PVTOL: Designing Portability, Productivity and Performance for Multicore Architectures

Hahn Kim, Nadya Bliss, Jim Daly, Karen Eng, Jeremiah Gale, James Geraci, Ryan Haney, Jeremy Kepner,
Sanjeev Mohindra, Sharon Sacco, Edward Rutledge
{hgk, nt, jdaly, keng, jgale, jrgeraci, haney, kepner, smohindra, ssacco, rutledge}@ll.mit.edu
MIT Lincoln Laboratory, 244 Wood St., Lexington, MA 02420-9108

1 Introduction

Modern DoD sensors continue to increase in fidelity and sampling rates, resulting in increasingly larger data sets. Form factor requirements remain the same or even shrink, e.g. UAV's, while real-time processing continues to be a requirement.

These factors lead to increasingly stringent requirements for real-time signal processing applications for these sensors. Data processing systems must increase processing power with limited power (e.g. 10's to 100's of Watts) and space (e.g. several cubic feet). Multicore architectures help hardware designers meet the required computational performance within these constraints. Multicore architectures, however, add complexity to the programming model that is unfamiliar to most programmers, e.g. parallel computation and explicit management of the memory hierarchy. This drives the need for technologies that hide this complexity, allowing programmers to focus on algorithms, not architectures.

The Parallel Vector Tile Optimizing Library (PVTOL) is a high-performance C++ real-time signal processing middleware library. PVTOL provides high productivity via a partitioned global address space (PGAS) programming model and is portable across a range of traditional and multicore architectures [1]. PVTOL's architecture has several layers, shown in Figure 1. This abstract describes API constructs that support task parallelism, data parallelism and computational functors.

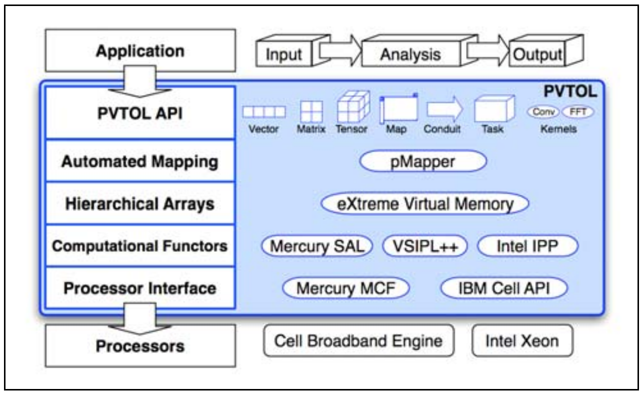

Figure 1: PVTOL Architecture

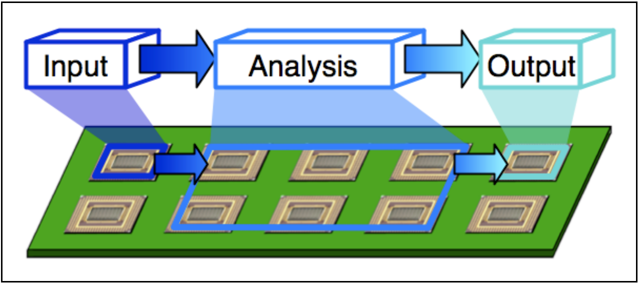

Figure 2: Tasks and Conduits

2 Tasks & Conduits

PVTOL supports task parallelism, i.e. assigning different computational tasks to different processors, using the Task and Conduit classes. *Tasks* encapsulate single-program multiple data (SPMD) code that executes on one or more processors. Processors can be either traditional or multicore processors. *Conduits* send data between tasks. Conduits hide the details of interconnect technologies, e.g. Gigabit Ethernet, PCI, etc., providing portability.

Figure 2 shows a basic signal processing pipeline. An input task captures data from a source, e.g. sensor. An analysis task performs computation. An output task sends the results elsewhere, e.g. a network. The arrows indicate conduits that connect the tasks.

3 Hierarchical Maps & Arrays

PVTOL's predecessors, PVL and VSIPL++, use maps to concisely describe how to allocate parallel arrays across multiple processors, enabling data parallelism. Multicore processors complicate matters by exposing the underlying processor and memory hierarchies.

A *processor hierarchy* contains a main processor and one or more co-processors that depend on the main processor for program control. For example, IBM's Cell Broadband Engine contains 9 cores: 1 PowerPC Processing Element (PPE) and 8 Synergistic Processing Elements (SPE). Applications start on the PPE, then spawn threads onto the SPE's. This hierarchical model supports other co-processor architectures, e.g. GPU's and FPGA's.

Each level in the processor hierarchy may have its own memory. Unlike caches, levels in the *memory hierarchy* may be disjoint, requiring explicit control over data movement through the hierarchy. On the Cell, the PPE first loads data into main memory, then SPE's transfer data from main memory into local store via DMA's. Table 1 contains example processor and memory hierarchies.

Figure 3 shows an example hierarchical array on a cluster of Cells. *Hierarchical maps* allocate *hierarchical arrays* across the processor and memory hierarchies. A hierarchical map for the Cell has three maps. The first two maps, cellMap and speMap, distribute the array across the Cell processor hierarchy. cellMap distributes an array across multiple Cell processors; speMap distributes data on each Cell across SPE's. Since SPE's have 256 KB of memory, subarrays are often broken into smaller blocks. The third map, speLsMap, distributes data across the Cell memory hierarchy by partitioning SPE data into blocks.

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government

Table 1: Example processor hierarchies

Architecture	Main Processor / Memory	Co-Processor / Memory
IBM Cell	PPE / XDR RAM	SPE / Local store
NVIDIA	Any x86 CPU / RAM	Tesla C870 GPU / GDDR3 RAM
Cray XR1	AMD Opteron / DDR SDRAM	Xilinx Virtex-4 / DDR SDRAM

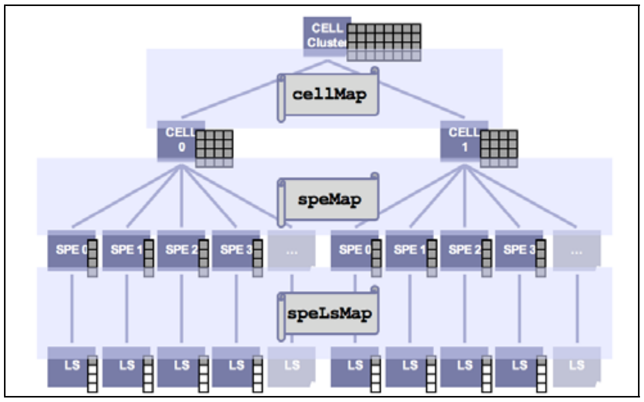

Figure 3: An example of a hierarchical array on two Cells.

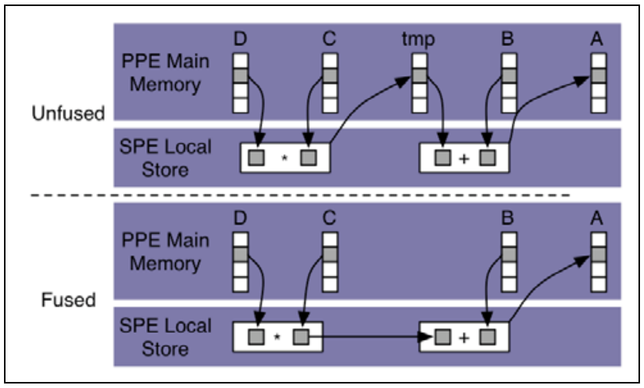

Figure 4: Unfused and fused versions of A=B+C*D on the Cell

4 Functors

A *functor* is an object that is called as if it were a function. In C++, functors define the () operator so that invoking a functor is syntactically identical to calling a function. A benefit of functors is the ability to store state, e.g. FFT twiddle factors or memory for intermediate values computed by SVD.

Functors can be written using the PVTOL API. This provides an easy way for library developers to add new functors to the library and for application developers to write custom functors for their application. Functors use the hierarchical array's get/put API to move data blocks across the memory hierarchy. Once loaded, data blocks are processed using optimized kernels, which range from assembly code to optimized vendor libraries, e.g. Mercury's Scientific Algorithm Library for the Cell and Intel's Integrated Performance Primitives.

Functors can improve performance by fusing data-parallel functors in an expression. Consider the expression A = B + C * D, which operates on hierarchical arrays. SPE's process one block at a time. A naïve implementation would process the * for all blocks in C and D, then the +, requiring a temporary variable to store the result of the *, adding DMA transfers between local store and main memory. A more efficient implementation will *fuse* the operations so that the entire expression is applied to a set of blocks across all arrays. Figure 4 shows the memory transfers required for unfused and fused versions of this expression. PVTOL will automatically fuse operations at runtime.

5 Results

We are currently working on a project at the Lincoln Laboratory that is building a real-time image processing application on the Cell. The first implementation directly used Mercury's MultiCore Framework (MCF). Figure 5 compares performance and programmer effort of three prototype implementations of a key kernel, the image projective transform, using ANSI C, MCF and PVTOL. The PVTOL version achieves nearly the same level of performance as the MCF implementation while providing a similar level of effort as ANSI C.

Our preliminary design for the application shows that PVTOL reduces the overall software lines of codes (SLOCs) from ~1600 to ~600. We are actively building PVTOL for this project and will present updated performance and productivity results.

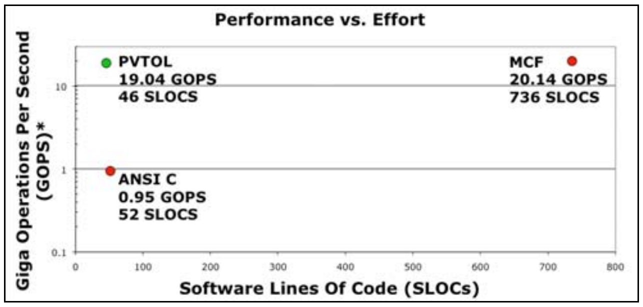

Figure 5: GOPS vs. SLOCs for the projective transform

6 Summary

PVTOL contains a number of programming constructs that hide the complexity of multicore architectures. Task and Conduits allow pipelining and round-robining of data. Hierarchical maps concisely describe how to allocate hierarchical arrays across processor and memory hierarchies and provide a simple API for moving data across the hierarchies. New functors can be easily developed using the PVTOL API and can be fused for more efficient computation.

We are actively building PVTOL for Intel and Cell architectures. We intend to add more functors and will look at supporting more processing architectures, including other multicore architectures, GPU's and FPGA's.

References

[1] H.Kim, N. Bliss, R. Haney, J. Kepner, M. Marzilli, S. Mohindra, S. Sacco, G. Schrader, E. Rutledge. "PVTOL: A High-Level Signal Processing Library for Multicore Proessors." *High Performance Embedded Computing Workshop 2007.* Lexington, MA. September 2007.