stored by value;
- Native types outputs are stored as a pair made of their address and a static value containing their size;
- Arrays are stored as a pair made of the address of the array elements and a static value containing the array size;
- User-defined types are stored as a pair made of the address of the object's data and size. Those values are retrieved by an eventual `begin` and `size` method. If such methods are not available, we try to call an overloaded `begin` and `size` free function. If this fails, we take the address and `sizeof` of the object. This protocol ensures that user can provide ways to efficiently transfer the user-defined types by providing one of these functions or methods. Types introspection is used to check for the existence of those methods and functions.

This structure is then filled with the actual data passed as argument to the kernel call before launching the SPE threads. As an example, the following function prototype :

```
void f( int, int[5], float& );
```

is turned into this structure :

Listing 7. Transfer structure

```
struct f_args
{
  int                       arg0;
  pair<int, int_<5> >       arg1;
  pair<float, int_<4> >     arg2;
};
```

and the following function is generated to fill it :

Listing 8. Transfer structure filler

```
void f_args_fill( f_args& a, int v0, int v1[5], float&
     v2 )
{
  a.arg0 = v0;
  a.arg1.first = &v1[0];
  a.arg2.first = &v2;
};
```

On the SPE side, the `argN` objects provide an implicit template cast operator that retrieves the values of the N^{th} tuple element through DMA and an affectation operator that transfers a value to the corresponding data in the PPE address space. The automatic template arguments deduction allows a compact and intuitive syntax such a way that the compiler can call the correct DMA transfer primitives based on the argument index and value type.

4. Experimental Results

We want to show that SKELL BE does not induce performance loss. To do so, we ran several sets of tests. The first test relies on synthetic applications combining skeletons to evaluate the overhead of meta-programs. The other tests evaluate the performance of numerical algorithms and of an image processing algorithm – the Harris and Stephen corner detector[12] – using different mappings. All tests ran over an IBM QS20 blade in optimal bandwidth usage conditions[20], were compiled using the `gcc` tool-set and evaluates the number of cycles per elements of each kernel.

4.1. Synthetic benchmarks

The first benchmark aims to assess the fact that the overhead induced by the meta-programming layer is negligible for a single skeleton. To do so, we compare the execution time of a synthetic kernel using each skeleton with an increasing number of SPEs built using SKELL BE or with hand-written code. First tests assess that running a function through the SEQ and the CHAIN operator doesn't incur a large overhead. The measurements show that the overhead introduced by those skeletons is indiscernible when considering the accuracy on execution time measurements. Examining the assembler code source shows that the only difference between a direct call or chaining and the skeleton version is a pointer indirection to resolve the actual function address from the function adapter object internally used.

The tests for the PIPE skeleton build a pipeline of 2 to 8 SPEs in which data transfer is negligible. Each stage of these pipeline executes a function which duration T ranges between 10ms and 200ms. Same tests are done for the PARDO skeleton. Figure 2 and 3 summarize the results of these measurements. In most cases, the overhead is always less than 1.5 percent.

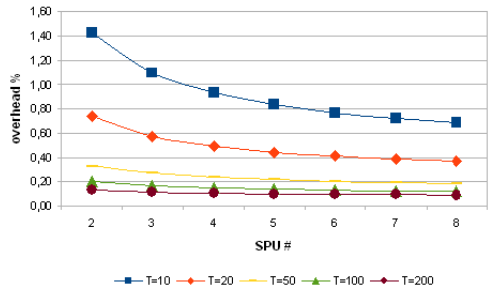

Figure 2. Overhead introduced by the `pipe` skeleton

4.2. Scalability benchmarks

The next benchmarks implement three computation kernels by hand using the Cell SDK, with the experimental XLC single source compiler with OpenMP support for the CELL and SKELL BE . It measurements relative speed-up between 2 to 8 SPEs and assess the scalability of our tool.

DOT kernel

We perform a dot product between two arrays of 10^9 single precision floating-point elements. The OpenMP version uses a reduction directive while hand-written and SKELL BE version explicitly gather sub-dot product onto the PPE and performs the final sum out of the SPEs. In this case, OpenMP minimum cycles per element is 66.08, hence providing a relative speed-up of 3.32 while the manual implementation provides a relative speed-up of 7.98, OpenMP performances being limited by the workload handling that generates a large number of communications. In this configuration, SKELL BE provides a maximum relative speed-up of 7.85, hence an overhead which is never greater than 5%.

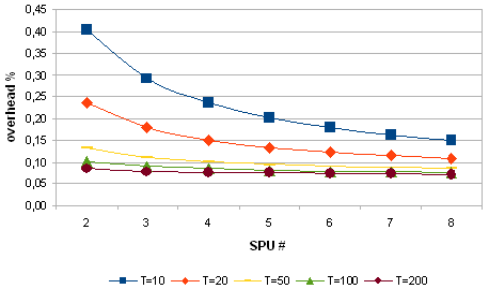

Figure 3. Overhead introduced by the `pardo` skeleton

SPE	OMP	Manual	Skell	Overhead
1	219.7	65.9	67.9	3.1%
2	263.7	32.9	34.5	4.7%
4	131.9	16.5	17.3	4.78%
8	66.1	8.3	8.7	4.9%

Figure 4. `DOT` Benchmark

CONVO kernel

We perform a convolution between a 4096×4096 pixels images and 3×3 convolution mask. In all version, the mask is duplicated on all SPEs.

SPE	OMP	Manual	Skell	Overhead
1	2402	649	672	3.6%
2	4289	391	411	5.0%
4	2146	172	181	5.2%
8	1073	98	103	5.4%

Figure 5. `CONVO` benchmark

Manual implementation yields a maximum speed-up of 6.58 compared to the OpenMP speed-up of 2.24 and the SKELL BE speedup of 6.52. The overhead introduced by SKELL BE is still around 3 to 5% but has increased. While `DOT` only transfered back one value per SPE, `CONVO` sends back large slices of data back to the PPE. In this case, the

marshaling code need to take care of proper alignment of address and size before proceeding to the transfer. Those checks that can only be done at runtime explains the additional overhead.

SGEMV kernel

Last benchmark involves a matrix-vector product between a 4096×4096 elements matrix and a 4096×1 vector. Considering the size of the matrices, all data are stored on the PPE and transfered to each SPE.

SPE	OMP	Manual	Skell	Overhead
1	200.7	179.8	187.9	4.6%
2	208.5	79.9	83.9	4.9%
4	104.3	42.2	44.5	5.5%
8	52.2	23.6	25.0	5.9%

Figure 6. SGEMV **Benchmark**

Again, SKELL BE scalability is on par with the hand-written code. Overhead almost reach 6% with 8 SPEs and exhibit the same increase than between DOT and CONVO, thus validating the link between the overhead and the communication marshaling.

Benchmark analysis

Those tests show that for those basic kernels, SKELL BE overhead never exceeds 6%. In all cases, this overhead is induced by the marshaling code around communications that handles kernel execution. The advantage is that, contrary to the current experimental version of OpenMP, scalability is preserved. The poor results of OpenMP may be linked to the fact that OpenMP use a generic software cache for handling communications[5] while our hand-written code use specific data transfers schemes.

4.3. Harris and Stephen corner detector

Next step is to show how SKELL BE behaves for complete algorithms with different mapping strategies. We chose the Harris and Stephen corner detector[12], which is a basic block with a large number of image processing kernels. This detector computes what Harris and Stephen call a coarsity matrix that gives for each pixel its 'cornerness' value.

Basically, a corner is defined as a pixel which gradient is strong in both directions. The sequential implementation of this algorithm is given on figure 7. The parallel version of the Harris operator can be built in various ways: the computation kernels can be fully or partially chained and the resulting new kernels can then be replicated and/or pipelined over the 8 SPEs. We consider three versions of the same algorithm: the full-chain version in which the four kernels are

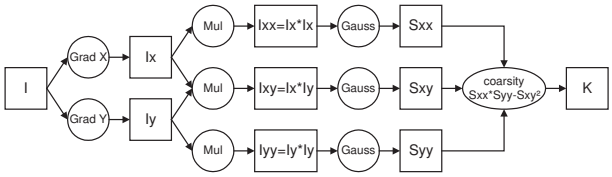

Figure 7. Harris and Stephen operator algorithm

chained and replicated on each SPEs; the half-chain version in which the Gradient and the Multiplication kernels are chained, then pipelined with the chaining of the Gauss and Coarsity kernels and replicated four times; the no-chain version in which all kernels are pipelined and replicated twice. All these kernels uses a dedicated type for handling images tiles and take care of border transfers. Listing 9 presents the three versions of the detector using the `seq` function, `pardo` functions and the infix version of `chain` (represented by the comma operator) and `pipe` (represented by the bitwise or operator).

Listing 9. Three SPE kernels for the Harris and Stephen operator

```
void full_chain_harris(tile const&,tile&)
{
 run( pardo<8>((seq(grad),seq(mul)
              ,seq(gauss),seq(coarsity)));
}

void half_chain_harris(tile const&,tile&)
{
 run( pardo<4>(  (seq(grad),seq(mul))
               | (seq(gauss),seq(coarsity)));
}

void no_chain_harris(tile const&,tile&)
{
 run( pardo<2>( seq(grad)  | seq(mul)
              | seq(gauss) | seq(coarsity));
}
```

Table 8 present the results obtained with those kernels.

Version	Full-chain	Half-chain	No-chain
Manual	11.26	8.36	9.97
SKELL BE	11.86	8.64	10.43
Overhead	5.33%	3.35%	4.61%

Figure 8. Harris and Stephen overhead

We compare the number of cycles per points spent by a hand-coded version of those kernels (using explicit DMA transfers) and the three previously defined kernels for 512x512 images. The worst case overhead introduced by SKELL BE is slightly more than 5%, which validate

our approach. The overhead is larger than previously as the adapter function calls and various marshaling code for DMA transfers handling overhead accumulate.

4.4. Impact on executable size

Template meta-programing is often blamed to produce code bloat and to generate excessively large executable file due to code replication. With the CELL processor and its 256KB local store, such code bloat should be prevented or at least kept under control. To evaluate the impact of this technique on our code, we compare the size of compiled kernels on one SPE using an hand-written approach and SKELL BE .

Kernel	DOT	CONVO	SGEMV	HARRIS
C code	1.1KB	1.3KB	2.3KB	5.3KB
SKELL BE code	12.6KB	14.5KB	22.7KB	49.4KB

Figure 9. Impact on executable size

Globally, the whole executable largely fits in the 256 KB boundary of a SPE local store but he SKELL BE kernel compiled code is ten times larger than the hand-written version. This overhead is mainly induced by the fact that SKELL BE generate a SPMD program in which all the code for all the SPEs are compiled, thus making code eight times larger. When larger code need to be compiled, source files can be compiled 8 times while passing the actual SPE PID as a preprocessor symbol and have the meta-programing layer generating code only for this PID.

4.5. Impact on compilation time

Another common problem with complex template meta-programing is its impact on compilation time. Analysis shows that the actual compilation time of any given SKELL BE program can be decomposed in two parts : an upfront 1.5s overhead due to preprocessing directives handling user-defined functions integration and an overhead proportional to the number of skeleton types used as compiler caches template type instance as it encounters them. Fine analysis using internal gcc timing shows that this overhead is mainly due to name look-up and is no less than 1s per skeleton type instance. In the worst case, like for example the half-pipe version of the Harris detector, compilation time can take as much as 10s. Those 10s have to be compared to the other tools presented in section 2.4 which are using no to a few templates and compile arbitrary skeleton code in less than 3s.

5. Concluding remarks and perspectives

Designing tools for high-level programming is a non-trivial tasks as such tools should be both expressive, efficient and easy to use by non-specialists. Such tools can be efficiently implemented as *Embedded Domain Specific Language* into a wide-spread language like C++ . In the case of the Cell processor, one of the most challenging task is to properly define an application mapping onto the processor. In this paper, we propose such an *EDSL* for building application mappings using Algorithmic Skeletons from a set of simple constructors and gave a proper implementation based on C++ meta-programing .

Experimental results assess the ability of this library to deliver high-performance and proper scalability. Benchmarks of simple kernels show that the global overhead is never greater than 6% and that the integration of third-party library function is done without any lose of performances. The benchmark of the complete image processing algorithm shows that the expressiveness provided by the tool allows to test many mapping strategies with minimal code impact.

Future works are headed toward three directions :

- In terms of features, we work on supporting multi-buffering, emulating code overlay inside SKELL BE and providing higher-level skeletons like *Divide and Conquer*,*Pyramid* or asynchronous *Farm* which extends the expressiveness of SKELL BE by supporting more irregular processing patterns.
- On a larger scope, the unification of SKELL BE and a similar tool for clusters (QUAFF [8]) interface is currently under way so we can deploy code on clusters of Cell processors or other similar heterogeneous architectures.
- Other architectural target can be considered like FPGA or GPUs. In this case, the method could be extended to not only perform compile-time generation but also use **multi-stage programming** to generate actual compilable file, compiled at run-time and retrieve an entry point to the compiled program and run it on the actual accelerator.

Other tools based on the same techniques are planned to take care of the automatic generation of algebraic code over the SPEs and the PPEs using the AltiVec extension. Such a tool, paired with SKELL BE , would lift a large part of the difficulty of Cell software development and accelerate the rate at which highly demanding applications are ported on this architecture.

Acknowledgement

This work was supported by the OCELLE project, funded by the French National Agency for Research (ANR).

References

[1] M. Alind, M. V. Eriksson, and C. W. Kessler. Blocklib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7:225–255, 1995.

[3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing*, 2006.

[4] M. Cole. Bringing skeletons out of the closet : A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 3:389–406, 2004.

[5] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engineTM architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[6] U. W. Eisenecker. Generative programming with c++. *Modular Programming Languages*, 1204:351–365, 1997.

[7] J. Falcou. High Level Parallel Programming EDSL - A BOOST Libraries Use Case. In *BOOST'CON 09*, Aspen, CO, May 2009. BOOSTPRO Consulting.

[8] J. Falcou, T. Chateau, J. Sérot, and J. Lapresté. QUAFF: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, Septembre 2006.

[9] J. Falcou and J. Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In *Proceedings of the International Conference ParCo*, Aachen, Allemagne, Septembre 2007 2007.

[10] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[11] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[12] C. Harris and M. Stephens. A combined corner and edge detector. In *4th ALVEY Vision Conference*, pages 147–151, 1988.

[13] C. A. Herrmann and T. Langhammer. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Sci. Comput. Program.*, 62(1):47–65, 2006.

[14] P. Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[15] H. Kuchen. A skeleton library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.

[16] B. Massingill, T. Mattson, and B. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999.

[17] M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Conference*, 2006.

[18] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[19] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, January 2006.

[20] T. Saidani, J. Falcou, L. Lacassagne, and S. Bouaziz. Parallelization schemes for memory optimization on the cell processor. *Transactions on High-Performance Embedded Architectures and Compilers*, 3(3), 2008.

[21] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12):1685–1708, 2002.

[22] T. Sheard and S. P. Jones. Template metaprogramming for haskell. In *Haskell Workshop 2002*, 2002.

[23] A. L. Varbanescu, A. S. van Amesfoort, T. Cornwell, A. Mattingly, B. G. Elmegreen, R. van Nieuwpoort, G. van Diepen, and H. Sips. Radioastronomy image synthesis on the cell/b.e. In *Proceedings of EuroPar'08*, pages 749–762, 2008.

[24] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[25] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[26] T. L. Veldhuizen. C++ templates as partial evaluation. In O. Danvy, editor, *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, January 1999. University of Aarhus, Dept. of Computer Science.