Clustering Cores for Parallel Thread Execution

Muhamed F. Mudawar

Computer Engineering Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
mudawar@kfupm.edu.sa

Ayman A. Hroub

Computer Engineering Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
aymanh@kfupm.edu.sa

Abstract

In recent years, we have observed a strong trend towards using accelerators, such as GPUs, to speed up scientific applications. This results in a complex heterogeneous system in which traditional CPUs are used for the execution of sequential threads, while GPUs are used for accelerating parallel threads. Instead of following this trend, this paper introduces a new explicitly parallel instruction set architecture which can be used equally for sequential and parallel thread execution. It proposes the idea of clustering many cores as a new structure for parallel thread execution. The PAR cluster is intended to execute the portions of the programs that have intensive data-level parallelism that can be expressed explicitly as parallel blocks at the instruction level.

In this paper, we focus on describing the proposed PAR cluster hardware model and its performance. The PAR cluster is based on a newly defined PAR instruction set architecture. The PAR cluster executes the same parallel block for a specified number of threads.

The simulation results show that the PAR cluster has high throughput and high utilization of the hardware functional resources. In addition, they show that this architecture is scalable in terms of the number of number of cores and shareable data cache banks. The number of cores was scaled to 64 and the maximum achieved IPC on a single PAR cluster is 174.3 instructions per cycle.

Keywords: PAR cluster, many-core, multi-threading, parallel thread execution.

Acknowledgement: *The authors acknowledge the support provided by King Abdulaziz City for Science & Technology (KACST) through the Science and Technology Unit at KFUPM for funding project 08-ELE43-4 as part of the National Science, Technology and Innovation Plan.*

1. Introduction and Previous Work

A multithreaded processor is a processor that can handle different threads simultaneously. The first multithreaded processors appeared in 1970s and 1980s to solve the problem of remote memory access. From those days until now, many multithreading architectures have been proposed either for general or special purpose computing. Two general approaches were used for multithreading: the single chip multiprocessor which integrates two or more independent processors on a single chip and the multithreaded pipeline which is able to pursue two or more threads of control in parallel within a single processor [1].

El-Kharashi et al. [2] predicted that multithreaded processors will be the upcoming generation for multimedia chips because multimedia applications suffer from long latencies as a result of network contention, frequent memory references and limited communication bandwidth. Having multithreaded processors will tolerate these latencies by switching to another thread at each long latency operation. El-Kharashi mentioned some motivations of having multithreaded processors, such as hiding latencies, having dynamic task scheduling, increasing concurrency, improving multichip behavior, and alleviating the operating system overhead. They also listed the general hardware requirements for a multithreaded processor, which include: handling multiple contexts, hardware thread scheduler, sharing resources, advanced memory management, scalable memory protection, efficient communication and built-in synchronization.

Many ideas have been proposed to increase the performance of multithreaded processors. Zahran and Franklin [3] proposed a speculative multithreaded architecture with dynamic thread resizing at runtime. Threads are extracted from a sequential program by the

compiler or by hardware and they are speculatively executed in parallel.

Ungerer et al. surveyed and classified various multithreading techniques in research and commercial processors [1]. They classified the multithreading techniques as interleaved, blocked, and simultaneous multithreading.

In 2001, IBM introduced Power4-based systems in which two processor cores are integrated on a single chip. Recently, they introduced the Power7 as a next generation server processor [7].

In 2005, Sun Microsystems developed the Niagara processor [8] which is a multithreaded processor designed to provide high performance for commercial server applications. This kind of architecture helps in hiding the latency of memory access.

Lindholm et al. [9] describes the Tesla architecture that was introduced in 2006 in the GeForce 8800 GPU. Tesla architecture is based on a scalable processor array. GeForce 8800 GPU consists of 128 streaming-processor (SP) cores organized as 16 streaming multiprocessors (SMs). The 16 multithreaded processors are also organized in eight independent processing units called texture/processor clusters (TPCs). Tesla architecture is scalable and achieves high throughput for throughput applications which extensive data parallelism, intensive floating-point arithmetic, modest task parallelism and modest inter-thread synchronization.

The SIMT architecture has been introduced in SM which creates, manages and executes threads in groups of 32 parallel threads called warps. The threads in the same warp are of the same type and they start from the same address but during the execution they are free to branch independently. Tesla introduced the cooperative thread array (CTA) which is an array of threads that execute the same thread program and cooperates to compute a result.

More recently, Intel announced the Intel Many Integrated Core Architecture (Intel MIC Architecture), a general-purpose, many-core coprocessor that improves the programmability of co-processing devices by supporting a well-known shared-memory execution model based on the Intel architecture [11]. The Intel coprocessor consists of 32 general-purpose cores, based on a new Intel Pentium design that can execute 64-bit scalar as well as 512-bit vector instructions. Each core can execute four hardware threads with round-robin scheduling between instruction streams.

2. Motivation

The motivation of this research work is to define an explicitly parallel instruction set architecture, which can serve the development of future many-core architectures and simplify the parallel programming model. This paper defines a special **PAR** instruction that spawns an array of vector threads (called Vthreads) for the parallel execution of an instruction block. A *stop bit* marks the end of an instruction block. The stop bit simplifies the control of sequential and parallel instruction blocks and serves as a barrier for parallel thread execution. This research work also advocates the idea of hardware thread scheduling, in which threads are spawned, queued, and scheduled by the hardware with no support provided by the runtime system. This paper also proposes the PAR cluster to accelerate the execution of thread arrays and parallel instruction blocks. The PAR programming model and instruction set is described next.

3. The PAR Programming Model

The PAR programming model will be introduced through an example. Consider the parallel multiplication of vector A of length *n* by matrix B of size *n* by *n* doubles. The high-level algorithm uses the **par** keyword to describe the parallel block that includes a nested **for** loop. Each thread *i* computes one element of the result vector C.

```
VMM (int n, double A[n],B[n][n],C[n]){
  par (n) {
    int i = par_index();
    double sum = 0.0;
    for (k=0; k<n; k++) {
      sum += A[k] * B[k][i];
    }
    C[i] = sum;
  }
}
```

The **par** keyword spawns an *array* of *n* threads, as illustrated in Figure 1. Each thread has a unique index *i* that computes one element *C*[*i*]. The number of threads in the **par** block is *n*. The body of a **par** block corresponds to a scalar thread. Different threads can be scheduled to run sequentially on the same core or in parallel on multiple cores, according to implementation.

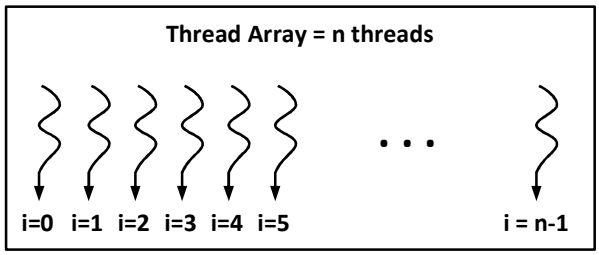

Figure 1: Spawning an array of *n* threads

3.1 Scheduling Threads

Scalar threads are flexible. They can run asynchronously at different speed and select different control paths. However, scalar threads intensify parallel computation and increase the scheduling overhead. To reduce this overhead, *vector threads* are used.

Instead of scheduling scalar threads, the **`par`** scheduler groups them into an array of *vector threads*, called Vthreads. Each Vthread has a unique index *i* and *length* V, as shown in Figure 2. In general, V is equal to a hardware defined *vector length* VL. The only exception is the last Vthread that has a length equal to the remainder V=(*n*%VL), if *n* is not divisible by VL. The constant VL is chosen always to be a power of 2 and can vary according to implementation. The number of Vthreads in a **PAR** block is equal to $\lceil n / \mathrm{VL} \rceil$.

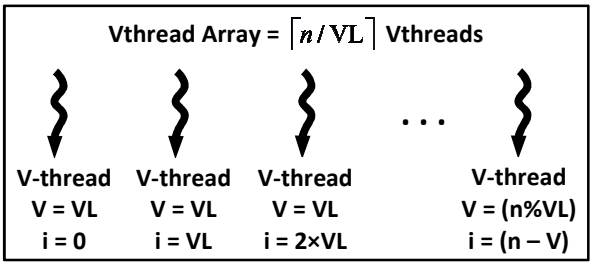

Figure 2: Grouping Threads into V-threads

3.2 The PAR instruction

The **PAR** instruction spawns an array of Vthreads for the parallel execution of an instruction block. The Vthreads execute the same parallel block that begins at label **`L1`** and terminates with a *stop bit* (**#** symbol) that marks the end of an instruction block. The **PAR** instruction uses a register to specify the number *n* of parallel threads. It computes the number of Vthreads as equal to $\lceil n / \mathrm{VL} \rceil$.

The VMM procedure is coded as shown below. Each Vthread computes VL elements of C starting at index *i*. The number of threads *n* is specified by register **`r1`**.

```
VMM: // spawn Vthreads
     par   r1, L1 #

L1:  // PAR Block: r0 = par index
     set   v1  = 0          // v1  = sum
     mov   r8  = r2         // r8  = &A
     sll   r9  = r0, 3      // r9  = i*8
     add   r10 = r3, r9     // &B[0][i]
     add   r11 = r4, r9     // &C[i]
     sll   r12 = r1, 3      // n*8 bytes
     loop  r1, L2           // loop n
     st8   [r11] = v1 #     // store sum

L2:  // LOOP Block
     ld8   r13 = [r8]       // load A[k]
     ld8   v2  = [r10]      // B[k][i]
     add   r8  = r8, 8      // &A[k]
     add   r10 = r10, r12   // &B[k][i]
     fma   v1  = r13, v2 # // sum
```

The **`loop`** instruction expands the loop block *n* times. The loop block starts at label **`L2`** and terminates with a stop bit. The loop iterates are sequential.

3.3 Index Register

Register **`r0`** is the *index register*. It is initialized by the hardware scheduler. Each Vthread receives a unique index in **`r0`**, as shown in Figure 2. The index value is a multiple of VL.

3.4 Inherited Registers

The PAR architecture defines registers **`r0`** to **`r15`** as inherited. These registers are *shared* by all Vthreads executing the same parallel instruction block, even when the V-threads do not need all of them. Inherited registers also facilitate the execution of Vthreads on multiple PAR clusters. These inherited registers are transferred and copied to the scalar registers of other PAR clusters, when Vthreads are scheduled to run on multiple clusters. However, no register copying is required when Vthreads are scheduled to run on the same PAR cluster.

3.5 Vector Registers

The **PAR** instruction also allocates vector registers **`v0`** to **`v15`**. Each vector register consists of VL elements that correspond to the VL parallel threads of a Vthread. Vector

registers are allocated dynamically when a Vthread is scheduled to run on a PAR cluster. They are freed when a Vthread terminates, when the stop bit of the parallel block is reached.

3.6 Scalar and Vector Instructions

A scalar instruction operates on inherited scalar registers only (**`r0`** to **`r15`**). It is executed once, regardless of VL. On the other hand, a vector instruction can operate on a mix of vector and scalar registers. It is equivalent to VL scalar instructions. For example, the first **`ld8`** instruction in the loop block is a scalar load that transfers the value of **`A[k]`** into register **`r13`**, while the second **`ld8`** instruction is a vector load that transfers a contiguous vector from memory into vector register **`v2`**. Scalar registers **`[r8]`** and **`[r10]`** specify the memory addresses.

Vector instructions can be used only inside a PAR block invoked by a **PAR** instruction or a nested block. On the other hand, sequential threads can execute only scalar instructions. The PAR architecture does not allow the execution of a vector instruction or the access to a vector register outside a PAR block.

3.7 The PAR Instruction Set Architecture

A newly explicitly parallel instruction set has been defined, called the PAR architecture. This architecture is centered on the concept of the **PAR** instruction. It features structured instruction blocks which can be sequential or parallel. Stop bits mark the end of instruction blocks. They are used to schedule sequential and parallel blocks. This architecture also features parallel vector threads that can be executed on a cluster of V-cores. It also features predication used for conditional execution of instructions. This architecture is described in more details in [12].

4. PAR Cluster Hardware Model

A PAR cluster consists of two types of cores, as depicted in Figure 3. Scalar cores (called S-cores) are optimized for the execution of sequential threads, while Vector cores (called V-cores) are optimized for the execution of parallel V-threads. An S-core is a traditional core with an instruction cache (I-cache), a data cache (D-cache), a scalar register file, and multiple function units. Instructions are issued out-of-order and complete in-order on an S-core to optimize sequential thread performance. Although two S-cores are shown in Figure 3, this number can vary according to implementation.

A program begins execution as a sequential thread on an S-core. The **PAR** instruction spawns parallel V-threads that are scheduled to run on V-cores. The PAR scheduler allocates vector registers to enable a V-thread to run in parallel on all V-cores belonging to a single cluster. The PAR scheduler can also send PAR packets to other clusters to schedule V-threads to run on multiple clusters. The PAR packet carries the ID of the parent thread, the number of V-threads with their starting index, the address

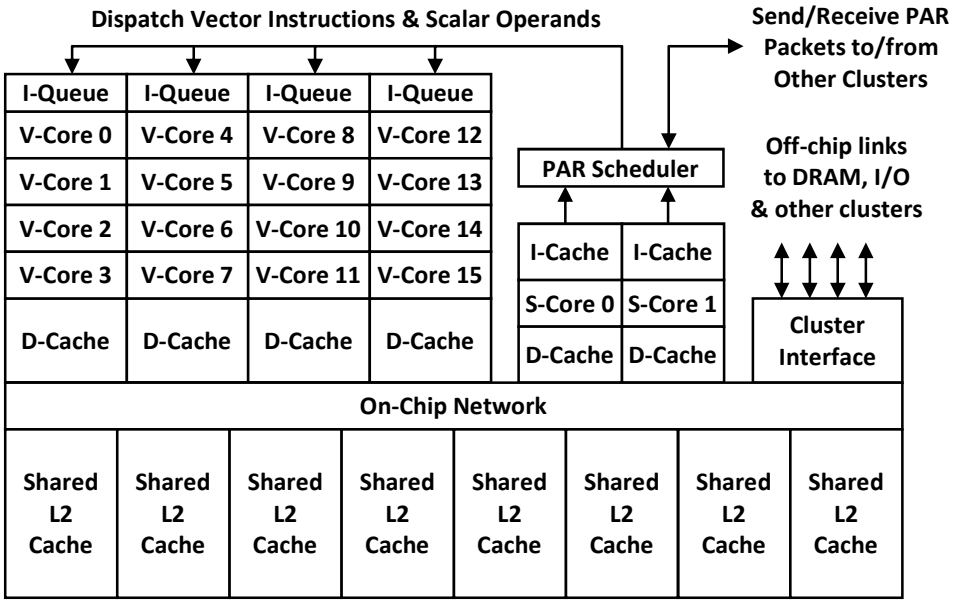

Figure 3: PAR Cluster

of the target parallel block, and the values of the inherited registers. This enables the execution of V-threads on many clusters. However, we only discuss here the execution of V-threads on a single cluster.

A V-thread runs in parallel on all the V-cores of a cluster. The parent thread that has issued the **PAR** instruction is running on an S-core. A V-thread executes a mix of scalar and vector instructions. The instruction unit on the S-core dispatches scalar instructions to execute on the S-core and vector instructions to execute in parallel on all V-cores. Vector instructions can operate on vector registers only or a mix of vector and scalar registers. If a vector instruction has a scalar operand then the scalar operand is carried with the dispatched vector instruction. Vector instructions are queued until they are issued for execution. Each vector instruction is executed on all V-cores. Each vector instruction also carries a unique hardware context ID to identify its vector registers allocated by its parent thread. Therefore, the V-cores are workers only. They do not fetch or control the instruction flow. They only react by executing the awaiting vector instructions in their queues. V-cores are also multithreaded. They can simultaneously execute vector instructions that have different parent IDs.

V-cores are grouped, such that each group shares an instruction queue and a D-Cache, as shown in Figure 3. All clustered S-cores and V-cores share a common L2 Cache. The shared L2 cache is split into banks. It is interconnected to the various data caches through on-chip network that provides parallel data access and coherence.

5. PAR Simulator

We developed a simulator in C++ to evaluate the PAR architecture. PARsim is a cycle accurate event-driven multithreaded simulator. It simulates the structure and functionality of the PAR cluster and evaluates its performance. PARsim receives two input files: (1) the configuration file which configures the PAR cluster, and (2) the benchmark file. PARsim generates a performance statistics report.

5.1 Benchmarks and Core Configuration

The following benchmarks are coded using the PAR ISA to evaluate the PAR cluster performance:

1. Dense Matrix-Matrix Multiplication (DMMM)
2. Jacobi Iterative Method (JIM)
3. Gauss-Seidel on a 2D grid (GS)
4. RGB to YIQ conversion
5. RGB to CMYK conversion
6. High Pass Grey-Scale Filter (HPF)
7. Scaled Vector Addition (SVA)

In addition, each V-core is configured to have two ALUs, an FPU, and a Load/Store unit. The instruction issue queue has 8 entries.

5.2 Simulation Results

Figure 4 shows how the IPC changes with respect to the number of simultaneous threads supported by a V-core. For all benchmarks, we noticed that the IPC increases with the number of threads. We also noticed that increasing the number of threads beyond four had little effect on the IPC value. Increasing the number of simultaneous threads will increase the work done by one vector instruction, which improves the utilization of the functional units. Increasing the number of simultaneous threads beyond four has little impact on the IPC because of the instruction mix in the benchmarks.

Four simultaneous threads per V-core give the highest IPC with the lowest hardware cost. For the selected benchmarks, the minimum IPC is 1.51 and the maximum is 2.75 for a 4-way V-core.

Adding more V-cores to a PAR cluster will replicate the work and reduce the execution time. Figure 5 shows the IPC for a PAR cluster. For 64 V-cores, the minimum IPC is 96.28 instructions/cycle while the maximum is 174.26 instructions/cycle. Adding more V-cores will not affect the complexity of the frontend I-Queues, but will increase the cost and complexity of the backend D-caches.

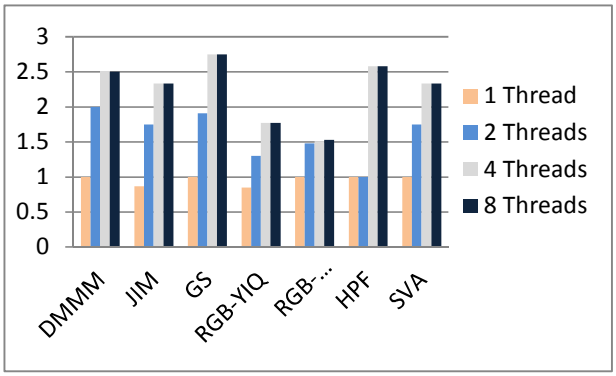

Figure 4: IPC for a Single V-Core

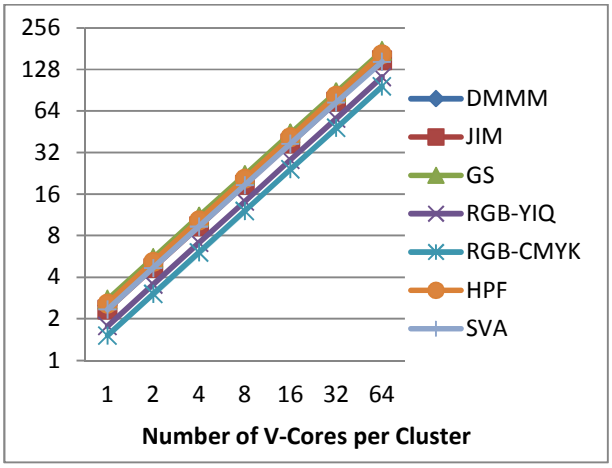

Figure 5: IPC on a PAR Cluster

6. Conclusion and Future Work

The technology trends and the application demand motivate the architects to design more productive chips that handle thread-parallelism at the instruction set level. In this research work, we proposed a scalable PAR cluster which supports S-cores for sequential thread execution and V-cores for parallel thread execution. V-cores are driven and shared by S-cores. We developed PARsim, which is a cycle-accurate simulator that implements the PAR cluster features and evaluates its performance. The simulation results show that the PAR cluster provides high throughput and high utilization of the functional resources.

In addition, the simulation results showed the scalability of the PAR cluster and its programming model. The same binary code can run without modification on different PAR clusters. Communication and synchronization goes through a shared L2 cache that provides data parallel access to the V-cores.

We are currently implementing a PAR cluster on a high-density Xilinx Virtex-6 FPGA. Future work will report the low-level implementation issues and the performance resulting from direct execution on a hardware prototype.

7. References

[1] T. Ungerer, B. Robič, and J. Šilc, "Multithreaded Processors," *The Computer Journal*, pp. 320-348, 2002.

[2] M. Watheq El-Kharashi, F. ElGuibaly, and K.F. Li, "Multithreaded Processors: the upcoming generation for multimedia chips," in *IEEE Symposium on Advances in Digital Filtering and Signal Processing*, 1988, pp. 111-115.

[3] M. Zahran and M. Franklin, "Dynamic thread resizing for speculative multithreaded processors," in *21st International Conference on Computer Design*, 2003, pp. 313- 318.

[4] Il Park, B. Falsafi, and T. Vijaykumar, "Implicitly-multithreaded processors," in *30th International Symposium on Computer Architecture*, 2003, pp. 39-50.

[5] S. Wallace, B. Calder, and D.M. Tullsen, "Threaded multiple path execution," in *The 25th Annual International Symposium on Computer Architecture*, 1998, pp. 238-249.

[6] H. Akkary and M.A. Driscoll, "A dynamic multithreading processor," in *31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998, pp. 226-236.

[7] R. Kalla, B. Sinharoy, W. Starke, M. Floyd, "Power7: IBM's Next-Generation Server Processor", *IEEE Micro*, pages 7-15, March/April 2010.

[8] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," *IEEE computer Society*, pp. 21- 29, 2005.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Computer Society*, pp. 39-55, 2008.

[10] F. Latorre, J. González, and A. González, "Efficient resources assignment schemes for clustered multithreaded processors," in *IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1-12.

[11] Intel, "Introducing Intel Many Integrated Core Architecture', press release, 2011, www.intel.com/ technology/architecture-silicon/mic/index.htm.

[12] M. Mudawar, "The PAR Architecture", Technical Report, NSTIP Project 08-ELE43-4, March 2012, Computer Engineering department, KFUPM.