A Polymorphic Register File Architecture

Cătălin Ciobanu*, Georgi Kuzmanov*,
Alex Ramirez†,‡, Georgi Gaydadjiev*

*Delft University of Technology, Computer Engineering Laboratory, The Netherlands
†Polytechnic University of Catalonia, Computer Architecture Department, Spain
‡Barcelona Supercomputing Center - CNS, Spain

ABSTRACT

Previous vector architectures divided the available register file space in a fixed number of registers of equal sizes. We propose a novel register file organization which allows dynamic creation of a variable number of multidimensional registers of arbitrary sizes - the Polymorphic Register File. We have selected Floyd 64x64 as our benchmark. Simulation results suggest a speedup of up to 8X compared to an idealized Cell PPU scalar processor and a large reduction in the number of executed instructions. Preliminary results indicate that the proposed architecture outperforms the Cell SPU by around 50% without exceeding the 256KB storage size of the Local Store.

KEYWORDS: Vector processors, Vector register file, Polymorphism, Cell, DLP, Vector ISA

1 Introduction

In classic vector architectures such as the IBM/370 [Buc86], the number of data elements which can be stored in one vector register is a micro-architectural parameter - the *section size* of the machine. More recently, the Single Instruction Multiple Data extensions used fixed width vector registers (128-bit wide for Altivec [IBM08]) which can store multiple data elements. In both approaches, the number of the available registers is clearly defined in the architecture (16 vector registers for the IBM/370, 32 registers in the case of Altivec).

In this poster, we propose a novel register file organization - the Polymorphic Register File (PRF), which targets the efficient processing of multidimensional arrays of different sizes and dimensions. The total size of the register file amounts up to a fixed volume, while the actual number of dimensions and sizes of the vector registers is parameterizable during runtime, and multiple register sizes and dimensions can be used simultaneously. Currently, only 1D and 2D matrices are supported by the architecture, but it can be extended for any number of dimensions.

The main advantages of the Polymorphic Register File are:

- **Code compression** - achieved by reducing the number of overhead instructions required for address generation and looping;
- **Storage efficiency** - Polymorphism eliminates the need for padding of small matrices and allows the programmer to make full use of the available register storage space by supporting variable section sizes and vector dimensions;
- **Variable number of registers** - depending on the workload, the programmer has the freedom of dividing the available storage in an arbitrary number of registers of convenient sizes and dimensions;

[1]E-mail: {c.b.ciobanu, g.k.kuzmanov, g.n.gaydadjiev}@tudelft.nl, aramirez@ac.upc.edu, alex.ramirez@bsc.es

[2]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).

- **Improved performance** - increased performance by allowing vectorization on multiple directions and therefore increasing the average vector length.

The paper is organized as follows: in Section 2, we provide motivation for a Polymorphic Register File. In Section 3, we describe the architecture of the proposed Polymorphic Register File. In Section 4, we present a simple application and simulation data. Future work is presented in Section 5. Finally, the paper is concluded in Section 6.

2 Motivation

Let us assume the product computation of two matrices: A[2][4] and B[4][3] with the result stored in C[2][3]. We are considering a scalar processor using the Cell PPU PowerPC instruction set and a 2D polymorphic register file-augmented vector co-processor. We also assume that the dimensions of the matrices are small enough so that no sectioning instructions are required. In both cases, the data type used is 64-bit floating point.

By compiling the C code for the Cell PPU, the scalar PowerPC assembly code contains **41** instructions, and a total of **269** instructions are committed on our simulator. By connecting a vector co-processor to the PPU, the assembly code is reduced to **26** PowerPC and **11** vector instructions, excluding the synchronization overhead. Only these **37** instructions are committed as we can eliminate both the looping and most of the address generation overhead instructions. Figure 1 shows how the two dimensional registers can be configured to store A, B, and C in a hypothetical 2DPRF.

Based on this example, we can clearly identify several key motivating benefits from using a 2D polymorphic register file: **Static code compression** - the assembly code using the 2DPRF consists of 10.1% less instructions compared to the PPU; **Storage efficiency** - the registers were defined to contain exactly as many elements as required, eliminating storage waste; **Variable number of registers** - we defined three registers to store the operands; **Potential performance gain** - reduced number of committed instructions. This reduction of 7.27 times compared to the PowerPC further reduces Flynn's bottleneck [Fly66].

3 The Polymorphic Register File architecture

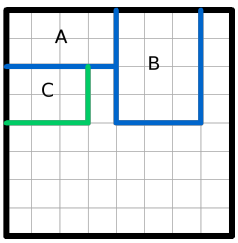

Figure 1: Defining the three example matrices in the Polymorphic Register File

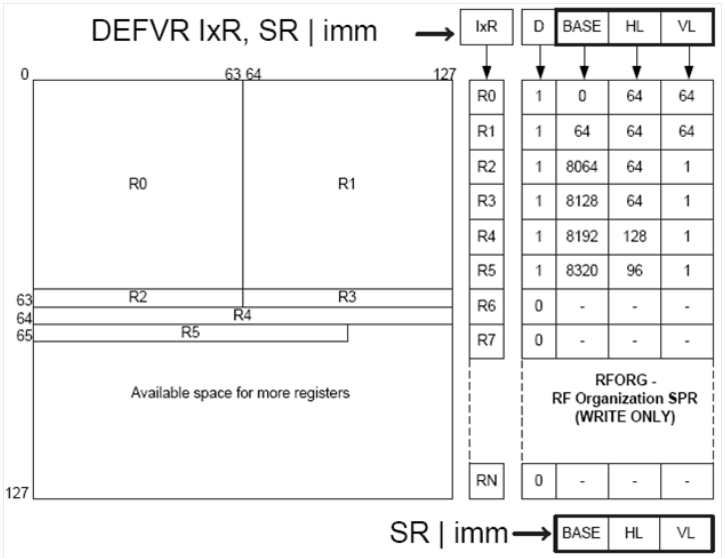

Figure 2: The Polymorphic Register File

Figure 2 illustrates an organization of our polymorphic register file, assuming that the size 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 only 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.

Both 1D and 2D register operations are supported simultaneously, using the same instructions. 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.

4 Simulation results - Floyd

Floyd's algorithm [Flo62] computes the shortest paths between all pairs of vertices in a weighted, directed graph. Given a graph $G = (V, E)$ with N nodes and an 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 implementation consists of three nested loops, and the basic operation is $d_{ij} = min(d_{ij}, d_{ik} + d_{kj})$. While a 1D vector processor can only process one line (N elements) of the cost matrix at a time , our proposed 2D Polymorphic Register File is able to process up to N^2 elements with one vector instruction.

We have implemented the Polymorphic Register File as part of a Scientific Vector Accelerator (SVA) in a cycle accurate simulator written in Unisim [ACG+07], an extension of SystemC. The processor module implements the instruction set of the PowerPC Processor Unit (PPU) in the Cell processor [KDH+05]. We assume each scalar instruction is executed in one cycle with perfect Instruction and Data caches having one cycle latency. The Vector Accelerator 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.

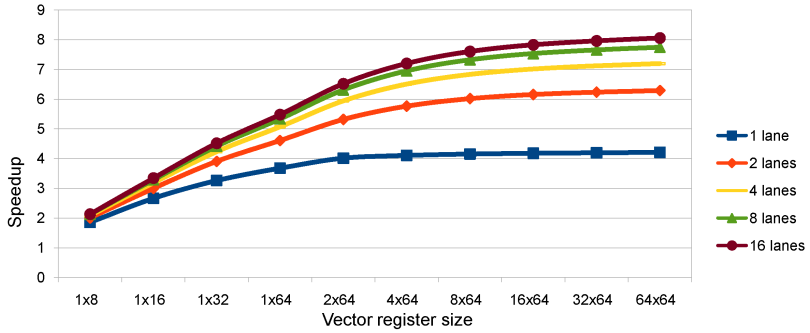

Figure 3: 2D vectorized Floyd 64x64 performance vs. the scalar PPU implementation

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. The bandwidth between the Local Store and the vector unit was set to 16 bytes, equal to the bandwidth used in the Cell processor. The execution model assumes that two fully pipelined

Table 1: Code compression rates for Floyd, 64 nodes

Reg. Size	1x8	1x16	1x32	1x64	2x64	4x64	8x64	16x64	32x64	64x64
Compression	4.3	8.6	17.2	34.4	68.8	136.9	271.4	532.7	1027.6	1918.9

arithmetic units, each having 5 stages, are available for each vector lane. The data type used for the cost matrix is double precision floating point (64-bit).

The instruction compression rates without taking into consideration the synchronization, initial parameter passing and outer loop overhead are presented in Table 1. The number of instructions committed by the vector accelerator is significantly lower compared to the same benchmark running on the PPU, with compression rates up to 1919 times.

Figure 3 presents the speedups obtained by using the 2D polymorphic register file compared to the PowerPC processor. Speedups up to 8X can be achieved when 16 vector lanes are used and the vector registers can store 64x64 elements. This provides additional scaling when comparing with the maximum speedup obtained by only using a 1D register - 5.4X in the case of the 1x64 register. We observe that 4 vector lanes are sufficient to sustain 89% of the peak performance.

5 Future Work

Preliminary benchmarking results against the Cell BE using a Playstation3(PS3) indicate that our proposed register file architecture is around 50% faster when running Floyd 64x64 with 16 vector lanes and 64-bit floating point data compared to one Cell SPU executing the 32-bit integer version of the same benchmark. 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 Cell SPE. If 4 vector lanes are used, our accelerator is around 40% faster than the SPU. The 32-bit integer PS3 SPU code runs Floyd 64x64 4.33 times faster compared to the 32-bit scalar PS3 PPU, without considering the DMA transfer time to the Local Store. The 32-bit integer version of Floyd running on the PS3 PPU is around 20% faster than the 64-bit floating point Floyd running on our idealized PPU simulator. Future work includes a detailed performance comparison to the Cell BE and PowerXCell 8i processors as well as evaluating the hardware complexity and power consumption of the Polymorphic Register File.

6 Conclusions

We have performed an initial evaluation of our proposed register file organization. Simulation results suggest a potential performance gain of up to 8 times compared to an idealized scalar processor as well as an important reduction in the number of executed instructions. Preliminary results indicate a performance improvement of around 50% compared to the Cell SPU.

References

[ACG+07] D. August, J. Chang, S. Girbal, D., Gracia-Perez, G. Mouchard, D. Penry, O Temam, and N. Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007.

[Buc86] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[Fly66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, December 1966.

[IBM08] IBM. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*, 1.11 edition, May 2008.

[KDH+05] 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.