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, Eddie Rutledge

HPEC 2008

25 September 2008
Outline

• Background
  – Motivation
  – Multicore Processors
  – Programming Challenges

• Tasks & Conduits

• Maps & Arrays

• Results

• Summary
SWaP* for Real-Time Embedded Systems

- Modern DoD sensors continue to increase in fidelity and sampling rates
- Real-time processing will always be a requirement

Modern sensor platforms impose tight SWaP requirements on real-time embedded systems

* SWaP = Size, Weight and Power
Embedded Processor Evolution

- 20 years of exponential growth in FLOPS / W
- Must switch architectures every ~5 years
- Current high performance architectures are multicore

Multicore processors help achieve performance requirements within tight SWaP constraints

- Multicore processor
- 1 PowerPC core
- 8 SIMD cores
Parallel Vector Tile Optimizing Library

- PVTOL is a portable and scalable middleware library for multicore processors
- Enables unique software development process for real-time signal processing applications

Make parallel programming as easy as serial programming
Tasks & Conduits
Concurrency and data movement

Maps & Arrays
Distribute data across processor and memory hierarchies

Functors
Abstract computational kernels into objects

Portability: Runs on a range of architectures
Performance: Achieves high performance
Productivity: Minimizes effort at user level
Outline

• Background

• Tasks & Conduits

• Maps & Arrays

• Results

• Summary
Multicore Programming Challenges

Inside the Box

- Desktop
- Embedded Board

Threads
- Pthreads
- OpenMP

Shared memory
- Pointer passing
- Mutexes, condition variables

Outside the Box

- Cluster
- Embedded Multicomputer

Processes
- MPI (MPICH, Open MPI, etc.)
- Mercury PAS

Distributed memory
- Message passing

PVTOL provides consistent semantics for both multicore and cluster computing
Tasks & Conduits

- Tasks provide concurrency
  - Collection of 1+ threads in 1+ processes
  - Tasks are SPMD, i.e. each thread runs task code

- Task Maps specify locations of Tasks

- Conduits move data
  - Safely move data
  - Multibuffering
  - Synchronization

<table>
<thead>
<tr>
<th>DIT</th>
<th>Read data from source (1 thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAT</td>
<td>Process data (4 threads)</td>
</tr>
<tr>
<td>DOT</td>
<td>Output results (1 thread)</td>
</tr>
<tr>
<td>Conduits</td>
<td>Connect DIT to DAT and DAT to DOT</td>
</tr>
</tbody>
</table>

* DIT – Data Input Task, DAT – Data Analysis Task, DOT – Data Output Task
int main(int argc, char** argv) {
    // Create maps (omitted for brevity)
    ...

    // Create the tasks
    Task<Dit> dit("Data Input Task", ditMap);
    Task<Dat> dat("Data Analysis Task", datMap);
    Task<Dot> dot("Data Output Task", dotMap);

    // Create the conduits
    Conduit<Matrix<double>> ab("A to B Conduit");
    Conduit<Matrix<double>> bc("B to C Conduit");

    // Make the connections
    dit.init(ab.getWriter());
    dat.init(ab.getReader(), bc.getWriter());
    dot.init(bc.getReader());

    // Complete the connections
    ab.setupComplete(); bc.setupComplete();

    // Launch the tasks
    dit.run(); dat.run(); dot.run();

    // Wait for tasks to complete
    dit.waitTillDone();
    dat.waitTillDone();
    dot.waitTillDone();
}
Pipeline Example

Data Analysis Task (DAT)

class Dat
{
private:
    Conduit<Matrix <double> >::Reader m_Reader;
    Conduit<Matrix <double> >::Writer m_Writer;

public:
    void init(Conduit<Matrix <double> >::Reader& reader, 
              Conduit<Matrix <double> >::Writer& writer)
    {
        // Get data reader for the conduit
        reader.setup(tr1::Array<int, 2>(ROWS, COLS));
        m_Reader = reader;

        // Get data writer for the conduit
        writer.setup(tr1::Array<int, 2>(ROWS, COLS));
        m_Writer = writer;
    }

    void run()
    {
        Matrix <double>& B = m_Reader.getData();
        Matrix <double>& A = m_Writer.getData();
        A = B;
        m_reader.releaseData();
        m_writer.releaseData();
    }
};

Tasks read and write data using Reader and Writer interfaces to Conduits

Readers and Writer provide handles to data buffers

class Dat
{ private:
    Conduit<Matrix <double> >::Reader m_Reader;
    Conduit<Matrix <double> >::Writer m_Writer;

public:
    void init(Conduit<Matrix <double> >::Reader& reader, 
              Conduit<Matrix <double> >::Writer& writer)
    {
        // Get data reader for the conduit
        reader.setup(tr1::Array<int, 2>(ROWS, COLS));
        m_Reader = reader;

        // Get data writer for the conduit
        writer.setup(tr1::Array<int, 2>(ROWS, COLS));
        m_Writer = writer;
    }

    void run()
    {
        Matrix <double>& B = m_Reader.getData();
        Matrix <double>& A = m_Writer.getData();
        A = B;
        m_reader.releaseData();
        m_writer.releaseData();
    }
};
Outline

• Background

• Tasks & Conduits

• Maps & Arrays
  – Hierarchy
  – Functors

• Results

• Summary
Map-Based Programming

- A map is an assignment of blocks of data to processing elements
- Maps have been demonstrated in several technologies

<table>
<thead>
<tr>
<th>Technology</th>
<th>Organization</th>
<th>Language</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Vector Library</td>
<td>MIT-LL*</td>
<td>C++</td>
<td>2000</td>
</tr>
<tr>
<td>pMatlab</td>
<td>MIT-LL</td>
<td>MATLAB</td>
<td>2003</td>
</tr>
<tr>
<td>VSIPL++</td>
<td>HPEC-SI†</td>
<td>C++</td>
<td>2006</td>
</tr>
</tbody>
</table>

Grid specification together with processor list describe where data are distributed.

Distribution specification describes how data are distributed.
PVTOL Machine Model

• Processor Hierarchy
  - Processor:
    Scheduled by OS
  - Co-processor:
    Dependent on processor for program control

• Memory Hierarchy
  - Each level in the processor hierarchy can have its own memory

PVTOL extends maps to support hierarchy
PVTOL Machine Model

- **Processor Hierarchy**
  - Processor: Scheduled by OS
  - Co-processor: Dependent on processor for program control

- **Memory Hierarchy**
  - Each level in the processor hierarchy can have its own memory

Semantics are the same across different architectures
Hierarchical Maps and Arrays

- PVTOL provides hierarchical maps and arrays
- Hierarchical maps concisely describe data distribution at each level
- Hierarchical arrays hide details of the processor and memory hierarchy

Program Flow
1. Define a Block
   - Data type, index layout (e.g. row-major)
2. Define a Map for each level in the hierarchy
   - Grid, data distribution, processor list
3. Define an Array for the Block
4. Parallelize the Array with the Hierarchical Map (optional)
5. Process the Array
int main(int argc, char *argv[]) {
    PvtolProgram pvtol(argc, argv);

    // Allocate the array
    typedef Dense<2, int> BlockType;
    typedef Matrix<int, BlockType> MatType;
    MatType matrix(4, 8);
}
int main(int argc, char *argv[]) 
{
    PvtolProgram pvtol(argc, argv);

    // Distribute columns across 2 Cells
    Grid cellGrid(1, 2);
    DataDistDescription cellDist(BlockDist(0), BlockDist(0));
    RankList cellProcs(2);
    RuntimeMap cellMap(cellProcs, cellGrid, cellDist);

    // Allocate the array
    typedef Dense<2, int> BlockType;
    typedef Matrix<int, BlockType, RuntimeMap> MatType;
    MatType matrix(4, 8, cellMap);
}
int main(int argc, char *argv[]) {
    PvtolProgram pvtol(argc, argv);

    // Distribute into 1x1 blocks
    unsigned int speLsBlockDims[2] = {1, 2};
    TemporalBlockingInfo speLsBlock(2, speLsBlockDims);
    TemporalMap speLsMap(speLsBlock);

    // Distribute columns across 2 SPEs
    Grid speGrid(1, 2);
    DataDistDescription speDist(BlockDist(0), BlockDist(0));
    RankList speProcs(2);
    RuntimeMap speMap(speProcs, speGrid, speDist, speLsMap);

    // Distribute columns across 2 Cells
    vector<RuntimeMap *> vectSpeMaps(1);
    vectSpeMaps.push_back(&speMap);
    Grid cellGrid(1, 2);
    DataDistDescription cellDist(BlockDist(0), BlockDist(0));
    RankList cellProcs(2);
    RuntimeMap cellMap(cellProcs, cellGrid, cellDist, vectSpeMaps);

    // Allocate the array
    typedef Dense<2, int> BlockType;
    typedef Matrix<int, BlockType, RuntimeMap> MatType;
    MatType matrix(4, 8, cellMap);
}
Functor Fusion

• Expressions contain multiple operations
  – E.g. $A = B + C \cdot * D$

• Functors encapsulate computation in objects

• Fusing functors improves performance by removing need for temporary variables

Let $X_i$ be block $i$ in array $X$

**Unfused**

Perform $\text{tmp} = C \cdot * D$ for all blocks:
1. Load $D_i$ into SPE local store
2. Load $C_i$ into SPE local store
3. Perform $\text{tmp}_i = C_i \cdot * D_i$
4. Store $\text{tmp}_i$ in main memory

Perform $A_i = \text{tmp} + B_i$ for all blocks:
5. Load $\text{tmp}_i$ into SPE local store
6. Load $B_i$ into SPE local store
7. Perform $A_i = \text{tmp}_i + B_i$
8. Store $A_i$ in main memory

**Fused**

Perform $A = B + C \cdot * D$ for all blocks:
1. Load $D_i$ into SPE local store
2. Load $C_i$ into SPE local store
3. Perform $\text{tmp}_i = C_i \cdot * D_i$
4. Load $B_i$ into SPE local store
5. Perform $A_i = \text{tmp}_i + B_i$
6. Store $A_i$ in main memory

\[\cdot *\] = elementwise multiplication
Outline

• Background

• Tasks & Conduits

• Maps & Arrays

• Results

• Summary
Persistent Surveillance
Canonical Front End Processing

Processing Requirements ~300 Gflops

Stabilization/Registration (Optic Flow)
~600 ops/pixel (8 iterations) x 10% = 120 Gflops

Projective Transform
~40 ops/pixel = 80 Gflops

Detection
~50 ops/pixel = 100 Gflops

Signal and image processing turn sensor data into viewable images

Logical Block Diagram

4U Mercury Server
• 2 x AMD CPU motherboard
• 2 x Mercury Cell Accelerator Boards (CAB)
• 2 x JPEG 2000 boards
• PCI Express (PCI-E) bus

GPS/INS
data

Sensor package

Video

and

GPS/IMU
data

4 x

AMD motherboard
CAB
JPEG 2000
CAB
JPEG 2000
Disk controller

PCI-E
Post-Processing Software

Current CONOPS

• Record video in-flight
• Apply registration and detection on the ground
• Analyze results on the ground

Future CONOPS

• Record video in-flight
• Apply registration and detection in-flight
• Analyze data on the ground

```plaintext
read(S)
gaussianPyramid(S)
for (nLevels) {
    for (nIters) {
        D = projectiveTransform(S, C)
        C = opticFlow(S, D)
    }
}
write(D)
```

Disk

S

D
Real-Time Processing Software

Step 1: Create skeleton DIT-DAT-DOT

Input and output of DAT should match input and output of application

Tasks and Conduits separate I/O from computation

* DIT – Data Input Task, DAT – Data Analysis Task, DOT – Data Output Task
Real-Time Processing Software

Step 2: Integrate application code into DAT

Replace disk I/O with conduit reader and writer

Replace DAT with application code

Tasks and Conduits make it easy to change components
Real-Time Processing Software

Step 3: Replace disk with camera

```
read(S)
gaussianPyramid(S) for (nLevels) {
  for (nIters) {
    D = projectiveTransform(S, C)
    C = opticFlow(S, D)
  }
}
write(D)
```

Replace disk I/O with bus I/O that retrieves data from the camera

Input and output of DAT should match input and output of application

Replace disk I/O with bus I/O that retrieves data from the camera
Performance

<table>
<thead>
<tr>
<th># imagers per Cell</th>
<th>Registration Time (w/o Tasks &amp; Conduits)</th>
<th>Registration Time (w/ Tasks &amp; Conduits*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All imagers</td>
<td>1188 ms</td>
<td>1246 ms</td>
</tr>
<tr>
<td>1/2 of imagers</td>
<td>594 ms</td>
<td>623 ms</td>
</tr>
<tr>
<td>1/4 of imagers</td>
<td>297 ms</td>
<td>311 ms</td>
</tr>
<tr>
<td>Real-time Target Time</td>
<td>500 ms</td>
<td>500 ms</td>
</tr>
</tbody>
</table>

Tasks and Conduits incur little overhead

* Double-buffered
Performance vs. Effort

Benefits of Tasks & Conduits

- Isolates I/O code from computation code
  - Can switch between disk I/O and camera I/O
  - Can create test jigs for computation code
- I/O and computation run concurrently
  - Can move I/O and computation to different processors
  - Can add multibuffering

Without Tasks & Conduits
- Runs on 1 Cell procs
- Reads from disk
- Non real-time

With Tasks & Conduits
- Runs on integrated system
- Reads from disk or camera
- Real-time

SLOCs

Registration Software Lines of Code

Without Tasks & Conduits: 37771
With Tasks & Conduits: 38652
2-3% increase
Outline

• Background

• Tasks & Conduits

• Hierarchical Maps & Arrays

• Results

• Summary
## Future (Co-)Processor Trends

<table>
<thead>
<tr>
<th>Multicore</th>
<th>FPGAs</th>
<th>GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM PowerXCell 8i</strong></td>
<td><strong>Xilinx Virtex-5</strong></td>
<td><strong>NVIDIA Tesla C1060</strong></td>
</tr>
<tr>
<td>- 9 cores: 1 PPE + 8 SPE</td>
<td>- Up to 330,000 logic cells</td>
<td>- PCI-E x16</td>
</tr>
<tr>
<td>- 204.8 GFLOPS single precision</td>
<td>- 580 GMACS using DSP slices</td>
<td>- ~1 TFLOPS single precision</td>
</tr>
<tr>
<td>- 102.4 GFLOPS double precision</td>
<td>- PPC 440 processor block</td>
<td>- 225 W peak, 160 W typical</td>
</tr>
<tr>
<td>- 92 W peak (est.)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tilera TILE64</th>
<th>Curtis Wright CHAMP-FX2</th>
<th>ATI FireStream 9250</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 64 cores</td>
<td>- VPX-REDI</td>
<td>- PCI-E x16</td>
</tr>
<tr>
<td>- 443 GOPS</td>
<td>- 2 Xilinx Virtex-5 FPGAs</td>
<td>- ~1 TFLOPS single precision</td>
</tr>
<tr>
<td>- 15 – 22 W @ 700 MHz</td>
<td>- Dual-core PPC 8641D</td>
<td>- ~200 GFLOPS double precision</td>
</tr>
</tbody>
</table>

- * Information obtained from manufacturers’ websites
Summary

• Modern DoD sensors have tight SWaP constraints
  – Multicore processors help achieve performance requirements within these constraints

• Multicore architectures are extremely difficult to program
  – Fundamentally changes the way programmers have to think

• PVTOL provides a simple means to program multicore processors
  – Refactored a post-processing application for real-time using Tasks & Conduits
  – No performance impact
  – Real-time application is modular and scalable

• We are actively developing PVTOL for Intel and Cell
  – Plan to expand to other technologies, e.g. FPGA’s, automated mapping
  – Will propose to HPEC-SI for standardization
Acknowledgements

Persistent Surveillance Team
• Bill Ross
• Herb DaSilva
• Peter Boettcher
• Chris Bowen
• Cindy Fang
• Imran Khan
• Fred Knight
• Gary Long
• Bobby Ren

PVTOL Team
• Bob Bond
• Nayda Bliss
• Karen Eng
• Jeremiah Gale
• James Geraci
• Ryan Haney
• Jeremy Kepner
• Sanjeev Mohindra
• Sharon Sacco
• Eddie Rutledge