A Compiler Extension for Parallelizing Arrays Automatically on the Cell Heterogeneous Processor

Youssef Gdura and William Paul Cockshott

University of Glasgow

Abstract. This paper describes the approaches taken to extend an array programming language compiler using a Virtual SIMD Machine (VSM) model for parallelizing array operations on Cell Broadband Engine heterogeneous machine. This development is part of ongoing work at the University of Glasgow for developing array compilers that are beneficial for applications in many areas such as graphics, multimedia, image processing and scientific computation. Our extended compiler, which is built upon the VSM interface, eases the parallelization processes by allowing automatic parallelisation without the need for any annotations or process directives. The preliminary results demonstrate significant improvement especially on data-intensive applications.

Keywords: Parallel Programming, Parallelizing Compilers, Multicore architectures, Virtual Machines.

1 Introduction

High productivity computing systems are very essential in developing high performance solutions for intractable problems such as image processing, video streaming, advanced simulation, physical phenomena... etc. Yet, the challenge for ordinary programmers is how to exploit the performance potential of modern multiple core architectures and improve productivity [17,4,5,31]? One of the key solutions to this challenge is to develop a programming tool that simplifies parallel programming. The simplicity is often measured by the level of abstraction that the programming models or tools can provide to ease the parallelisation process and exploit available parallelism implicitly.

The advent of modern multicore architectures, such as recent Intel and AMD homogeneous processors and the IBM Cell Broadband (Cell BE) heterogeneous processor, in the mainstream industry has rapidly raised the demand for friendly parallel programming tools. Consequently, more efforts have been put, in the recent years, on designing and developing parallel tools that relieve programmers from managing parallelism explicitly [17,23,24,21,31,6,32,1]. Explicit-based models usually require substantial details, such as identifying parallelism, message passing, data movement, alignment and synchronization, to be hand coded [31,2,33]. The alternative solution to this manual approach, however, is to provide tools that liberate partially or completely if at all possible programmers

from the onerous parallelisation process. Nowadays, the most widely used tools for programming multi-core machines are semi-implicit tools. Semi-implicit parallel programming models, such OpenMP, Offload, and Intel Threading Building Blocks (TBB), have had good success is the last few years. Yet, these tools often require programmers interventions to append some code such as annotations, directives or libraries to identify parallel components. Exploring possible parallelism explicitly can make these approaches unfavorable for parallelising sequential applications, especially the existing ones as they require alterations and adjustments. Thus, the definitive solution is to have tools, such as compilers, capable to parallelise code that is written in sequential style implicitly.

Fully automatic parallelisation of sequential applications by compilers would be very appropriate and practically productive approach, yet compilers often do not succeed to deliver full parallelisation because identifying every available parallelism usually requires sophisticated compiler analysis and code mapping [19,16,26,32,11,17,34]. The complexity of the analytic process, however, could be simplified if a compiler concentrates only on large data structures, such as arrays, for extracting parallelism implicitly. Arrays operations generally are implicitly parallel operations, and hence focusing on such data structures can substantially reduce the complexity of compilation process. We have taken this approach in developing a Pascal compiler with extensions to parallelise array operations automatically on the Cell BE machine.

We describe in this paper the approaches we have taken to extend the Glasgow Vector Pascal (VP) compiler and build one compiler system using a Virtual SIMD Machine (VSM) model. The work included adding a new instruction set to the PowePC machine description in order for the compiler to work together with VSM on executing vector operations in parallel on the Cell's cores. The advantages of this approach are: Firstly it reduces the complexity of fully automatic parallelization by focusing only on array expressions. Secondly, data parallelism is already exhibited in the array expressions and therefore distributing data across multiple accelerators can be handled easily. Thirdly, using the VSM model as an abstract model eases parallel programs development by concentrating on algorithms rather than on parallelization issues such as communication, partitioning, alignment, and synchronization. To examine the Vector Pascal Cell compiler (VP-Cell), we run a number of BLAS-1 and BLAS-2 kernels on the Cell processor. The preliminary results show the VP-Cell can achieve on these kernels speedups of up to 6x by using one of the Cell accelerators compared with its master processors and up to 20x when 4 cores were used compared with its master processors.

In this paper, the first section provides a brief overview of the Cell processor, Glasgow Vector Pascal (VP) language and the VSM model. The following section reviews tools that have been recently developed for programming the Cell architecture. We then descipe our compiler system and show, in the last section, how the extended compiler collaborates with the VSM model to parallelise array expressions. After that, we look at the results of experiments and the achieved performance.

2 Background

2.1 The Cell Broadband (Cell BE) architecture

The Cell BE (or Cell) processor is a Sony, IBM and Toshiba (SIT) machine introduced in 2007. It has unique and unconventional characteristics as compered to other general purpose processors counterpart. It includes heterogeneous cores that use two different instruction sets and a heterogeneous memory architecture. It consists of a master Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs). Both the PPE and SPEs support SIMD operations on 128 bit registers[3]. The PPE processor is a general purpose PowerPC architecture with 512MB main memory and has 32 x 128-bit vector registers. The PPE handles overall control of the system such as running OS's and coordinating the SPEs. The SPEs are independent simple RISC vector processors [3], and each SPE has 128 x 128-bit registers, 256KB Local Storage (LS), two execution units and an MFC interface which handles communication and data transfers. The two main mechanisms that the Cell offers for communication and data transfers are Direct Memory Access (DMAs) and mailbox.

The tools that the manufacturers provide for programming Cell only allow the development of programs that need to be parallelised explicitly on the Cell processor. This is not an easy task due to the machine's heterogeneity of memory structures and instruction sets.

2.2 Glasgow Vector Pascal

Vector Pascal is an extension to the standard Pascal programming language. There are three different implementations of Vector Pascal. The first implementation was developed by Turner in 1987 at Iowa State University, US [35]. Formella and ela from the Saarlandes University, Germany, introduced in 1992 the second version [25]. Early 2000's, an independent version was released by Glasgow University, UK. As we are attempting to port it to the Cell processor, the term Vector Pascal (or VP) shall, henceforth, refer to the Glasgow's implementation [7].

The VP front end compiler is a machine-independent compiler that applies the formal machine description approach which was introduced by Susan Graham in 1980 [18]. VP is based on Kenneth Iverson's notations to express data parallelism and to support SIMD instruction set extensions [22]. It also introduced new data type, such as pixels, and new operators, such as genetic set operations and Input / Output of arrays. Arrays' operations in VP can be applied the same way that could have been applied to scalars leaving the compiler to decide whether the operation is an array or a scalar operation based on the declaration of the operation's operands.

Standard Pascal allows arrays assignment where the right hand side of the assignment is another array. Vector Pascal extends this to allow the right hand side to be an arithmetic expression whose variable references are unsubscribed

```
var v1,v2,v3 : array[0..1024] of real ;
sc:real;
m : array[0..1024, 0..1024] of real ;
begin
read(m);
v1:= m[3];              { assign 3rd row of m to v1 (Std. Pascal}
v2:= 2;                 { assign 2 to each element of v2 }
m:= m+v2*5;             { add 10 to every element of m}
v3:= v1+v2;             { add two vectors }
sc := v1.v2;            { form dot product of two vectors}
end
```

Fig. 1. VP Array Expressions

array names; see Figure 1.

Because the degrees of parallelism that SIMD machines can provide depends on the length of registers and data type to be processed, Vector Pascal front end compiler is designed to handle different degrees of parallelism. For example, if arrays size $arrS$ is not exactly an integer multiple of the register size$regS$ the compiler generates SIMD instructions to calculate n elements of the arrays where $n = INT\,(\,\frac{arrS}{regS}\,) * regS$ and generates PPE scalar instructions to handle the remainder $MOD\,(arrS\,,\,regS)$ elements. The extended compiler, however, take the same steps using virtual registers instead of the machine physical registers to parallelise arrays on the SPE via the VSM model.

In the example shown in Figure 1, the array bounds are known at compile time. It is possible in ISO extended Pascal [20] to declare arrays whose length will be set at run time. These dynamic arrays also support data parallel operations. Figure 2 illustrates code to read in an image as a file of lines of real numbers preceded by the line widths and the number of lines, and to print out a file of the interline interpolations of the input lines. In cases like these, the compiler plants code for both SIMD and scalar versions of the array operations and selects at run time which version to run depending on whether the vectors are long enough to fit into the SIMD registers.

The VP was chosen as a base language for a number of reasons. The most important factor is that its front end compiler already designed to support adjustable degrees of parallelism, and therefore its code generators can be easily switched to produce look-alike SIMD code that operate on large registers. This feature also goes well with the VSM model we used to extend our proposed compiler. The other factor is that VP supports arrays that implicitly express data parallelism.

2.3 Virtual SIMD Machine (VSM)

The virtual SIMD machine is a parallelising tool that imitates a SIMD instruction set through stub routines. These routines carry out array operations, such

```
type rvec(n:integer) = array [1..n] of real;
prv = ^ rvec;                        {^ indicates pointer type }
var a,b,c,d:prv;
w,h,i:integer;
begin
readln(w);readln(h);
new(a,w); new(b,w); new(c,w);      {allocate buffers }
readln(a^);                          { ^ is pointer deref }
for i:=2 to h do
begin
readln(b^);
d^:= 0.5 *(a^+b^));
writeln(d^);
c:=a; a:=b; b:=c;                 { swap buffers }
end;
end
```

Fig. 2. Use of dynamic arrays

as load, store, add...etc, on one or more SPEs using virtual (large) registers. The current VSM implementaion uses 4KB virtual registers. VSM is designed to completely hide all the underlying details of the machine architecture. It contains two co-operating interpreters: a master (PPE) interpreter and an SPE interpreter[15].

PPE Interpreter The PPE interpreter is a collection of stub routines that are basically responsible for data partitioning, communication, scheduling and dispatching micro-tasks to the SPE's. We are not attempting here to convert the implementation of these routines in detail, we will rather be looking at the information or parameters that are needed for the compiler code generator to invoke these routines. The main PPE stub routines are:

- Load & Store Routines
  Each routine takes two unsigned integer arguments. The first argument determines the virtual register's number to be used in load/store operation, and the second determines the effective address in the PPE main memory from which the requested operation must kick off.
- Computation Operations
  The other PPE routines deals with computation operations, and most of them take two unsigned integer arguments. The arguments must determine the vector register's numbers on which the requested operation will be carried on.

The SPE interpreter handles all the SPE's underlying details, and hence users or compilers do not need to know about the low level machine details. This is, in fact, one of the advantage of using the VSM model. More details on the VSM model can be found in [15].

Launching SPE threads This also an impotrant routine which manages SPE threads. This routine first checks if there is enough number of SPEs or not. If it available SPEs are enough, it then creates SPE threads (aka contexts) and loads the SPE interpreter (program) into each SPE's local memory; otherwise it exits.

Array languages compilers can use this VSM as an interface to access the SPE's hardware by decomposing high level array expressions automatically into sequence of instructions that invoke the PPE stub routines. The VSM can be also used as a set of C API library routines which can be called explicitly to perform array operations on the SPEs without the need to do any data partitioning, communication and synchronization processes.

3 Related Work

In this section, we look at three semi-implicit programming tools; CellVM, Offload and Open Multi-Processing (OpenMP). These tools have been recently introduced to exploit the performance potential of the Cell processor using two different parallelisation techniques. The CellVM model focuses on task or threads parallelisation schemes while the other two focus on data parallelisation schemes.

3.1 CellVM

CellVM is an extension of a compact Java VM called JamVM, and it was designed to offload Java threads on the Cell processor. It was introduced in 2008 to overcome the barriers of the standard Java VM's to incorporating the SPEs for executing Java instructions. It is built of two collaborate Java VM interpreters: ShellVM and CoreVM [30]. The ShellVM runs on the PPE to maintain the overall control of the machine resources, and the CoreVM runs on the SPEs to execute most Java instructions[30]. The current CellVM prototype system executes in parallel one thread per SPE[30]. However, if an application has more threads than the available SPEs the additional threads will then be executed on the PPE [29]. The CellVM design does not completely hide the underlying details of the machine.

The other tool, which is similar to CellVM, is called Hera-JVM. It was introduced in 2010 as a part of research study on the Cell processor. It is a runtime system that also uses annotations to manage migration of Java application threads between the Cell's different core types [28].

3.2 Offload

Offload C++ is an extension to C++ for parallel programming. It focuses on extracting parts of a master code to run on accelerator cores rather than focusing on parallelizing code [6,12]. In 2010, Codeplay released the second version of the Offload C++ suite for programming the Cell BE processor under Linux [6,9].

It is based on wrapping techniques to offload code on the Cell's SPEs. An offload block must be annotated with the keyword `offload` enclosed within

braces. The compiler offloads any annotated parts in C++ source code to the SPEs using threads [12] and any function that is called from within an offload block. It also loads data that is defined inside an offload block into the SPEs local storage. The basic constructs Offload C++ uses are: offload scopes, dereferencing pointers and callgraph duplication technique. Offload scopes refer to both offload blocks and offload functions [12]. Non-offload variables, such as global variables, can be accessed by offload scopes using replicating techniques. Offload also provides the `outer` construct as a pointer qualifier to distinguish between a host memory pointer and a local memory pointer. The call-graph duplication technique is used to provide multiple compilation units which are needed sometime when nonoffload functions are involved.

Reports show that performance improvement can be gained using offload on the Cell accelerator cores. However, Offload requires programmers interventions to determine the part to parallelise on the SPEs. It also requires some skills to avoid any negative consequence of using the duplication technique. This technique may produce a series problem as the Cell accelerators have very small storage space; for instance, if a standard function is called several time by offload scopes, this approach then may dramatically increase the code size[9] .

3.3 OpenMP

OpenMP is an API that supports parallel programming on a rang of architectures[23] in C/C++ and Fortran Languages, and it has been in use for many years. It provides a number of environmental variables, directives and library routines to explicitly manage multi-threaded parallelism on shared-memory architectures[27]. The fundamental C/C++ clauses, for example, are: the environment variables `OMP_NUM_THREADS` and `OMP_NESTED` to set the maximum number of threads and to nest one parallel section into another parallel one respectively, and the common clauses to manage data are `shared` and `private` which show whether data is shared among threads or private to each thread. OpenMP also offers directives to determine the parts of code to parallelise. The directives begin with the keyword "`#pragma omp`". The two main parallel directives are `for` and `parallel`. The first directive splits up loop iterations on threads while the later splits up one thread into a new group of threads.

The OpenMP compiler is a two phases compiler. In the first phase, it translates sequential source code into PPE machine code and translates (clones) the parts that are defined as parallel into SPE code. Each generated code is then compiled using the standard PPE and SPE compilers[36].

OpenMP also assists in relaxing some of the complexities that are associated parallel programming such as data partitioning and communication, yet it often requires some skills and effort to get the best performance. It also, like CellVM, Hera-JVM and Offload, requires programmers interventions to determine the parts to execute or parallelise on the SPEs.

4 VP-Cell Compiler System

The VP-Cell compiler system is a single source compiler that allowed unchanged applications code to be ported and automatically parallelized on the Cell's SPEs. The system is built of two components, an extended VP compiler and VSM.

In this section, we start with briefly introduction on the PowerPC original back end compiler and then talk about the VP compiler extension and how it was integrated with the VSM model.

4.1 Overview of the PowerPC Standard Compiler

This overview is presented to give an idea on the existing VP back end compiler which will help in our discussion of the compiler extension in the following sections. The VP front end compiler is already implemented, but it needed slight changes which will be discussed shortly.

VP code generators are Java classes that are automatically generated from formal machines descriptions written in Intermediate Language for Code Generators(ILCG). A machine description typically defines registers, semantics of instructions, addressing patterns and operations and how all of these resources are mapped down to assembler syntax. We introduce here the three basic ILCG clauses used for describing machine resources: `register`, `pattern` and `instructions`. The following examples, which give an idea on ILCG notations, are drawn from the PowerPC description:

- Register declaration
  The keyword `register` is used in ILCG to define a new register and its presentation in the machine assembly.

  ```
  register int32 R3 assembles['r3'];
  register int32 R4 assembles['r4'];
  ...
  ```

  Registers can be grouped under one name as following:

  ```
  pattern reg means[R3|R4| ... | Rn];
  ```

- Patterns definition
  In the ILCG, the keyword `pattern` can be used to define non-terminal symbols or instructions. Before using non-terminal symbols in a pattern, they must be previously defined in the machine description. Code generators use non-terminal patterns to match a number of possibilities [8]. For example, the following code defines an non-terminal symbol called `real` which can have two possibilities; `double` or `float`.

  ```
  pattern real means [double | float];
  ```

  An ILCG instruction can also defined as a pattern that takes a number of arguments and determine the semantic and the corresponding assembly code. For example,

- Arithmetic Operations are defined as following

```
instruction pattern Op(operator op,reg r1,reg r2,reg r3, int t)
means[r1 := (t)op((t)^(r2),(t)^(r3))]
assembles [op' 'r1','r2','r3];
```

- Example: Referencing Memory

```
pattern addrmode (reg r)
means[ mem(^(r))]
assembles [r];
```

These examples gives an idea on how to used ILCG notations to define registers and instructions of the proposed extended compiler in section [sub:PowerPC-Compiler-Extension].

Machine specifications are store as a ILCG file which should include all the instructions that are to be implemented for the target machine. The ILCG file is then passed though a code-generator generator to create a Java class of the code-generator for the target machine [10]. A code-generator is used to match against patterns in the source program. If a match succeeds, the code generator will produce the assembly code associated with the matched pattern.

4.2 VP-Cell System Overview

The VP-Cell compiler system depends on the VP compiler extension transformations to collaborate with the VSM model. The system works as following: The VP compiler generates a PowerPC assembly code that correspond to sequential source code excluding arrays expressions and examines any array expression in the source code. If an expression includes operations on arrays of adequate size for parallelisation, the compiler then delegates the evaluation of the array expression to the Cell accelerators (SPEs) through the VSM; otherwise the evaluation is performed on the master processor (PPE). The adequate size here means arrays that have the same length or multiple of the VSM virtual registers. The main steps of the code generator are demonstrated in Figure 3.

4.3 PowerPC Compiler Extension

We extended the original PowerPC machine description by adding new vector registers and a new virtual SIMD instruction set.

Vector Registers The vector registers are expected to be large and their length (`VECLEN`) can vary, yet it must be the same size as the VSM virtual registers. We had experimented with register lengths between 1024 and 16384 bytes, but we assume here that the registers length is 4096 byte. The current implementation defined 8 vector registers of size 1024, labeled NV_0 to NV_7. The following ILCG code shows the declaration of single-precision floating-point $NV's$ register set:

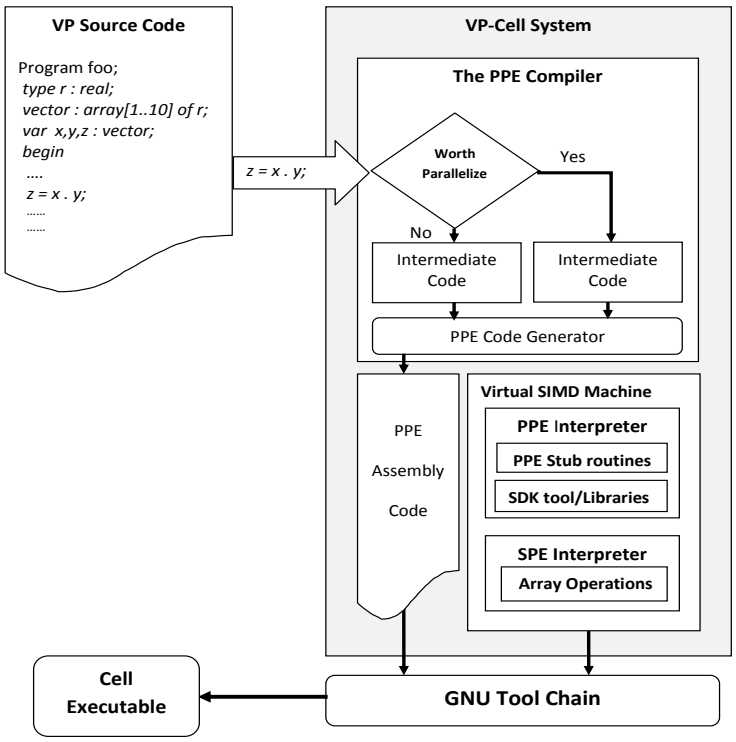


Fig. 3. Schematic Diagram of the VP-Cell Compiler System

```
/* Definition of the Virtual SIMD Machine Registers */
define(VECLEN,1024)          /* experimental parameter */
register ieee32 vector(VECLEN) NV0 assembles[' 0'];
register ieee32 vector(VECLEN) NV1 assembles[' 1'];
....
register ieee32 vector(VECLEN) NV7 assembles[' 7'];

/* Group above registers under type vreg that ranges over */
/* the vector registers */

pattern vreg means[NV0|NV1|NV2|NV3|NV4|NV5|NV6|NV7];
```

These registers are used to generate a set of RISC like register load, store, and operate instructions. One vector register can be processed by one SPE or multiple SPEs. If multiple SPEs are used, then the data in each register NV_i is equally distributed on the available SPEs. The partitioning process, if required, is the responsibility of the PPE stub routines. To illustrate how the starting addresses is computed when multiple SPEs are used, assume that the length of

the 32-bit floating point virtual registers is 4096 bytes, 4 SPEs are used, and the starting address for a load, for example, is 0XA0000. Then, the PPE routine should broadcast to the four SPEs the following starting addresses:

```
000A0000 →SPE0 ; Start address of the vector
000A0400 →SPE1 ; offset of 256 fl. points from start
000A0800 →SPE2
000A0C00 →SPE3
```

In regard to the number of registers, the VP compiler is designed to evaluate one expression at a time on the SPEs, and therefore we defined only 8 virtual registers which will be, according to Ershov formula, enough to evaluate even very long expression at a time. Ershov [13,14] suggested that the number of registers $NReg$ needed for the computation is

$$NReg \leq log_2(n+1)$$

where n represents the number of binary operators and $n+1$ is the number of operands in an expression.

Virtual SIMD Instruction (VSI) Set We also introduced new instruction set to allow parallelising operations such as Load, Store, Add, ...etc, on the SPEs. These new instructions are only required to call the right VSM routines and obviously passing the needed information. From the implementation points of view, passed information must be placed in the designated registers in accordance with the C functions calling conventions. Since all PPE routines takes two arguments and according to the PowerPC ABI, these two arguments must be passed in registers "$r3$" and "$r4$". The following samples show the semantics of some VSIs and how they were mapped into the machine code.

- Virtual SIMD Load and Store Instructions
  Each instruction pattern take two arguments: an address and an integer number, and each require two assemble instructions to pass these two parameters and one assembly instruction to call the PPE routine for a particular operation. The following ILCG code shows how the Load instruction is described in the extended PowerPC machine description:

```
instruction pattern LOADFLT( addrmode rm, vreg r)
means[ r :=(ieee32 vector(VECLEN))^(rm)]
assembles['li 3, ' r
       '\n la 4,0(' rm ')'
       '\n bl LoadVec'];
```

The first line determines the instruction pattern's name and its arguments. In this example, r is a variable of type *vreg* which has just been defined above as a non-terminal symbol for a group of vector registers, and rm is a memory reference as defined in section 4.1. In the second line, the keyword `means` defines the semantic of the instructions. Given this description, if the code generator find a match in the ILCG tree of the source code to the semantic of the `LOADFLT` instruction, it then generates the three assembly instructions which are listed in last part of the `LOADFLT` instruction. The first assembly instruction loads the register's number in the general register $r3$, the second loads the starting address in $r4$ and the last instruction is a branch and link instruction which results in a call to the PPE routine `LoadVec`.

– Virtual SIMD Computation Instructions
Most computations Instructions patterns are very similar to load and store instructions, the only difference is in type of the first arguments. A computation Instruction takes two integer values which determine the vector register numbers involved in the operation. The following ILCG code shows how the ADD instruction is defined.

```
instruction pattern ADDFLT(vreg  n,vreg m )
means[n:= +(^(n),^(m))]
assembles['li r3, ' n
     '\n li r4, ' m
     '\n bl AddVec'];
```

Both arguments here refer to vector register numbers. To comment on this particular `ADDFLT` instruction, assume that the vector registers to be added are 0 and 1, then if the code generator succeed in the matching process, it will produce the following three assembly instructions:

```
li r3, 0
li r4, 1
bl AddVec
```

The first two instructions load 0 in register $r3$ and 1 in register $r4$. This implies that the values 0 and 1 are passed to the first function to be called. The third instruction results in a call to the PPE routine (`AddVec`). What happen behind the scene is that this routine will internally send messages to each SPE requesting it to add the elements in vector register 0 to that of the corresponding elements of vector register 1 and to keep the results in the same portion in vector register 0.
Most of the instruction pattens are very similar to the instructions shown in the previous examples. However, there are a few instructions that are not similar whether in the type of argument or the number of arguments. Some binary operations differ from regular ones in the arguments types. For example, the replicate operation does not two vector registers, like the `ADDFLT` operation, instead it takes a scalar and a vector register in which the scalar will be replicated. In contrast, unary operations, such as sqrt, sin

and cos, require only one argument. The following example shows how the SQRT instruction pattern is implemented in ILCG code:

```
instruction pattern SQRTFLT(vreg r)
means [r:=SQRT(^(r))]
assembles['li 3,' r
       '\n bl SqrtVec'];
```

The first line defines `SQRTFLT` as an instruction that takes only one parameter. The second line specifies the instruction's semantic. Note here that the source register r is also used as a destination register. The semantic of this unary operation is mapped into two assembly instructions. The first loads register $r3$ with the virtual register's number r while the second invokes the PPE routine (`sqrtVec`).

4.4 Building the VP-Cell Compiler System

The last step in building VP-Cell as one compiler system is to connect the extended compiler and the VSM model together. This steps essentially requires three modifications on the VP front end compiler.

- Adjusting the degree of parallelism
  The VP front end compiler includes a method that can be overloaded to set the degree of parallelism which depends on the register length and data type. We overloaded this method by setting the length of register to be used to 4096 Byte.
- Launching SPE threads
  The compiler extension uses the VSM routine for launching SPE threads. The overhead associated with creating and launching threads usually high, and to reduce such overhead, we modified the code generator to automatically plant a call to the responsible routine in the prelude session. This allows overlapping between the PPE interpreter initialisation and the first execution of an array expression. Once the SPE contexts are launched, the VSM will keep running waiting for tasks from the PPE.
- Terminating SPE threads
  Since the SPE interpreter is running in the background, the front end compiler was modified to plant code that invokes the SPE threads termination routine at the epilog session.

It is important to note that no changes were necessary in the parser or high level optimiser of the compiler itself in order to achieve this. The rest of the process simply piggybacked on the way the compiler front end decomposes array operations for whatever processor it is targeting.

5 experimental results

We start here with a demonstration of how the extend compiler's code generator transforms VP source code into ILCG notations in both sequential and parallel forms. After that, we look at the preliminary results obtained using the VP-Cell compiler system on basic linear algebra kernels. Finally we draw conclusions about the overall performance improvement and also about the strengths and weaknesses of the implementation.

5.1 Code Generator Results

We intend here to show a sample output produced by the extended code generator for parallelising array expressions under the VSM model. To simplify the illustration, we take a simple example of adding two vectors and keeping the result in another vector. Given vectors x , y and z, Figure 4 shows VP source code for adding two 1D array `x` and `y`.In this example, `x, y` and `z` are vectors of length 4096 of 32-bit floating-point. The code adds `x and y` and keeps the result in vector `z.` The compiler after parses the source code, it generates without any parallelisation the ILCG code given in Figure 1.

```
const n=4096;
var x,y,z=array[0..n] of real;
z := x + y;
```

Fig. 4. Vector Pascal Code

Algorithm 1 ILCG Code Using Machine Registers

```
1. for(mem(+(Base, -49180)), 1 , 4096 , 1 ,
2. assign (mem(ref ie32,
3. +(*((i32)^(mem(ref i32,+(Base, -49180))), 4), +(Base, -49156))),+(
4. (ie32)^(mem(ref ie32,
5. +(*((i32)^(mem(ref i32,+(Base, -49180))), 4), +(Base, -16388)))) ,
6. (ie32)^(mem(ref ie32,
7. +(*((i32)^(mem(ref i32,+(Base, -49180))), 4), +(Base, -32772))))))))
```

The first line in Figure 1 indicates that the `for` loop will start from 1 up to 4096 and each iteration is incremented by (1). The other lines correspond to the expression `z := x + y` given in Figure 4. Note here that the symbol "`ie32`" (in ILCG notation `ieee32`) in Figure 1 refers to 32-bit floating point, and hence the machine's standard floating-point registers can be used here for

```
lis r4, Base@ha
ladd  r4, Base@l(r4)
la r22, -24580(r4)
la r17, -16388(r4)
la r23, -20484(r4)
li 3, 0 Use virtual register 0 to load x
la 4,0(r23) Load Starting address of x in r4
bl speLoadVec
li 3, 1 Use virtual register 1 to load y
la 4,0( r17) Load Starting address of y in r4
bl speLoadVec
li 3, 0 Add virtual registers 0 and 1
li 4, 1
bl speAddVec
li 3, 0 Store virtual register 0 in z
la 4,0(r22) Load Starting address of z in r4
bl speStoreVec
```

Fig. 5. Generated Assembly Code Base on New Virtual SIMD Instructions

this operation. The -16388, -20484 and -24580 values are the starting address `offsets` of vectors `x,y` and `z` from the `Base` location respectively.

The generated ILCG code that uses vector registers of size 4096B (1024 floating-point values) is shown in Figure 2. The two differences between the

Algorithm 2 ILCG Code Using Vector Registers

```
1. for(mem(ref i32,+(Base, -49180)), 1 , 4096 , 1024 ,
2. assign(mem(ref ie32 vector (1024),
3. +(*((i32)^(mem(ref i32,+(Base, -49180))), 4), +(Base, -24580))), +(
4. (ie32 vector (1024))^(mem(ref ie32 vector (1024),
5. +(*((i32)^(mem(ref i32,+(Base, -49180))), 4), +(Base, -16388)))),
6. (ie32 vector (1024))^(mem(ref ie32 vector (1024),
7. +(*((i32)^(mem(ref i32,+(Base, -49180))), 4), +(Base, -20484)))))))
```

code in this figure and the sequential form which is shown in Figure1 are: firstly, Line 1 in Figure 2 implies that the evaluation will be performed on 1024 elements in parallel in only 4 consecutive calculations instead of 4096 iterations on the PPE. Secondly, the type "`ie32`" in the sequential form becomes "`ie32 vector (1024)`" which suggests to the compiler to use, if available, vector registers of size 4096B (1024 floating points values).

Figure 5 shows the result of the VP-Cell compiler system after matching the ILCG intermediate representation given in Figure 2 with the new instructions of the extended compiler.

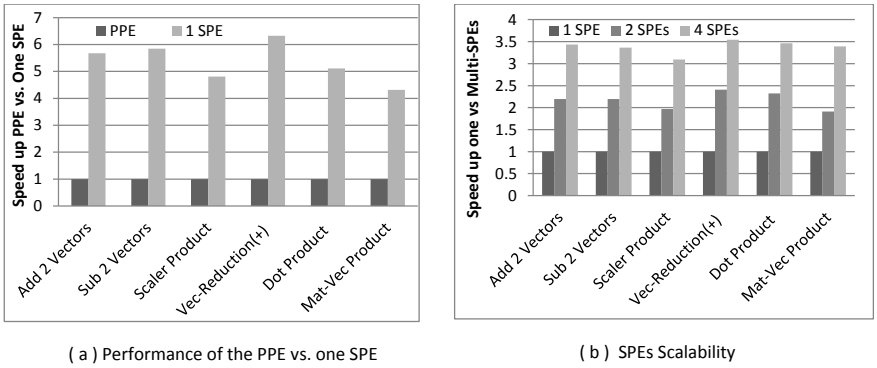
5.2 Performance Results

We present the preliminary results of the VP-Cell compiler system performance on basic linear algebra kernels. The results show the performance of a single accelerator relative to the master processor and the scalability attained using multiple SPEs. We run a number of BLAS-1 and BLAS-2 kernels on the Cell processor in the Playstation 3 (PS3) [3], and each kernel run10^5 times. The experiments include vector operations such as reduction operation, cross and dot products of two vectors and matrix-vector product which is a typical example of BLAS-2 [11]. The operations were carried out on `4096` single precision floating-point 1D arrays and `4096x4096` 2D arrays using VSM 1024 32-bit floating-point registers; that is, 4096 Bytes.

Single Core Performance Figure 6 (a) presents the performance of the selected BLAS kernels when run on one SPE compared with the PPE. The achieved speedups on the different kernels using one SPE compared to the PPE range from 4.2 times to 6.2 times. The kernels can be split into two groups based on their speedups. The first group includes the addition and subtraction of two vectors and a vector addition reduction kernels. The execution times of these kernels on the PPE were very close. The average speedup on this group was around 6 times as fast as the PPE. Yet, the chart shows that the SPE performed better on the reduction operation because of two factors. Firstly the reduction operation requires one load while the other two operations each requires two loads to get the two vectors. Secondly, the result of the reduction is a scalar, and thus it does not need to store back an entire vector instead it sends back only the scalar value. The average speedup on the other group, which includes scalar product, dot product and matrix-vector product kernels, is about 4.5 times faster than the PPE. Our interpretation of the performance degradation in the second group is due to that all these kernels involve multiplication operations which is relatively costly compared to add and subtract operations in the first group.

SPEs Scalability The previous experiments assessed the performance of the kernels on the PPE and compared with one SPE. We also evaluated the performance of the same kernels on multiple SPEs using the same source code used on the PPE. We only used 1,2, and 4 SPEs because the VSM partitions each virtual register equally on the SPEs, and therefore their size must be divisible by the number of SPEs. Also, we could not use 8 SPEs as PS3 only provides six SPE cores for user applications. Here we look at the SPE's performance and how the VSM virtual registers can effect the SPE's performance as a result of varying the number of the SPEs.

Figure 6 (b) shows the speedup obtained by the same kernels using 1,2, and 4 SPEs. This figure reports significant improvement when the kernels were parallelised by the VP-Cell compiler on 2 SPEs. In fact, diagram (b) shows a super-linear speedup when 2 SPEs were used. This is mainly due to the trade-off between computation and memory access time. With large registers size, for



(a) Performance of the PPE vs. one SPE

(b) SPEs Scalability

Fig. 6. Performance of the PPE vs SPEs

example when one SPE is used, the memory access time dominated the parallelisation process. In contrast, when 2 SPEs were used, the registers were halved and consequently the memory access time is reduced. However, the achieved speedups on 4 SPEs were not as good as on 2 SPEs because with 4 SPEs, each register was chopped into 4 parts, and therefore each core now operate on a small block of data. In this situation, the computation time of this size of data does not pay off the data movement (DMA transfer) overhead.

The overall SPE performance on the different kernels scaled linearly as the number of SPE cores increases as shown in the diagrams (a) and (b) in Figure 7. The chart (c) in the same figure provides an overview of the performance of running each kernel on 4 SPEs compared with executing the same code on the PPE. For these BLAS kernels, the 4 SPEs provides from a 15x to a 22x speedup compared to the PPE performance. It is noteworthy to mention here the existing PPE compiler has not been optimized yet to support SIMD extensions.

5.3 Conclusion

We presented here a compiler system that has the capability to automatically parallelise arrays expressions on the Cell's SPE's. We have extended the PowerPC back end compiler and connected with VSM mode to build one compiler system. The approach provides a single-source compiler for parallelising array expressions. The compiler allows unchanged applications code to be ported and automatically parallelised on the Cell heterogeneous architecture, and thus this approach simplify parallel programs development. Though the preliminary results on the BLAS kernels showed a significant improvement as the number of SPE increase, the compiler sometimes can not use all the SPEs available on the Cell due to alignment and divisibility constrains between the VSM registers length and the number of SPEs.

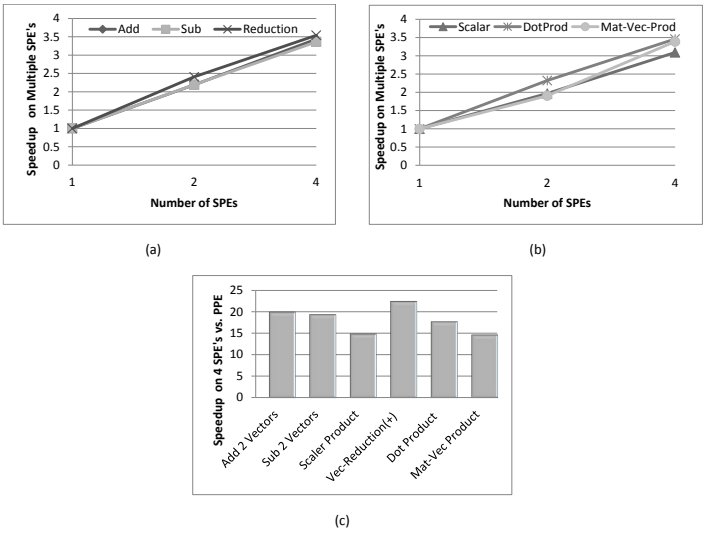


Fig. 7. SPEs Performance

References

1. A., R.: The Codeplay Sieve C++ Parallel Programming System (2006)
2. A., S.: Introduction to Parallel Programming Concepts. Tech. rep., Louisiana State University (2009)
3. Arevalo, A.e.: Programming the Cell Broadband Engine Architecture. International Technical Support Organization (2008)
4. Asanovic, K., ela: A view of the parallel computing landscape. Commun. ACM, 52(10) pp. 56–67 (2009)
5. Breitbert j., F.C.: OpenCL, An Affective Programming Model for Data Parallel Computations at the Cell BE. Parallel and Distribute Processing, Workshop and PhD Forum (IPDPSW) pp. 1–8 (2010)
6. Clarke, P.: Parallel Programming Tool Offered for Cell Processor (2010)
7. Cockshott, P., Michaelso, G.: Orthogonal Parallel Processing in Vector Pascal. J. Comput. Lang. Syst. Struct., 32, pp. 2–41 (2006)
8. Cockshott, P.: Vector pascal reference manual. SIGPLAN Not. 37(6), 59–81 (2002)
9. Cooper, P., a.e.: Offload automating code migration to heterogeneous multicore systems. HiPEAC, ser. LNCS pp. 337–352 (2010)
10. Cooper, P.: Porting the Vector Pascal Compiler to the Playstation 2. Master's thesis, University of Glasgow Dept of Computing Science, http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/PS2.pdf (2005)
11. DiPasquale N., Gehlot V., W.T.: Comparative Survey of Approaches to Automatic Parallelization. MASPLAS'05 (2005)
12. Donaldson A., Dolinsky D., R.A., G., R.: Automatic offloading of c++ for the cell be processor: a case study using offload. MuCoCoS10, IEEEComputer Society pp. 901–906 (2010)
13. Ershov, A.: On programming of arithmetic operations (1958)

14. Flajolet P., Raoult j., V.j.: The number of registers required for evaluting arithmetic expressions. Theoretical Computer Science, 9 pp. 99–125 (1979)
15. Gdura Y., C.P.: A Virtual SIMD Machine Approach for Abstracting Heterogeneous Multicore Processors (2001)
16. Geoffrey C., Williams R., M.P.: Parallel Computing Works. Morgan Kaufmann Inc., San Francisco, CA (1994)
17. Ghuloum, A.: What akes Parallel Programming Hard (2007)
18. Graham, S.L.: Table driven code generation. IEEE Computer 13(8), 25–37 (August 1980)
19. Guillon, A.: An apl compiler: The sofremi-agl compiler (1987)
20. Hetherington, T.: An introduction to the extended pascal language. ACM SIGPLAN 28, 42–51 (1993)
21. Intel: Parallel programming essentials via the intel tbb (2010)
22. Iverson, K.: A programming language. Wiley, New York (1966)
23. K., B., ela: Supporting OpenMP on Cell. International Journel of Parallel Programming pp. 289–311 (2008)
24. Krall, A., Lelait, S.: Compilation techniques for multimedia processors. International Journal of Parallel Programming 28(4), 347–361 (2000)
25. et la., F.A.: The SPARK 2.0 system a Special Purpose Vector Processor with a VectorPASCAL Compiler. The Twenty fifth Annual Hawii International Conference on System Sciences, HICSS 25 (1992)
26. et la., H.M.: Improving the effectiveness of parallelizing compilers (1995)
27. Marowka, A.: Performance of OpenMP on Multicore Processors. In: ICA3PP: Proceedings of the 8th International Conference, 2008,. pp. 208–219. Springer-Verlag, Berlin Heidelberg (2008)
28. McIlroy, R.: Using program behaviour to exploit heterogeneous multi-core processors. In: PhD Thesis. Glasgow University (2010)
29. Mcilroy R., S.J.: Hera-JVM: Abstracting processor heterogeneity behind a virtual machine. The 12th Workshop on Hot Topics in Operating Systems (HotOS 2009) (2009)
30. NOLL A., GAL A., F.M.: Cellvm: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. Tech. Rep. 06-17, Tech. Rep. (2006)
31. P., M., ela: Is Parallel Programming Hard, and if so, why? (2009)
32. Pingali, K.: Parallel programming languages (1998)
33. Rauber T., R.G.: Parallel Programming: For Multicore and Cluster Systems. New York: Springer-Verlag (2010)
34. Tournavitis G., Wang Z., F.B., M., B.: Towards a Holistic Approach to Auto-Parallelization. In: ACM, PLDI09 (October 2009)
35. Turner, T.: Vector Pascal a Computer Programming Language for the Array Processor. Ph.D. thesis, Iowa State University, USA (1987)
36. Wei H., Y.J.: Mapping openmp to cell: An effective compiler framework for heterogeneous multi-core chip. the 3rd international workshop on OpenMP (2008)