

Programming the EMU Architecture: Algorithm Design Considerations for Migratory-threads-based Systems

Mehmet E Belviranli
Oak Ridge National Laboratory
Oak Ridge, Tennessee
belviranlime@ornl.gov

Seyong Lee
Oak Ridge National Laboratory
Oak Ridge, Tennessee
lees2@ornl.gov

Jeffrey S Vetter
Oak Ridge National Laboratory
Oak Ridge, Tennessee
vetter@ornl.gov

ABSTRACT

The decades-old memory bottleneck problem for data-intensive applications is getting worse as the processor core counts continue to increase. Workloads with sparse memory access characteristics only achieve a fraction of a system’s total memory bandwidth. The EMU architecture provides a radical approach to the issue by migrating the computational threads to the location where the data resides.

In EMU architecture, data distribution and thread creation strategies play a crucial role in achieving optimal performance in the EMU platform. In this work, we identify several design considerations that need to be taken care of while developing applications for the new architecture and we evaluate their performance effects on the EMU-chick hardware.

1 INTRODUCTION

Applications that fall into ‘data-intensive’ category, such as sparse matrix-based BLAS operations and graph analytics, only managed to achieve a tiny fraction of the R_{max} (i.e., maximal achieved performance). These types of workloads usually exhibit ‘weak-locality’ where frequent irregular accesses are scattered across a large memory range; therefore, their compute efficiencies are majorly limited by data movement capabilities of the underlying architectures. As the computation scales over a slower fabric, the dramatic increase in latency forces the need to further focus on ‘data-movement’ and reduce memory-imposed transfer and communication bottlenecks.

EMU[2] is a new architecture that attempts to address the weak-locality problem in data-intensive applications by migrating the threads to location where the data resides. The memory is accessed via a global address space, and since the data does not move, the need for cache coherency is eliminated. The programming model is based on Cilk [1], and thread migration is automatically performed by the device runtime as the memory accesses occur.

In this study, we discuss the architectural considerations that need to be taken into account while designing applications for EMU systems. We first enumerate, discuss, and compare several programming considerations for the EMU architecture. Then, we present an exploratory performance analysis on the effects of these considerations on the EMU-chick hardware.

2 EMU ARCHITECTURE

EMU is a novel architecture that provides scalable access to a common partitioned global address space (PGAS) through a simple programming interface. The hardware is hierarchically organized as nodes, nodelets, and gossamer cores from top to bottom, and each nodelet owns a partition of the globally addressable memory. Different from existing PGAS and active messaging based solutions, a thread in EMU migrates to the memory location of the operand that the current instruction operates on. Further details on the EMU platform can be found in [2].

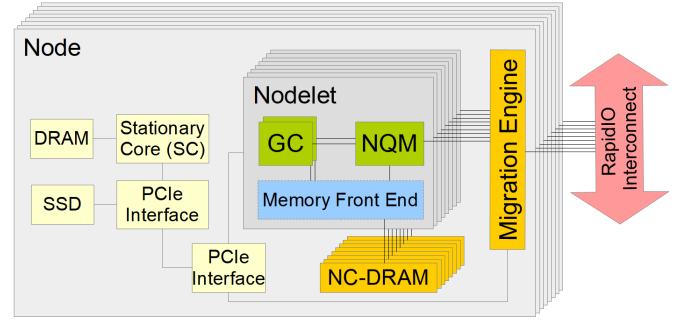


Figure 1: Emu-chick hardware overview.

The specific hardware discussed in this study is the 8-node EMU-chick system [3], and it is implemented using one FPGA per node in a single chassis. The architecture, on the other hand, is designed to scale up to hundreds of nodes that can fit inside a single rack. Figure 1 shows an overview of the components that the current hardware is composed of, and the Table 1 gives the specifications of the EMU-chick product.

Table 1: EMU-Chick hardware specifications

EMU-Chick Specifications	
Nodes	8 total
Nodelets	8 per node
Memory	64GB per nodelet
Compute cores	2 Gossamer cores (GC) per nodelet
Service cores	1 Stationary core (SC) per node
System Interconnect	4-lane Serial RapidIO 2.3 @ 6.25 Gbit

Each node incorporates eight nodelets, an array of DRAMs, a migration engine, PCI-Express interfaces, and a stationary processor (SP), accompanied with an SSD. A nodelet contains two Gosamer cores (GC), each of which supports 64 concurrent in-order, single-issue hardware threads. Each node has a 64-byte channel DRAM, divided into eight 8-byte narrow-channel-DRAMs (NC-DRAM).

2.1 Programming Model

EMU supports a PGAS model, and the memory is accessible via conventional pointer addressing. The parallelism is expressed via Cilk programming model [1], and the current LLVM-based EMU compiler supports three Cilk keywords: `cilk_spawn`, `cilk_sync`, and `cilk_for`.

`cilk_spawn` is used to create a new thread via a non-blocking call. The context is duplicated in full so that the children can further spawn threads independently. The thread migration occurs without programmer intervention.

3 DEVELOPING ALGORITHMS FOR EMU

In this section, we raise several design considerations unique to the EMU architecture and discuss their implications on applications

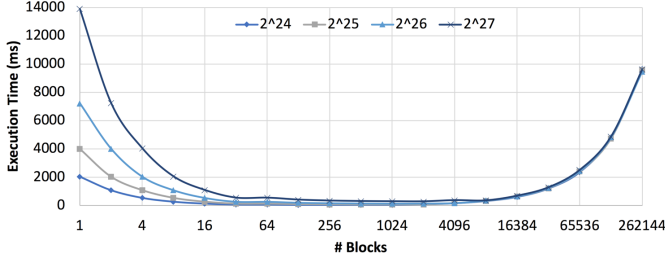


Figure 2: Execution time for scalar vector add for different data sizes and block sizes.

and we demonstrate the resulting performance of these designs using scalar vector-add on the EMU-chick hardware.

3.1 Memory allocation patterns

The way that the buffers are allocated in the memory has a direct implication on the number of thread migrations, hence the performance, for EMU applications. Because the computation is always done on the nodelet where the operands are located, memory should be distributed across the global address space while also keeping the indices belonging to different data ranges close enough so that a thread can execute an instruction with a minimal number of hops.

EMU provides three types of memory allocation interfaces: `mw_localmalloc` allocates a single chunk of memory on the nodelet that is local to the pointer given in the parameters. `mw_mallocldlong` allocates an array, where each element (i.e., long) is striped across all nodelets. `mw_malloc2d` lets the programmer specify the total number of blocks and the size of each block, and returns a double pointer.

Considering a 2D allocation pattern, the size of each block and the total number of blocks strongly co-relate to the number of thread migrations. The overhead will get larger as the number of total blocks increases. On the other hand, having larger blocks will limit the total memory bandwidth and the computation power. Figure 2 depicts the outcome of this trade-off relation for a scalar vector addition run on all 8 nodes and 64 nodelets.

3.2 Thread spawning strategies

Another factor affecting performance is determining how the child threads should be spawned. For example, if a parent needs to spawn a parallel thread for every iteration of a `for` loop, there are two options. In *flat spawn*, the parent thread sequentially iterates over the `for` loop and spawns children. The total spawning and parent migration overhead can reach up to $O(n)$ in this scenario. In *tree spawn*, the process of creating children in EMU can be parallelized

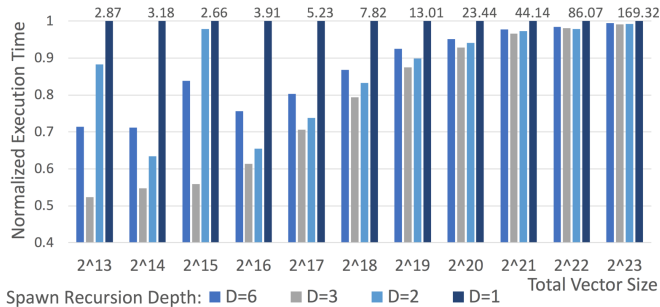


Figure 3: Effects of spawn recursion depth on performance for varying vector lengths and fixed number of blocks.

by using a recursive, hierarchal spawn. Tree spawn reduces the overhead complexity to $O(\log n)$.

Figure 3 depicts the results of an experiment that shows the effects of using different depths of recursive spawns for scalar vector-add on varying input sizes. In all cases, a non-flat spawn tree achieves better performance, since the overhead of serial spawn takes considerably longer.

3.3 Parallelism granularity

Each nodelet in EMU-chick can support up to 256 live threads and also can hold context information of up to 500 threads, depending on their total resource usage. In current HW implementation, threads that exceed this level will fail to spawn, and the issuing parent thread will be forced to execute them as a regular function call.

In order to control the number of threads spawned on each nodelet, we divide each block into further *chunks*. We spawn a thread for each chunk and within the chunk, we serially iterate over the indices so that the total number of threads are limited.

Figure 4 shows the effects of using different number of threads per nodelet, while the total number of blocks and block sizes are fixed for a given input size. The results show that as the number of threads per nodelet gets higher, the execution times increase dramatically, since thread contexts cannot be stored on the nodelet and every newly spawned thread is serially executed.

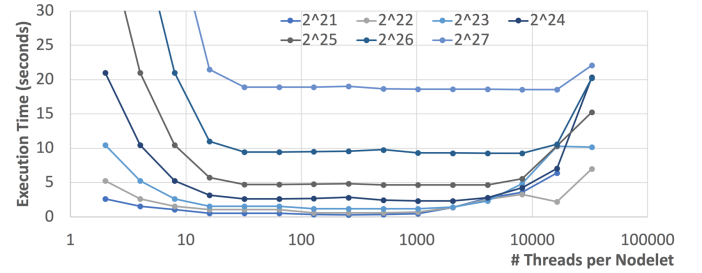


Figure 4: Effects of using different number of threads per nodelet, with fixed block size and counts.

4 CONCLUSION

EMU presents an unorthodox approach to solve increasingly worsening memory bottleneck problem in high performance computing. Threads are migrated into the location where the memory resides, and EMU achieves this without programmer intervention.

Our work shows that there are several unique architectural considerations that need to be addressed while developing for the EMU platform. While the programming interface mainly relies on Cilk, proper usage of intrinsics are required for performance effective algorithm design.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

REFERENCES

- [1] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. *Cilk: An efficient multithreaded runtime system*. Vol. 30. ACM.
- [2] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms (IA3 '16)*. IEEE Press, Piscataway, NJ, USA, 2–9. <https://doi.org/10.1109/IA3.2016.7>
- [3] EMU Technology. [n. d.]. Emu Chick Specifications. <http://www.emutechnology.com/products/emu-chick-specifications/>