

# A Polymorphic Register File for Matrix Operations

Cătălin Ciobanu, Georgi Kuzmanov, Georgi Gaydadjiev  
Computer Engineering Laboratory,  
Electrical Engineering Department,  
Delft University of Technology, The Netherlands  
{c.b.ciobanu, g.k.kuzmanov, g.n.gaydadjiev}@tudelft.nl

Alex Ramirez  
Barcelona Supercomputing Center  
Universitat Politècnica de Catalunya, Spain  
alex.ramirez@bsc.es

**Abstract**—Previous vector architectures divided the available register file space in a fixed number of registers of equal sizes and shapes. We propose a register file organization which allows dynamic creation of a variable number of multidimensional registers of arbitrary sizes referred to as a Polymorphic Register File. Our objective is to evaluate the performance benefits of the proposed organization. Simulation results using real applications (Floyd and CG) suggest speedups of up to 3 times compared to the Cell SPU for Floyd and 2 times compared to a one dimensional vectorized version of the sparse matrix vector multiplication. Moreover, in the same experimental context, a large reduction in the number of executed instructions of up to 3000 times for Floyd and 2000 times for sparse matrix vector multiplication is achieved.

**Index Terms**—Vector processors, Vector register file, Polymorphism, Cell, Vector ISA

## I. INTRODUCTION

In classic vector architectures such as the IBM System/370 [3], all the vector registers were of equal sizes and the number of data elements which can be stored in one vector register is a micro-architectural parameter, termed as the *section size* of the machine. The ISA did not impose the value of the section size, which could be chosen as any power of two between 8 and 512 at design time. This allowed for high end and low end machines to share the same architecture at different cost and performance levels. More recently, the Single Instruction, Multiple Data (SIMD) extensions of General Purpose Processors (GPP) used fixed width vector registers (128-bit wide for AltiVec) which can store multiple data elements. This is also the case with the Cell Broadband Engine processor [12], which is comprised of one PowerPC Processor Unit (PPU) and eight Synergistic Processor Units (SPUs): both the PPU and the SPU use 128-bit wide SIMD registers. The number of the available vector registers is clearly defined in the architecture (16 vector registers for the IBM/370, 32 AltiVec registers for the Cell PPU, 128 registers for the Cell SPU).

In this paper, we propose a register file organization referred to as a Polymorphic Register File (PRF), which targets the efficient processing of multidimensional data arrays. The total

size of the register file amounts up to a fixed volume, while the actual number of registers as well as their dimensions and sizes can be defined by the programmer and adjusted during runtime. As a result, multiple register sizes and dimensions can be used simultaneously. Currently, only 1D and 2D rectangular matrices are supported by the architecture but our proposal can be extended for any number of dimensions and even for some special data access patterns. In this paper, we do not analyze any possible underlying micro-architectural implementations. The main contributions of this paper are the following:

- Architectural definition of the Polymorphic Register File;
- Performance evaluation of the proposed Register File (RF) organization using kernels from real applications;
- Significant reduction in the number of executed instructions: 3000 times for Floyd and 2000 times for sparse matrix vector multiplication compared to a baseline PowerPC scalar processor;
- Speedups of up to 3 times for Floyd compared to the Cell SPU and up to 2 times compared to one dimensional vectorized version of the sparse matrix vector multiplication kernel;
- Simplified programming interface, which allows potentially lower programming efforts compared to traditional methods.

The rest of the paper is organized as follows: in Section II, we provide motivation for the Polymorphic Register File. The proposed architecture is presented in Section III. In Section IV, we present the targeted applications, and simulation data are presented in Section V. Section VI describes other related architectures. Finally, the paper is concluded in Section VII.

## II. PROBLEM STATEMENT AND PROPOSED SOLUTION

In order to get an indication of the impact of the Register File (RF) used in a processor architecture in terms of the static code size and the number of executed instructions when performing matrix multiplication, we analyze three different cases: a scalar and two types of vector processors, one equipped with one dimensional (1D) vector RF and the other - with two dimensional (2D) RF. Furthermore, we illustrate the need for a polymorphic RF.

**Motivating example:** Assume the product computation of two double precision floating point matrices:  $A[2][4]$  and  $B[4][3]$  with the result stored in  $C[2][3]$ . By compiling the C

This work was partially supported by the European Commission in the context of the Scalable computer ARchitectures (SARC) integrated project #27648 (FP6), the Dutch Technology Foundation STW, applied science division of NWO, the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533) and the Consolider contract TIN2007-60625 from the Ministry of Science and Innovation of Spain.

code of this matrix multiplication for the Cell PPU, the scalar PowerPC assembly code contains **41** instructions, as follows:

```
#Init, Addr gen & looping
- 9 instructions
.L1:          #i-loop
#Addr gen & looping - 7 instructions
.L2:          #j-loop
#Addr gen, memory & looping
- 10 instructions
.L3:          #k-loop
ldf 13,0(10)  #Memory (ld A[i][k])
ldf 0,0(8)    #Memory (ld B[k][j])
fmadd 12,13,0,12 #Processing
stfdx 12,6,31 #memory (st C[i][j])
#Looping & Addr gen - 11 instructions
```

---

Total: 41 instructions / 269 committed

By performing a 1D vectorization of the above kernel using the ijk form [7], the inner most loop can be replaced with a vector dot product instruction, decreasing the static code to 26 instructions:

```
v.setv1 4      #set Vector Length
#Addr gen & looping - 7 instructions
.L1:          #i-loop
#Addr gen & looping - 7 instructions
.L2:          #j-loop
v.ld $V0, &A[i][0] #Memory
                # (ld A[i][*])
v.ldstrided $V1, &B[0][j], 3 #Memory
                # (ld B[*][j])
v.vdot 0, $V0, $V1 #Processing
stfd 0,0(10)    #Memory (st C[i][j])
#Looping & Addr gen - 7 instructions
```

---

Total: 26 instructions / 72 committed

By storing each matrix in an assumed 2D register, we can possibly replace all three loops with a matrix multiplication instruction:

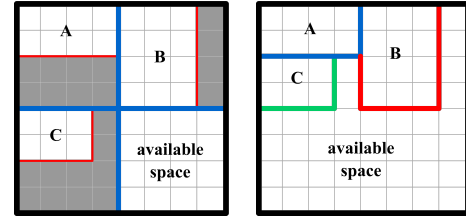
```
#Addr gen - 6 instructions
v.vdefvr $V0, 2, 4, 0 #def 2D reg A
v.vdefvr $V1, 4, 3, 4 #def 2D reg B
v.vdefvr $V2, 2, 3, 16 #def 2D reg C
v.setxvl 4          #set Vector Length
v.setyvl 4
v.ld2d $V0, 10      #memory (ld A)
v.setyvl 3
v.ld2d $1, 11       #memory (ld B)
v.mul2d $V2, $V0, $V1 #Processing
                # (C = A * B)
v.st2d &C[0][0], $V2 #memory (st C)
```

---

Total: 16 instructions / 16 committed

Rough estimations suggest that for this simple matrix mul-

tiplication example, by using a 1D vector processor and a 1D-organized storage, the number of assembly instructions is reduced by 1.5 times and by utilizing a 2D Polymorphic RF we can further shrink the static code length required to perform the same computation by a factor of 2.5. The difference is even higher if we consider the committed instructions: the 1D vector processor executes 3.7 times less instructions than the GPP by eliminating the inner-most loop, while a 2D polymorphic processor could execute 16 times less instructions than the baseline scalar machine by replacing all three loops with a single matrix multiplication instruction. This indicates that by using a 2D vector register file, we can vectorize the code on a second axis and therefore eliminate two or more nested loop levels. Code size is potentially reduced as well as the number of executed instructions.



(a) Predefined partitioning (b) Using Polymorphism

Fig. 1. Storing the three matrices in a 2D RF

Traditionally, all the vector registers defined by an architecture are of the same size. With the preceding example we illustrated how a 2D RF and appropriate code vectorization can potentially improve performance. Hereafter, we further investigate the possibilities for optimization in terms of RF space utilization. The minimum size of the two dimensional registers needed to store the three matrices of 2x4, 4x3 and 2x3 elements from the motivating example is 4x4 elements. Figure 1(a) suggests how a two dimensional register file of eight by eight elements can be statically partitioned into fixed size registers to store the two operands and the result from our example. The shaded portions indicate the unused space in the registers: 50% of register A, 25% of register B and 62.5% of register C do not contain any useful data. The result of the illustrated example is a waste of more than 45% of the allocated register file capacity if 2D registers with fixed size are used.

**Problem statement:** Based on the motivating example above, three problems can be identified, which we address in this paper, namely:

- 1) Performance is limited by the memory transactions and by the additional address arithmetic required from the programmer when a scalar architecture is employed for 2D matrix operations.
- 2) Operations on 2D matrices require excessive number of committed instructions when a scalar register file organization is implemented. When 1D and 2D register files are assumed, a potential reduction of the static code size and the number of committed instructions can be achieved.

- 3) Fixed size 2D register files can potentially alleviate problems 1 and 2, but they impose inefficient utilization of the storage space.

**Proposed solution:** We address the above problems and propose a so-called Polymorphic Register File to solve them at the architectural level.

**Definition:** a *Polymorphic Register File* is a parameterizable register file, which can be logically reorganized by the programmer or the runtime system to support multiple register dimensions and sizes simultaneously.

An illustration of how a polymorphic register file with the same total storage capacity, as the one in the motivating example, can be repartitioned is provided in Figure 1(b). The programmer's target is that each matrix fits perfectly into the logical register, so that the storage waste is minimized and hence more registers can be instantiated. It is further proposed that the dimensions of the logical registers can be changed during runtime therefore all the vectors and matrices stored in the polymorphic register file can fit and be processed more efficiently.

Based on the motivation example above, we can clearly identify several potential benefits from using a 2D polymorphic register file:

- Advantages of a **2D register file**:
  - **Potential performance gain** - reduced number of committed instructions. For the motivation example, the reduction by a factor of 4.5 of the committed instructions compared to the 1D vector processor, which in turn further reduces the Flynn bottleneck [9].
  - **Static code compression** - the assembly code using the 2DPRF consists of less instructions compared to a vector processor with 1D register organization and even lesser compared to a machine with a scalar RF;
- Advantages of a **polymorphic register file**:
  - **Storage efficiency** - the registers were defined to contain exactly as many elements as required, completely eliminating the potential storage waste of organizations with fixed register sizes and maximizing the available space for subsequent use;
  - **Variable number of registers** - unlike the traditional fixed set of registers of predefined size, the unallocated space can be further partitioned into an arbitrary number of registers of arbitrary dimensions.

We envision the occurrence of data fragmentation problems in the register file, which could be handled dynamically by the Operating System, or statically by the compiler. The problem is similar to ordinary memory fragmentation, with the additional complexity for a 2D space, rather than for a linear 1D case.

In the rest of this paper we will focus on evaluating the potential performance benefits due to the use of the 2D Polymorphic Register File for matrix operations.

### III. THE POLYMORPHIC REGISTER FILE ARCHITECTURE

Figure 2 illustrates the organization of the register file, assuming that the dimension of the physical register file is 128x128 elements. The logical registers are defined using the Register File Organization (RFORG) Special Purpose Registers (SPR). When defining a logical register, we need to specify the Base, the Horizontal Length and the Vertical Length of the register. The Base of a register can be computed from the 2D coordinates of the upper-leftmost element of the register. The D flag indicates whether a logical register has been defined and RN is the total number of available logical registers. The power-on organization can, for example, partition the available storage in 16 logical registers containing 32x32 64-bit elements. A special instruction restores the configuration of the register file to the initial state.

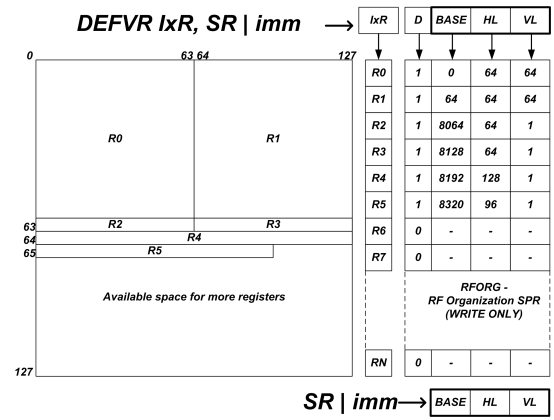


Fig. 2. The Polymorphic Register File

Both 1D and 2D register operations are supported simultaneously, using the same binary instructions. The microarchitecture checks if the dimensions of the operands are compatible with the semantics of the instruction (for example, the number of columns of the first register must be equal to the number of rows of the second register when performing matrix multiplication). A Bit Mask register is implicitly defined for each logical register, offering support for conditional processing. A special instruction enables or disables the masked execution mode, therefore the same instruction opcodes are used for both masked and non-masked mode. By adding 3 additional bits to each entry of the RFORG table, we can also specify the data type stored in the logic register (32/64-bit floating point or 8/16/32/64-bit integer), avoiding the duplication of the instructions for each data type.

We have defined the following instruction formats concerning the Polymorphic Register File: Vector Register to Memory (**VR-M**), Vector Register to Vector Register (**VR-VR**), Vector Register to Scalar reg (**VR-SR**), Vector Register to Bit Vector (**VR-BV**) and Scalar Register to Special Purpose Register (**SR-SPR**).

The **VR-M** format is used by the memory instructions such as load and store, and the instruction prototype is  $Vmem \langle dest \rangle \langle src \rangle \langle offset \rangle$ . The memory address and

the offset are provided in scalar registers, while the third argument is a logical vector register defined in the PRF. The **VR-VR** format is used by the arithmetic instructions such as add or multiply, with an instruction prototype `Varith <dest><src1><src2>`. All arguments are logical registers defined in the PRF. The **VR-SR** format is used by reduction operations (such as compute the maximum value). The format is `Vreduction <dest>, <src>`, where the first argument is a scalar register and the second is a PRF register. The **VR-BV** format is used by the operations which produce a bit vector as a result, such as comparing two vector registers. The format is `Vbit <dest>, <src1>, <src2>` where the first argument is the destination bit vector PRF register and the others are PRF vector registers. The **SR-SPR** format is used by instructions which write in the special purpose registers. The format is `Vspr [<src1>, <src2>, <src3>, <src4>]`. A subset of the supported instructions is presented in Table I.

TABLE I  
POLYMORPHIC RF - A SUBSET OF THE SUPPORTED INSTRUCTIONS

Instruction	Format
ld2d VRT, RA, RB	VR-M
st2d RT, VRA, RB	VR-M
add2d VRT, VRA, VRB	VR-VR
mul2d VRT, VRA, VRB	VR-VR
max2d RT, VRA	VR-SR
vcompare2d_gt BVT, VRA, VRB	VR-BV
set_2dmask RT	SR-SPR
resetconf	SR-SPR
vdefvr RT, RC, RA, RB	SR-SPR
setxvl RT	SR-SPR
setyvl RT	SR-SPR
update2d RT, VRA, RB	SR-SPR

One important aspect of vector architectures is the reduced number of address generation instructions required to process data of any size and allow the same code to be executed on machines sharing the same architecture but constructed with different section sizes. Sectioning is a technique used for processing vectors which are larger than the section size of the machine in batches equal in size to the number of elements which can be stored into a vector register, called sections. The last section may be shorter if the size of the data is not a multiple of the section size.

We have extended the sectioning mechanism presented in the IBM/370 vector architecture [16] in order to support two-dimensional vector registers of arbitrary sizes. The polymorphic nature of our proposal implies that there is no unique section size defined for an implementation of the architecture. Since at the moment the proposed Polymorphic Register File architecture supports unidimensional and bi-dimensional vectors, we define the notion of horizontal and vertical section sizes corresponding to the dimensions of a logical vector register in the X and Y directions.

In order to define the dimensions of the data, which are being processed from the external memory, the Polymorphic Register File Architecture defines two SPRs, the X Vector Length(**XVL**) and Y Vector Length(**YVL**) registers. The **setxvl** and **setyvl** instructions are used to set the values in

those registers. The Horizontal Size (**HSIZE**) and Vertical Size (**VSIZE**) registers limit the number of elements which are processed by each vector instruction when sectioning is employed.

The **update2d** instruction is placed at the end of a sectioning loop and has three parameters: 1) the number of instructions, which comprise the sectioning loop; 2) a logical vector register (the **model** register), which is used as a reference to how many elements are processed in the current section in each dimension; and 3) a scalar offset register, which is used to keep track of the position of the current section in the two-dimensional data being processed as well as being used by the load and store instructions as an offset to the memory addresses. The **update2d** instruction performs three tasks:

- Increment the offset register so that, by being added to the memory pointers, the result points to the first element of the next section;
- Update **HSIZE** and **VSIZE** taking into account how much data has been already processed in the X and Y directions. This is done by comparing the dimensions of the model register with the difference between **XVL**, **YVL** and the offset register;
- Decide if the current sectioning loop is completed by comparing the current offset with the **XVL** and **YVL**. If the loop is not done, branch to the first instruction of the sectioning loop. The number of instructions needed to jump is a parameter of the instruction.

#### IV. TARGET ALGORITHMS

We consider two target algorithms for our evaluation, namely Floyd and the Conjugate Gradient (CG) Method.

**Floyd:** This algorithm finds the shortest paths between all pairs of vertices in a weighted, directed graph [8], [6]. Given a graph  $G = (V, E)$  with N nodes and a NxN weight matrix W, the algorithm computes the cost matrix **d**, where  $d_{ij}$  represents the shortest path from node i to node j and **path**, the predecessor matrix.

The algorithm uses a dynamic programming approach in which the result is refined over N steps. For each step of the algorithm, the two inner loops update each element of the weight matrix d at most once. The algorithm compares the current cost ( $d_{ij}$ ) with the cost of using k as an intermediate node in the path ( $d_{ik} + d_{kj}$ ).

In order to update all the cost matrix d at step k, we only need to use the k-th line and the k-th column of d - the pivot row and column. For line k,  $\forall j, d_{kj} = \min(d_{kj}, d_{kk} + d_{kj})$  and for column k,  $\forall i, d_{ik} = \min(d_{ik}, d_{ik} + d_{kk})$ . Because  $d_{kk}$  is always 0, the pivot row and column are not updated in step k and therefore the code doesn't have read after write dependencies for the two inner loops and vectorization is possible on both dimensions of the cost matrix.

Figure 3 presents the reduction in the number of steps required to compute the two inner loops of the algorithm, assuming no data segmentation is required. When executed on the scalar processor, each step requires  $N^2$  iterations to complete the algorithm. When we vectorize the code on a 1D

vector machine, each line of the cost matrix can be processed with one set of vector instructions. In the third case, we can store the whole cost matrix in a 2D register and we only need one iteration to complete a step of the algorithm.

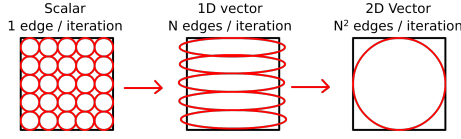


Fig. 3. 1D and 2D vectorization of Floyd

When the cost matrix doesn't fit in the 2D register, we can process it in blocks [10]. Figure 4 presents the data required to process a block of the cost matrix. Apart from the block itself, we also access the corresponding part of the pivot row and column.

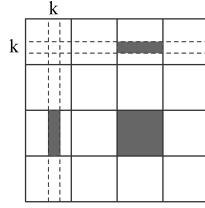


Fig. 4. Floyd: cost matrix segmentation. The data required to process the block is shaded

**CG:** This algorithm is part of the NAS Parallel Benchmarks [2]. CG is one of the most commonly used iterative methods used for solving systems of linear equations whose matrix is symmetric and positive definite [20]. Because CG is an iterative method, it can be applied to sparse systems that are too large to be handled by direct methods.

The main computational kernel in CG is Sparse Matrix - Dense Vector multiplication, accounting for 87.32% of the execution time in the scalar version of CG. The format used to store the sparse matrices is Compressed Sparse Row (CSR). Two dimensional vectorization is possible in this case by either processing multiple rows of the input matrix with the same instruction (which is useful especially when the number of non zeros per row is small) or by splitting a long row of the input matrix into multiple lines of a 2D register.

The polymorphism of the Register File is useful in this kernel, because we have the ability to resize the logical registers during runtime. Unlike Floyd, where each row had a constant number of elements, the number of non zeros in each row of the sparse matrix is not constant. We added two specialized instructions to the ISA in order to facilitate 2D vectorization: a zero padding instruction which is able to map a 1D register containing multiple rows of the sparse matrix to a dense 2D register by zero padding, and a row dot product instruction which performs the dot product of the input registers row by row.

One particularity of this kernel is that the register file has to accommodate multiple data types at the same time: the

sparse matrix and the dense vector contain 64-bit floating point numbers, but the CSR format is composed of two extra vectors storing the index of the non-zeroes as well as the length of each row of the matrix, which are stored as 32-bit integers.

## V. EXPERIMENTAL RESULTS

We have implemented the Polymorphic Register File as part of a Scientific Vector Accelerator (SVA) within the SARC architecture using a cycle accurate simulator written in Unisim [1], an extension of SystemC. Figure 5 presents a block diagram of the simulator, where: P = scalar Processor, LS = Local Store, LSC = LS Controller, SVA = Scientific Vector Accelerator.

The Processor module implements the instruction set of the PowerPC Processor Unit (PPU) in the Cell processor [14] and it is based on the PPU model found in CellSim [4]. Because the Processor can access both cacheable and non cacheable data, it uses a dedicated connection to the Arbiter Bus when accessing the LS, which only contains non cacheable data. We assume each scalar instruction is executed in one cycle. We further assume perfect Instruction and Data caches having one cycle latency.

The Vector Accelerator is designed as a loosely coupled coprocessor. The communication between the Processor and the Accelerator is facilitated by a number of memory mapped control and data exchange registers which are accessed via the Arbiter Bus. At this moment, we do not support parallel execution of the Processor and the SVA modules. The processor is responsible for starting the SVA when the vectorized section of code is reached. The SVA assumes that all data are available in the Local Store, and the performance results do not measure the data transfer overhead between the main memory and the Local Store. The execution model assumes that two fully pipelined arithmetic units, with 5 stages for the addition operations and 12 stages for multiplication, are available for each vector lane. When accessing the vector register file, we assume that the address generation is done in a single cycle, but in a general case it could be pipelined in more cycles. We further assume that the number of ports available to the register file is sufficient to provide data to all vector lanes.

The SVA sends complex memory requests such as 1D/2D contiguous and strided accesses to the LSC, which is responsible for splitting these requests into smaller, simpler ones, which are then sent to the LS as well as modeling the appropriate latencies, which occur when performing complex memory accesses.

The latency of a memory access from the vector accelerator to the Local Store was set to 16 cycles in order to take into account the overhead incurred by the 2D memory accesses. Because all the modules connected to the Arbiter Bus must have the same bus width of 4 bytes, the SVA bypasses the Arbiter Bus when accessing the LS by using a dedicated 16 bytes wide link to the LSC, equal to the bus width used in the Cell processor between the SPU and the LS.

We have simulated the matrix multiplication example presented in Section II in order to quantify the number of extra

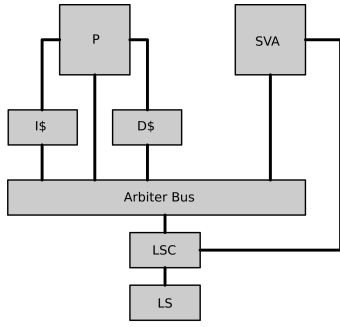


Fig. 5. The simulator block diagram

instructions required to send the parameters from the GPP to the Vector Accelerator as well as the reduction of the number of executed instructions when using the 2D PRF compared to the baseline scalar processor. In section II, we have estimated that by using a 2D Polymorphic RF, the exemplary matrix multiplication can be performed with 16 instructions, out of which 6 performed address generation. As it can be observed in Table II, sending the required parameters from the GPP to the co-processor requires 12 address generation, 16 initialization and 2 co-processor control instructions, so the total number of instructions becomes 40. The static code size including the parameter passing overhead is then similar to the static code size for the GPP, but the number of committed instructions is more than 6 times lower. The reason is that the looping instructions are completely eliminated and the vectorization on the second axis allowed us to encode the loading, storing and the matrix multiplication as just one instruction each.

Because our proposal mainly targets high performance execution of scientific workloads in which the amounts of data processed are significantly higher compared to the small data set used in our matrix multiplication example, we consider that our choice of a loosely coupled co-processor is feasible. For an embedded design, a tightly coupled or completely integrated approach might be more suitable.

TABLE II  
STATIC CODE SIZE AND COMMITTED INSTRUCTIONS FOR THE MATRIX MULTIPLICATION EXAMPLE

ISA	PowerPC	PowerPC + 2D PRF	PowerPC	PowerPC + 2D PRF
<b>Category</b>	<b>(static)</b>		<b>(committed)</b>	
Memory	6	3	72	3
Processing	1	1	24	1
Address generation	21	12	108	12
Looping	11	0	63	0
Initialize	2	16	2	16
Setup RF	0	6	0	6
Control coprocessor	0	2	0	2
<b>Total</b>	<b>41</b>	<b>40</b>	<b>269</b>	<b>40</b>
Compression rate	-	1X	-	6.7X

In the Floyd benchmark, the data type for the cost matrix is 32-bit integer. The instruction compression rates without taking into consideration the initial parameter passing and the

outer loop overhead are presented in Table III(a).

TABLE III  
CODE COMPRESSION RATES

(a) Floyd 64x64		(b) 1D SMVM	
2D Reg. Size	Code comp. rate	2D Reg. Size	Code comp. rate
1x8	6	1x8	11
1x16	13	1x16	18
1x32	27	1x32	26
1x64	55	1x64	34
2x64	110	1x128	37
4x64	219	1x256	37
8x64	434		
16x64	853		
32x64	1646		
64x64	3073		

The number of instructions committed by the vector accelerator is significantly lower compared to the same benchmark running on the PPU, with estimated compression rates of more than 3000 times. Figure 6 presents the speedups obtained by using the 2D polymorphic register file compared to the PowerPC processor. Speedups of up to 16 times can be achieved when 16 vector lanes are used and the vector registers can store up to 64x64 elements. This provides additional scaling when comparing with the maximum speedup obtained by only using a 1D register - 9X in the case of the 1x64 register. It can be observed that 4 vector lanes are sufficient to sustain 89% of the peak performance. The scalability of the vector accelerator is limited mainly by the available Local Store bandwidth.

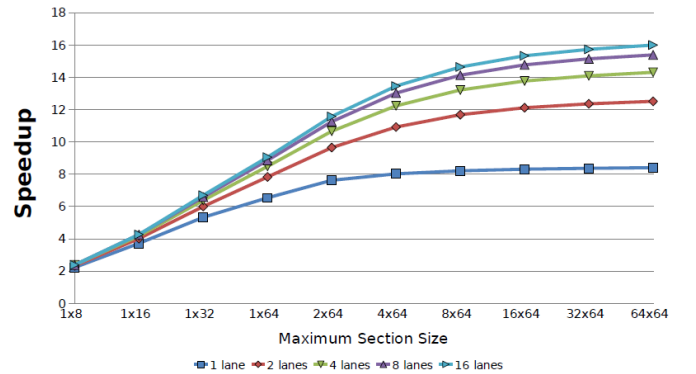


Fig. 6. Speedup of the 2D vectorized Floyd over the scalar implementation

Benchmark results against the Cell SPU using a Playstation3 (PS3) indicate that our register file architecture is around 3 times faster when running Floyd 64x64 with 16 vector lanes (Figure 7) compared to the Cell SPU. We assumed equal clock frequency and we limited the total size of the Polymorphic Register File and the Local Store used by our Scientific Accelerator to 256KB - the Local Store size in the SPU. We note that 4 vector lanes x 32 bits require the same throughput as the Cell 128-bit SPU interface. Thus, assuming identical bandwidth for our proposal and a Cell SPU, we

obtain 2.7 times higher performance. The PS3 SPU code runs the benchmark 4.33 times faster compared to the 32-bit scalar PS3 PPU, without considering the DMA transfer time to the Local Store.

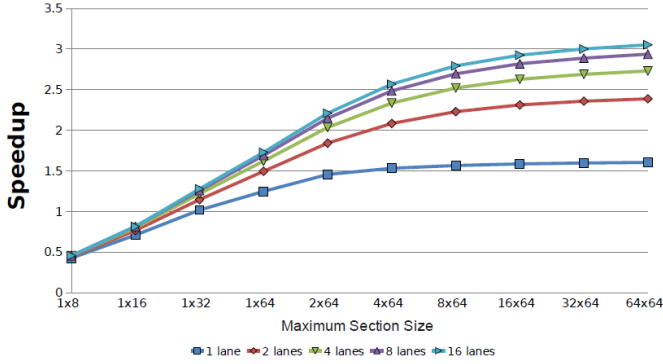


Fig. 7. Speedup of the 2D vectorized Floyd over the Cell SPU

TABLE IV  
CODE COMPRESSION RATES FOR 2D SMVM

(a) Group short rows

2D Reg. Size	Code comp. rate
1x16	6
2x16	7
3x16	9
4x16	11
5x16	13
6x16	13
8x16	13
12x16	13
16x16	13

(b) Split long rows

2D Reg. Size	Code comp. rate
1x128	16
2x128	32
4x128	65
8x128	131
16x128	260
32x128	518
64x128	1029
128x128	2024

By performing a one dimensional vectorization of the Sparse Matrix Vector Multiplication (SMVM) kernel of CG, we can obtain a speedup of approximately 4.9 times compared to the scalar processor for the CG class S test (Figure 8). The experiments indicate that 94% of the peak performance can be obtained by using just 4 vector lanes. The SVA allows us to switch off the unused lanes, therefore saving power. The performance saturates for a section size of 128 elements or more. The reason is that the average row length is 56, with a minimum of 8 and a maximum of 127. Figure 9 presents the distribution of the row lengths in the sparse matrix. The one dimensional vectorization leads to committed instructions compression rates of up to a factor of 37 (Table III(b)).

Our first strategy to perform 2D vectorization is to split a long row of the matrix into multiple lines of the 2D register. As it can be observed in Figure 10, this doesn't improve the performance over the 1D vectorization. The executed code compression rates are also smaller (Table IV (a)). The main reason is that there are not sufficient long rows to compensate for the number of overhead instructions needed to fit the long

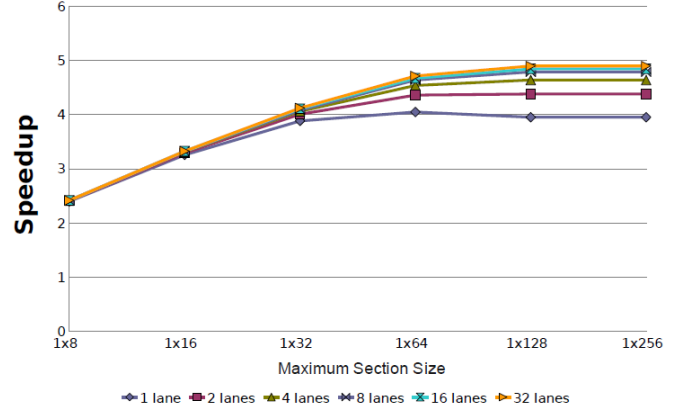


Fig. 8. CG SMVM Speedup 1D vs. scalar, class S

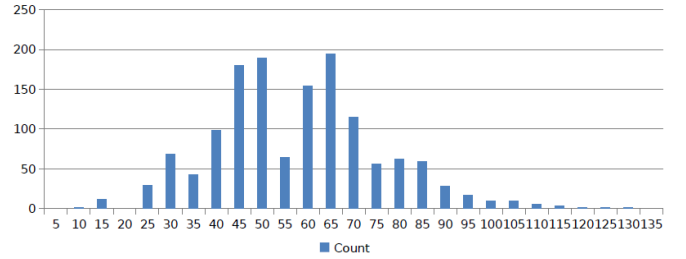


Fig. 9. CG SMVM: Distribution of the row lengths, class S

rows into the 2D registers. In order to get similar performance as the 1D vector accelerator with a section size of 16 we need to have a register with 5 lines of 16 elements each and use 4 vector lanes, which can sustain around 94% of the peak performance.

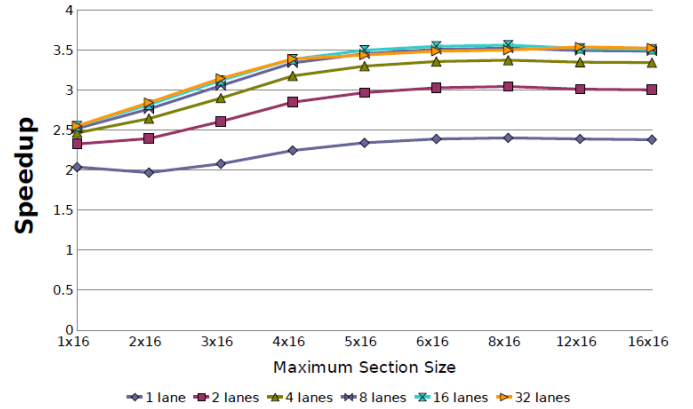


Fig. 10. CG SMVM 2D Speedup vs. scalar, hsize=16, class S

The second strategy is to store multiple consecutive rows of the sparse matrix in a two dimensional vector register. As it can be seen in Figure 11, we can increase the performance of our PRF-enabled vector accelerator to around 10.6 times, twice as fast as the 1D vector implementation. Using 4 vector

lanes is sufficient to sustain 80% of the peak performance. The instruction compression rates are also significantly higher, reaching more than a factor of 2000 (Table IV (b)). Even if the number of independent elements, which can be processed with a single instruction, is significantly increased by using a 2D register file, the overhead operations required to arrange the data in rectangular registers (the zero padding instructions) partially offset the potential reduction in execution time. In Figure 11, it can be observed that this algorithm scales up 16 vector lanes which deliver up to 96% of peak performance, which is a substantial improvement compared to the 1D vectorized version or the strategy which splits long rows into multiple lines of the 2D register file.

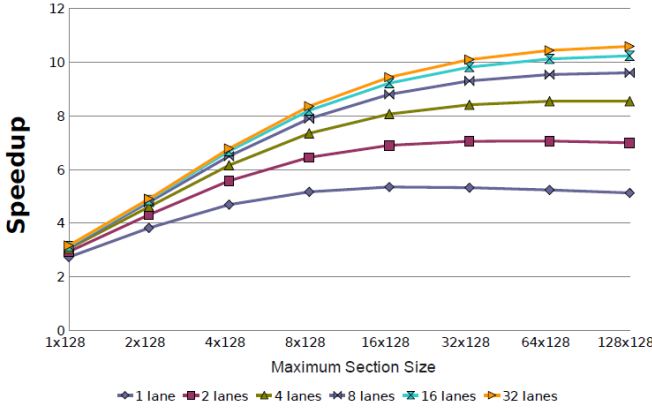


Fig. 11. CG SMVM 2D Speedup vs. scalar, hsize=128, class S

## VI. RELATED WORK

A memory-to-memory architecture is used for the Burroughs Scientific Processor (BSP) [15], [11]. The machine is optimized for the Fortran programming language, having the ISA composed of vector forms, which are very high level vector instructions with a large number of parameters. A single vector form is capable of expressing four operations performed on scalar, 1D or 2D arrays of arbitrary lengths. In order to store intermediate results, each arithmetic unit of the BSP includes a set of 10 registers, but they are not directly accessible by the programmer, being allocated by the vector forms. Our proposal also creates the premises for a high level instruction set. However, while BSP has a limited number of automatically managed registers which can be used for storing intermediate results, our approach can reuse data directly within the register file. This offers more control and flexibility to the programmer and to the compiler and potentially improves performance.

The Complex Streamed Instructions (CSI) approach doesn't use data registers at all [13]. CSI is another memory-to-memory architecture which allows the processing of two-dimensional data streams of arbitrary length. One of the main motivations behind CSI is to avoid having the section size as an architectural constraint. Our proposal allows us to arbitrarily choose the best section size for each workload by resizing the

vector registers, greatly reducing the disadvantages of a fixed section size. CSI has to rely on data caches to benefit from data locality. As also noted for the BSP, our approach can make use of the register file instead, reducing the need for high speed data caches.

The concept of Vector Register Windows (VRW) [17] consists of grouping consecutive vector registers to form register windows by interpreting the most significant bits of the address of a vector register as the window index. This offers a more flexible way of accessing long vectors or consecutive rows of matrices, as a register window is effectively a 2D vector register. The programmer can arbitrarily choose the number of consecutive registers which form a window, defining one dimension of a 2D register. However, contrary to our proposal, the second dimension is fixed to the Section Size, all the register windows must contain the same number of vector registers, and the total number of windows cannot exceed the number of vector registers. The latter severely limits the granularity to which the register file can be partitioned. These restrictions are not present in the Polymorphic Register File Architecture, giving the programmer a much higher degree of freedom when partitioning the register file. Therefore, the instructions can operate on matrices of different dimensions, reducing the overhead for reconfiguring the register windows.

Two dimensional register files have been used in other architectures, such as Matrix Oriented Multimedia (MOM), a matrix oriented ISA targeted at multimedia applications [5]. MOM also uses a two dimensional register file in order to exploit the available data level parallelism. The architecture supports 16 vector registers, each containing 16 64-bit elements. By using sub-word level parallelism, each MOM register can store a matrix containing at most 16x8 elements. The Polymorphic register file which we propose also uses sub-word level parallelism, but doesn't restrict the number or the size of the two dimensional registers, bearing additional flexibility.

Another architecture which also uses a two dimensional vector register file is Modified MMX (MMMX) [19]. This extension of MMX supports 8 96-bit wide multimedia registers and special load and store instructions which provide single-column access to the subwords of the registers. Our proposed Polymorphic Register File architecture does not constrain the matrix operations only to loads and stores and allows the definition of multi-column matrices of arbitrary sizes.

The Register Pointer Architecture (RPA) [18] focuses on providing additional storage to a scalar processor thus reducing the overhead associated with the updating of the index registers while minimizing the changes to the base instruction set. The architecture extends the baseline design with two extra register files: the Dereferencible Register File (DRF) and the Register Pointers (RP). The DRF increases the number of available registers to the processor. The RP provide indirect access to the DRF and are composed from three fields: a DRF index and two addresses (Begin and End) for wrap around circular addressing. The RP may be automatically incremented when used in an instruction. RPA is similar to our proposal as it

also facilitates the indirect accessing to a dedicated register file by using dedicated indirection registers. However, the parameters stored in the indirection registers are completely different given the distinct target application domains. While using RPA it would map to a scalar element, in our proposal, one indirection register maps to a matrix in the register file, being more suitable for vector processing by better expressing the available data level parallelism.

A VLIW processor which is able to use a variable number of registers depending on the workload is presented in [21]. In their approach, the number of available registers is adjusted by modifying the size of the physical register file using partial reconfiguration on the FPGA. However, the width of each individual register remains fixed. Our proposal considers that the total size of the physical register file is fixed, but the the number of registers as well as their shapes and dimensions can be set arbitrarily, offering a higher level view of the available storage and possibly reducing the number of instructions required to process the data. This in turn may improve performance as many overhead instructions can be eliminated.

Besides the improved performance and storage space utilization efficiency, the proposed Polymorphic RF provides easier programming interface compared to related works. By customizing the register dimensions and location during runtime, the programmer can potentially improve the RF utilization on one hand, but can also reduce the number of address arithmetic instructions at the other. The low overhead sectioning mechanism and the possibility to access the data in the register file in arbitrary shaped blocks can be automatically handled by the microarchitecture thus hiding the complexity of pointers and index manipulation. The productivity of a programmer writing vector code is further improved by defining a minimal instruction set extension with a small number of extra opcodes. This can be achieved as as the same binary instructions can be used regardless the dimensions and the data type of the registers. Such an approach holds both for normal vector instructions execution but also for selective data processing during conditional execution using bit vectors. We believe that such techniques and mechanisms can potentially reduce the time and effort required to transform a mathematical linear algebra formula into a high performance vector program.

## VII. CONCLUSIONS

We proposed a polymorphic register file architecture, which provides the system programmer with powerful means to organize the internal machine storage efficiently. Using cycle accurate simulations we have performed an evaluation of the proposed polymorphic register file in terms of performance acceleration. For the Floyd 64x64 benchmark, simulation results suggest a potential performance gain of up to 16 times compared to an idealized scalar processor and up to 3 times compared to the Cell SPU as well as a significant reduction in the number of executed instructions. For the sparse matrix vector multiplication kernel, simulation results suggest that by performing a two dimensional vectorization

we can achieve a speedup of up to 11 times compared to the scalar reference processor and up to 2 times compared to a one dimensional vector machine. In the future, we plan to analyze the performance of the proposed register file organization on a wider range of real-life applications. Furthermore, we shall investigate the most efficient microarchitectural implementations in terms of hardware complexity, performance and power.

## REFERENCES

- [1] David August and Jonathan Chang et al. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007.
- [2] D. Bailey, J. Barton, T. Lasinski, , H. Simon, and eds. The NAS Parallel Benchmarks. Technical Report Technical Report RNR-91-02, NASA Ames Research Center, Moffett Field, CA 94035, 1991.
- [3] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, page 51, 1986.
- [4] CellSim: Modular Simulator for Heterogeneous Multiprocessor Architectures. Online. <http://pcsstres.ac.upc.edu/cellsim/doku.php>.
- [5] Jesus Corbal, Roger Espasa, and Mateo Valero. MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications. In *Proceedings of the ACM/IEEE SC99 Conference*, pages 1–12, 1999.
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [7] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review*, 26(1):91–112, 1984.
- [8] Robert W. Floyd. Algorithm 97: Shortest Path. *Commun. ACM*, 5(6):345, 1962.
- [9] M.J. Flynn. Very High-speed Computing Systems. *Proceedings of the IEEE*, 54:1901–1909, December 1966.
- [10] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [11] G.T. Gray and R.Q. Smith. After the B5000: Burroughs Third-Generation Computers 1964 - 1980. *Annals of the History of Computing, IEEE*, 31(2):44–55, april-june 2009.
- [12] IBM. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*, 1.11 edition, May 2008.
- [13] B.H.H. Juurlink, D. Cheresiz, S. Vassiliadis, and H. A. G. Wijshoff. Implementation and Evaluation of the Complex Streamed Instruction Set. *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 73 – 82, 2001.
- [14] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.
- [15] D.J. Kuck and R.A. Stokes. The Burroughs Scientific Processor (BSP). *Computers, IEEE Transactions on*, C-31(5):363–376, may 1982.
- [16] Brian Moore, Andris Padegs, Ron Smith, and Werner Buchholz. Concepts of the System/370 Vector Architecture. In *ISCA '87*, pages 282 – 288, 1987.
- [17] D.K. Panda and K. Hwang. Reconfigurable Vector Register Windows for Fast Matrix Computation on the Orthogonal Multiprocessor. In *Application Specific Array Processors, 1990. Proceedings of the International Conference on*, pages 202 –213, 5-7 1990.
- [18] JongSoo Park, Sung-Boem Park, James D. Balfour, David Black-Schaffer, Christos Kozyrakis, and William J. Dally. Register Pointer Architecture for Efficient Embedded Processors. In *DATE '07: Proceedings of the conference on Design, Automation and Test in Europe*, pages 600–605, San Jose, CA, USA, 2007. EDA Consortium.
- [19] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors. In *Proceedings of the 2nd ACM Int. Conf. on Computing Frontiers*, pages 171–180, May 2005.
- [20] Jonathan R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.
- [21] S. Wong, F. Anjam, and M.F. Nadeem. Dynamically Reconfigurable Register File for a Softcore VLIW Processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2010)*, March 2010.