A Virtual SIMD Machine Approach for Abstracting Heterogeneous Multicore Processors

Youssef Gdura
Department of Computer Science,
Glasgow University,
Glasgow, UK
ygdura@dcs.gla.ac.uk

Paul Cockshott
Department of Computer Science,
Glasgow University,
Glasgow, UK
wpc@dcs.gla.ac.uk

Abstract — **The heterogeneous design of multi-core processors, such as the Cell processor, introduced new challenges in porting high-level languages. Our project is developing tools that hide the underlying details of the Cell processor and eases parallel programming. We present a Virtual SIMD machine (VSM) paradigm that can be used to parallelize array expression automatically. The novelty is the use of a virtual SIMD machine model to completely hide the underlying details required for programming the Cell processor. The VSM paradigm can also be used to develop an automatic parallelizing compiler for the Cell Broadband Engine (Cell BE). In this paper we give an overview of the VSM interface and present preliminary results that show the performance of our VSM and its behavior on multiple accelerator cores using basic arrays operations.**

Keywords-component; High-level Languages, Virtual Machine, Parallel techniques, Multicore Compiler.

I. INTRODUCTION

Many application areas, such as image processing and scientific computation, have enough parallelism to make good use of the multi-core technology, yet multi core architectures are still not fully used. This is due to the lack of parallel programming tools that can exploit parallelism and automatically parallelizing code for multi-core machines [2][3]. The most commonly used parallel programming tools nowadays are OpenMP and MPI [1][3][6]. These models offer semi-automatic parallelization tools that depend mainly on directives and run-time routines in selecting and parallelizing code segments.

Our project aimed at designing a Virtual SIMD Machine (VSM) model that a complier can use to parallelize large data structures, such as arrays, automatically. 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. Thirdly, it eases the task of developing programming parallel applications by concentrating on algorithms rather than on parallelization issues such as communication, partitioning, alignment, and synchronization.

We designed and implemented a VSM model that hides the Cell heterogeneity. It can be now used to automatically parallelize and execute array operations on the Cell accelerator cores. The VSM is a register-based virtual machine that is designed mainly to access the Cell's accelerator cores. Our VSM is a language-independent implementation written in C++. It is built of two co-operating interpreters, one for the Cell's master processor and one for the accelerator cores. The master's interpreter basically is stub routines that are responsible for data partitioning, communication and scheduling micro-tasks to execute in parallel on the Cell accelerator cores.

This paper briefly introduces the Cell processor. It then presents an overview the VSM paradigm and talks about the main challenges involved in designing and optimizing our VSM. The last section shows the outcome of the experiments that were conducted to assess the efficiency of our VSM and looks at the preliminary results of running basic linear algebra operations on the Cell processor.

II. THE CELL BE ARCHITECTURE

A. Overview

The Cell BE, or Cell, is a heterogeneous multi-core processor. It was design mainly for multimedia applications[5], and has used in other areas such as high performance computing. Cell BE has two quite distinct processors: a 64-bit PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs) [3]. Both PPE and SPEs support SIMD operations on 128 bit registers, but they have two different instruction sets; one for the PPE and one for the SPEs[5][8]. The PPE has 3 levels of storage (512 MB RAM, 64KB L1 and 512KB L2 cache) and 32 x 128-bit vector registers. Each SPE has only 256KB Local Store (LS) and 128 x 128-bit registers. The SPE local store is quite distinct from, and does not shadow or cache, the main memory.

B. Communication Within the Cell Processor

Each SPE has also a Memory Flow Controller (MFC) to handle communication and data transfers between the PPE and the SPEs. The MFC provide three means of communications: Mailboxes, Signal Notification Registers and Direct Memory Access (DMA) mechanisms. The first two mechanisms can be used to exchange 32-bit messages between the PPE and the SPEs. The Cell's DMAs operations, such as GET and PUT, can be used to move data between the main memory and local stores. A DMA's size can range up to 16KB and must be a

DOI: 10.5176_2010-2283_1.4.117

multiple of 16 bytes. To transfer data between the PPE and the SPEs using a DMA operation, A DMA requires main memory address, local storage address, the size of data to be transferred and a flag to group DMAs. What is important here is the Cell constrains DMA transfers. It required both addresses to be aligned on 16-byte boundaries or 128-byte boundary for better performance [7]. DMA transfers cannot be used for updating data shared by multiple SPEs. Instead, Cell provides special atomic DMA operations, such as *Getllar* and *Putllc*. The "*Getllar*" operation locks a cache-line (128B) and reserves it before transferring the 128 byte from the main memory to LS. The "*Putllc*" transfers a 128 byte from LS into the main memory only if the cache-line lock is reserved. Our VSM uses these atomic operations for synchronization purposes.

C. Programming Cell

The Cell processor potentially offers high levels of parallelism, but it is not easy to program due to its heterogeneity of memory structures and instruction sets. The two programming languages that are currently functioning on the Cell processor are C/C++ and FORTRAN. These languages support a number of parallel programming models such as OpenMP, Sieve C++ and Offload. Recent releases of the GNU tool chain and IBM XL offer compilers for C/C++ and FORTRAN on both architectures and support OpenMP for Linux platform.

III. The Virtual SIMD Machine (VSM) Paradigm

The VSM is an interface that designed to hide all the underlying details of the Cell BE architecture. The VSM paradigm is based on emulation techniques that imitate a SIMD instruction set on the SPEs using a Virtual SIMD Instruction (VSI) set and virtual registers. From the PPE point of view, the VSIs are implemented as stub routines that can be invoked once the required information, such as the virtual register number(s) and the starting addresses of the arrays to be processed, is supplied. A compiler can use this interface to evaluate arrays expressions on one or more SPEs automatically. Array languages compilers, in particular, can be extended to automatically incorporate the VSIs by decomposing high level array expressions into sequences of operations (micro-tasks) that can then be executed in parallel on the SPEs. The PPE stub routines are also available as a set of C API library routines. These API routines can be explicitly called from any programming language to perform array operations on the SPEs without the need to do any data partitioning, communication and synchronization processes.

A. Virtual SIMD Instructions

The VSIs are of two address register to register format. The VSIs are a set of RISC like register load, operate, store operations. The VSM register file consists of 8 virtual (vector) registers that can be used in computation operations or can be associated with DMA transfers to load and store data. From a compiler point of view, each VSI requires three PowerPC assembly instructions: One instruction supplies the destination/source virtual register's number. The second instruction loads either an effective address in case of the Load and the Store operations or the second virtual register's number which represents the second operand in a computation operation. The last assembly instruction is a call instruction that invokes a PPE stub routine.

B. VSM Message Protocol

The messaging protocol depends mainly on a mailbox mechanism to communicate between the PPE and the SPEs. The protocol consists of two distinct structures: forward messages and return messages. The forward messages are issued by the PPE to order the SPEs to execute an operation. The return messages (acknowledgments) are sent from the SPEs to the PPE. The messages can be in either one 32-bit word or two 32-bit words format; see Fig. 1 (a) and (b). The One-word format is used for computation operations while the two word format is used for memory access operations such as load and store. The messages must contain the operation code, the virtual register numbers, and, for Load or Store, a main store address.

C. The PPE Interpreter

The PPE interpreter is a set of stub routines that are basically responsible for data partitioning and communication. All stub routines have a similar task that can divided into four main steps:

- Partitioning data in blocks by computing the starting address of the data block to be used on each SPE.
- Combining passed parameters with a unique operation code and the starting address (if needed) into message(s).
- Writing the messages into the SPE's Inbound mailboxes.
- Waiting for a completion acknowledgment from the SPEs (if needed).

Most of the PPE routines were implemented in nonblocking mode to allow the PPE to continue its execution once the messages are delivered to the target SPEs and consequently allows overlapping operations. Blocking routines such as Store, however, stall the PPE until it receives acknowledgment with the completion of the requested operation from the SPEs, and they are relatively costly.

D. The SPE Interpreter

The SPE interpreter is a program that runs constantly on each SPE in the background. The program frequently checks if there is any message deposited into the SPE's Inbound mailbox. If a message dispatched into the inbound mailbox, the SPE program then pulls the message and the inbound mailbox will be automatically emptied. As soon as the message(s) is pulled, the PPE continues its execution, and the

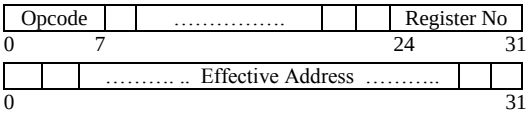

Figure 1 (a): Load and Store Messages

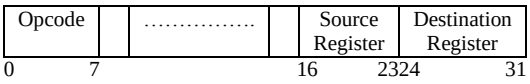

Figure 1 (b): Basic Operations Message

SPE program extracts the information sent within the pulled messages and starts performing the corresponding operation. If the requested operation is a blocking operation, the PPE then has to wait for an acknowledgment with the completion of the operation from the SPEs. The two very important issues that the SPEs have to handle during accessing memory are alignment and synchronization. We shall discuss these two issues shortly.

IV. Design Challenges

This section discusses the challenges encountered during the development of the VSM.

A. Virtua SIMD Registers

The VSM depends on DMA transfers to load/store data from/to main memory. The challenge here is to determine the appropriate DMA (virtual register) size. The size should be a compromise to balance the communication overhead and transfer costs. That is, not too big and not too small. We conducted a number of experiments for this purpose, and we found that the best virtual register size is 4*P KB where P is the number of the SPEs. The tests showed, on the other hand, using register smaller than 4KB degraded the performance because small data transfers do not hide DMA overhead cost.

B. Alignment

The alignment problem emerges here because of the architecture's memory alignment constraints. Alignment is also critical to performance because DMA must be aligned to a 128 bytes boundary for better performance. We developed two algorithms to handle the alignment on load and store operations. The algorithms can be used for any data type. In what follows, let define "MM_E" and "MM_A" as the Effective and the Aligned addresses on the main memory respectively, "LS_E" and "LS_A" to be the Effective and the Aligned addresses on the local storage respectively, VR_S to be the size of a virtual register and TB to be temporary buffer on the SPE of size VR_S+128, T is temporary buffer of size 128 bytes. Assume also that the register name is a starting address on an SPE's LS and that DMA transfers are aligned on 128-byte boundary on both sides.

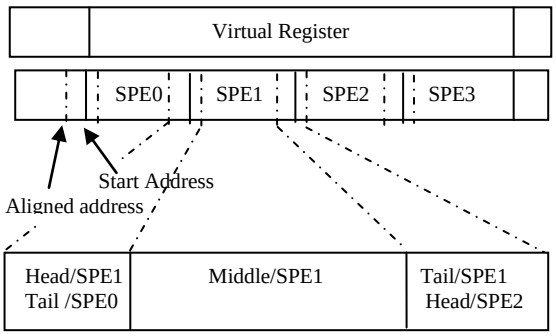

Figure 2: Splitting Data Block for Storing Process

1) Load Operation

The following algorithm handles the alignment problem when loading from main memory into an SPE's LS:

- Get effective address (MM_E)
- AB=MOD(MM_E,128)
- MM_A=MM_E - AB
- If AB = 0
  Get VR_S bytes start from MM_A into LS_E
- else
  Get VR_S+128 bytes start from MM_A into TB
  Copy VR_S bytes start from TB+AB to LS_E

2) Store Opertion

Again, due to the alignment constraints, writing back unaligned data from the SPEs requires reading and merging bytes from the main memory before rewriting those bytes as part of a DMA. We developed an algorithm that can be used by one or multiple SPEs to store unaligned data into main memory. The algorithm, as shown in Fig. 3, is based on operation-dividing technique in which each SPE's data block is divided into three parts: Head, Middle and Tail, see Fig. 2. The main advantage of this technique is that it allows overlapping DMA transfers within one SPE and among the used SPEs and suites well the architecture's resources.

The size of the Head and Tail was chosen to be 128 bytes for two reasons: First, all DMA transfers are aligned on a 128 bytes boundary, and secondly to tune our algorithm with the Cell's resources. The Cell offers atomic DMA operations, such as *getllar* and *putllc that* can be used to set, reserve and release locks on 128 bytes (cache line size). Given that the size of the Head and the Tail is an aligned 128 bytes block, the Middle portion size should be 128 bytes smaller than the virtual register size; that is, VR_S -128 bytes.

The algorithm for storing unaligned data is shown in Fig. 3, and it works as following: it performs an aligned read of 128 bytes including the first part of the block (the Head), updates it

- AB = MOD(MM_E,128)
- MM_A = MM_E - AB

// HEAD (PARTIALLY UPDATED)
- Set lock on 128 bytes start from MM_A
- Get 128 bytes start from MM_A into T
- Append the results to the tail of T
- PUT back T, and release the lock

// UPDATE THE MIDDLE PART
- MM_A = MM_A + 128
- LS_E = LS_E + 127 – AB
- MS = VR_S – 128
- Copy MS bytes start from LS_E to LS_A
- PUT MS bytes back start from LS_A to MM_A

// TAIL (PARTIALLY UPDATED)
- MM_A = MM_A + MS
- Set lock on 128 bytes start from MM_A
- Get 128 bytes start from MM_A into T
- Append the results to the head of T.
- PUT back T, and release the lock

Figure 3. Unaligned Store Algorithm

and then writes it back into main memory. It then aligns (if needed) the Middle part and writes it back into the main memory. The final step also performs an aligned read of 128 bytes including the last part of the block (the Tail), updates it and then writes it back into main memory.

C. Synchronizaion

In the alignment algorithm shown in Fig. 3, the Head and Tail of an SPE's block are apparently shared between the PPE and one of the SPEs or among the SPEs, and hence shared data, such as the Heads and Tails, must be over-written in a proper order. The synchronization process, in which we use locks to updating the Heads and Tails, is as follows: first an SPE requests to set a lock on 128 bytes from the main memory before reading it. Once the lock is granted, the SPE reserves the lock on the 128 bytes until it reads, updates and then writes the 128 bytes back. This synchronization process keeps memory coherent and avoids any race conditions that could be arise as different SPEs attempt to update the Heads and the Tails. The race condition problem is solved by reserving a lock on each 128byte until the granted SPE updates the data and then releases the lock.

D. Optimization

In regard to our VSM, storing unaligned data is considerably more costly than any other operation. For this reason, the algorithm for storing unaligned data was designed with the intention to optimize the instruction cost. The optimization technique we used here is based on the order of the three DMA transfers for storing the Head, Middle and Tail. The order of these DMAs, as presented by the algorithm in Fig. 3, provides some overlapping within one SPE and among the SPEs. The overlapping within one SPE, for example, occurs while storing the Middle part and the Tail. After an SPE issues a DMA request to the MFC for transferring the Middle part, the MFC takes control of the transferring process, meanwhile the SPE continues with processing the Tail by requesting a lock…etc. Once the Tail is stored back, the SPE should then verify if the Middle part has been completely transferred. The DMA's order also allows SPEs to overlap updating the Heads and Tails. This illustrated in Fig. 2, when SPE0 is locking its Head, SPE1 can be easily granted a lock on its Head too because the SPE1's Head is part of the SPE0's Tail.

V. PRELIMIARY RESULTS

This section first discusses the individual virtual SIMD instructions and their latency and then shows the performance of the VSM on basic linear algebra operations. The VSM interpreters and all testing programs were compiled, assembled and linked using GNU tool chain. We simulated a generated-compiler code using C code to test the effectiveness of our VSM in parallelizing arrays operations automatically. The C code explicitly calls the PPE stub routine. All tests were performed on a Playstation3 (PS3) running Fedora 7 Linux using single-precision floating point arrays and virtual SIMD register of size 4KB.

A. Virtual SIMD Instruction Latency

The average clock cycles per 32 bit word per virtual SIMD instruction, such as Load, Store and Arithmetic, is shown in Fig. 4. The arithmetic instructions are operate on single-precision float point values. The Load and arithmetic instructions costs are almost the same because they are non blocking instructions, but the Store is relatively costly because it is blocking instruction.

The cost of non blocking 32-bit instructions, such as load and arithmetic operation, vary between 1.1 clock cycles using one SPE to 0.6 cycles when four SPEs were used. The unaligned load operation, as you can see, costs slightly more than arithmetic operation because they are non blocking operations, but the variation is result of loading unaligned data which normally costs loading additional 128 bytes.

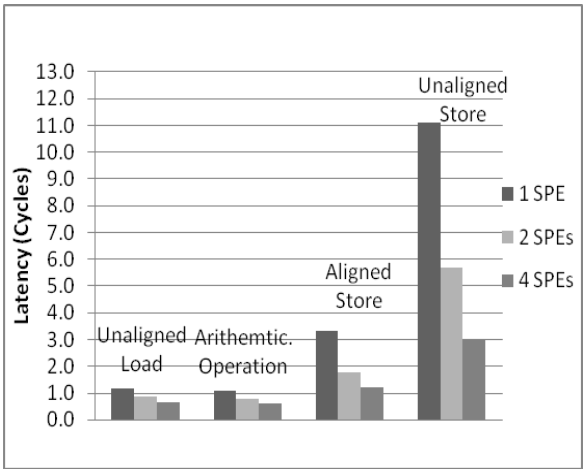

Figure 4. Virtual SIMD Instructions Latency

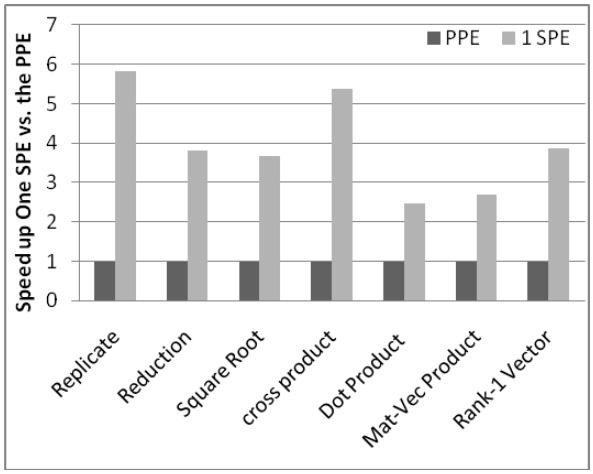

Figure 5. Performance of the PPE vs one SPE

On the other hand, The Store instruction is relatively costly because it implemented as a blocking operation and also handles the synchronization process. This explains why storing unaligned data using one SPE costs about 11 cycles that is up to 10 times as much as unaligned load and up to 5 times (3 cycles) when using 4 SPEs. The average cost of storing aligned 32-bit word is reduced be a factor of 3 as compared to storing unaligned 32-bit word. However, the Store instruction cost can relatively be reduced by combining as much operations as possible per an array expression.

Fig. 4 also points out that the cost of nonblocking instructions are reduced by roughly a factor of $p^{0.42}$ where p is the number of processors. Nonblocking instructions are not fully scalable because their low-latency does not allow a time window to overlap communication and reduce the cost. On the contrary, blocking instructions are almost fully scalable because the opportunity to overlap communication and data transfers is high. This can be seen clearly on the Store instruction latency which was reduced by a factor of $P^{0.94}$, where P is the number of processors.

Table 1: Basic Linear Algebra Kernels

Kernel Description	Expression	Virtual Instructions				
		Load	*Store*	*Multiply*	*Add*	*Square Root*
Replicate a scalar	$v_1 = s$	√	√	x	x	x
Vector Reduction	$s = redPlus(v_1)$	√	√	x	√	x
Square Root	$v_1 = sqrt(v_2)$	√	√	x	x	√
Cross Product	$v_1 = v_2 * v_3$	√	√	√	x	x
Dot Product	$s = v_1 . v_3$	√	√	√	√	x
Matrix-Vector Product	$v_2 = A * v_1;$	√	√	√	√	x
Rank-1 Update	$A = A + v_1 * v_2^T$	√	√	√	√	x

B. *The VSM Parallel Performance*

To assess the performance of our VSM, we run a number of BLAS-1 and BLAS-2 kernels on the Cell's two core types. The experiments include classical vector operations such as reduction operation, cross and dot products of two vectors and typical examples of BLAS-2 such as a matrix-vector product and rank-1 update operations [11]. Table 1 lists the BLAS kernels and the virtual instructions involved in evaluating each expression. The variables in Table 1 are of different ranks. For example, *A* is a square matrix of size *n x n*, while v_1, v_2 and v_3 are vectors of size *n* and s is a scalar. We tested the BLAS kernels using single precision floating-point arrays of size n = 4096) and virtual SIMD registers of size 4KB (1024 floating-point values) and run 10^6 times.

Before we discuss the VSM progress, it is important to point out that the reduction operation under our VSM is implemented as a blocking operation because the PPE has to wait for the used SPEs to return their sums. The SPEs use DMA mechanism to return the results. As the SPEs return their results, the PPE adds these results. Once all the SPEs finish, the PPE returns the sum. As a result of that kernels that require such operations are expecting to degrade the SPE's performances.

Fig. 5 presents the performance of the selected BLAS kernels when run on one SPE relative to the PPE. The achieved speedups on the different kernels using one SPE compared to the PPE range from 2.5 times to 6 times. The average SPE performance on the Reduction, Dot product, Matrix-vector product and Rank-1 Update kernels was around 3 times as fast as the PPE. The SPE performance was degraded here because each kernel requires calling to a reduction operation (blocking operation) which is relatively costly as was mentioned in the previous section. The SPE performance on the Square Root operation is also slower than its performance on the Reduction and Cross Product because the square root operation is considered as a complex SPE operation [7]. The best performance, however, was achieved

```
// Replicate  value "1.25" into the SPE's virtual register (VR0)
        repVec(0,1.25);

// Load vector v1 from the PPE into the SPE's VR1
        loadVec(1,v1);

// Perform Cross Product on the SPEs VR0 & VR1
// and keep the result in the SPE's VR 0
        mulVec(0,1);

// Store  the contents VR0 into vector v2 on main memory
        storeVec(0,v2);

// Perform reduction operation: load  vector v2 in a VR of size N,
// the  sum the elements of  VR and return the sum into variable S.
        S = redPlus(v2,N);
```

Figure 6. A Sample C Code for Calling the VSM Routines

on the Replicate and Cross Product kernels because their evaluations do not require a call to a blocking operation a side of the storing operation which is performed by all kernels.

Thus far, we have looked at the performance of a single SPE as compared to the PPE. To explore how the VSM parallelizes array operation automatically on multiple SPE, we run the same kernels given in Table 1 in parallel using 2 and 4 SPEs. We used 2 and 4 for divisibility purposes. The length of the virtual register that is used in these experiments was 4KB. The size is not dividable by 6, and we can not used 8 SPEs because only 6 SPEs are available on the PS3. Fig. 6 shows a sample of the C code that utilizes our VSM as API routines. Fig. 7 shows the scalability attained using multiple SPEs. The speedups obtained from using multiple SPEs was near-linear. The VSM showed near-linear performance with an overall speedup of a factor $P^{0.84}$ where P is the number of processors. The maximum speed up, however, gained on the Square Root kernel, it was by a factor of $P^{0..94}$ and the minimum was about $P^{0.75}$ on the rank-1 Update kernel.

VI. CONCULSION

We presented here a VSM approach to abstract and hide all the details of the Cell heterogeneity. The VSM can be used as an API by programming languages such as C, but it is also a suitable target code for optimizing array language compilers. We have integrated it into the Glasgow Vector Pascal compiler[9] where it allows us to parallelize array operations automatically. Space precludes discussion of this use of the VSM in the current paper. Suffice to say here that this has allowed unchanged applications code to be ported between and automatically parallelized on both Intel multi-core and Cell architectures. This approach eases the task of developing programming parallel applications by concentrating on algorithms rather than on parallelization process. The preliminary results to be reported elsewhere show that this approach can be used to parallelize data-intensive applications across both homogenous and heterogeneous multi-cores chips.

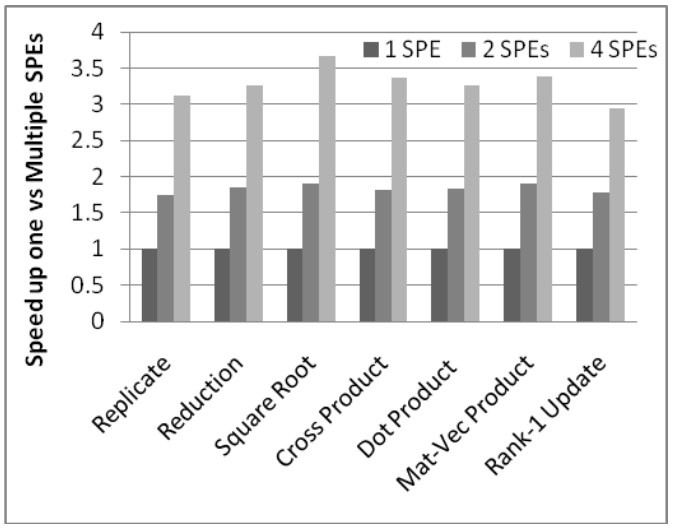

Figure 7. The SPEs Performance

The performance of new versions of VSM can be improved by using special intrinsic functions, such as Multiply and Add function, because many basic linear algebra kernels require a reduction operation.

References

[1] Liu, F., Chaudhary, V. 2003. A practical OpenMP compiler for system on chips. In *Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT)*, 54–68.

[2] Breitbert, j., Fohry, C., OpenCL, An affective programming model for data parallel computations at the Cell BE, Parallel & Distribute Processing, Workshop and PhD Forum (IPDPSW) (2010), pp. 1—8

[3] Olukotun, K. AND Hammond, L., A Future of Multiprocessors, ACM Transactions on Embedded Computing SysteA_EA, , Publication date: April 2008 (2005), pp. 26—29

[4] Scheinine A., Introduction to Parallel Programming Concepts, Louisiana State University (2009)

[5] Kahle, J.A. and Day, M.N. and Hofstee, H.P. and Johns, C.R. and Maeurer, T.R. and Shippy, D., Introduction to the Cell multiprocessor, IBM journal of Research and Development (2005), pp. 589—604

[6] Ami Marowka, Performance of OpenMP on Multicore Processors (2008), pp. 208--219

[7] Sandeep, K., Practical Computing on the Cell Broadband Engine, Springer, 2009.

[8] Arevalo, A. et al., Programming the Cell Broadband Engine™ Architecture, International Technical Support Organization (2008)

[9] Jackson I., Opteron Support for Vector Pascal, Dept Computing Science, University of Glasgow (2004)

[10] Cockshott, P. and Renfrew, K., SIMD programming for Windows and Linux, Springer (2004)

[11] http://www.cs.indiana.edu/classes/b673/notes/blas2.html