Programming Examples that Expose Efficiency Issues for the Cell Broadband Engine Architecture

William Lundgren (wlundgren@gedae.com, Gedae), Rick Pancoast (Lockheed Martin), David Erb (IBM), Kerry Barnes (Gedae), James Steed (Gedae)

HPEC 2007
Introduction

- Cell Broadband Engine (Cell/B.E.) Processor
- Programming Challenges
  - Distributed control
  - Distributed memory
  - Dependence on alignment for performance
- Synthetic Aperture Radar (SAR) benchmark
- Gedae is used to perform the benchmark
- If programming challenges can be addressed, great performance is possible
  - 116X improvement over quad 500MHz PowerPC board
Cell/B.E. Architecture

- Power Processing Element (PPE)
- Eight Synergistic Processing Elements (SPE)
  - 4 SIMD ALUs
  - DMA Engines
  - 256 kB Local Storage (LS)
- System Memory
  - 25 GB/s
- Element Interconnect Bus (EIB)
  - Over 200 GB/s
Gedae Addresses the Software Architecture

- Software architecture defines how software is distributed across processors like the SPEs
- Optimizing the software architecture requires a global view of the application
- This global view cannot be obstructed by libraries
Gedae’s Approach is Automation

- The functional specification is specified by the programming language.
- The implementation specification defines how the functionality is mapped to the HW (i.e., the software architecture).
- Automation, via the compiler, forms the multithreaded application.
Synthetic Aperture Radar Algorithm
Stages of SAR Algorithm

- **Partition**
  - Distribute the matrix to multiple PEs

- **Range**
  - Compute intense operation on the rows of the matrix

- **Corner Turn**
  - Distributed matrix transpose

- **Azimuth**
  - Compute intense operation on the rows of \([ M(i-1) \ M(i) ]\)

- **Concatenation**
  - Combine results from the PEs for display
Stages Execute Sequentially

- Range processing
- Distributed Transpose
- Azimuth processing
SAR Performance

Platforms used
- Quad 500 MHz PowerPC AltiVec Board
- IBM QS20 Cell/B.E. Blade Server (using 8 SPEs at 3.2 GHz)

Comparison of large SAR throughput
- Quad PowerPC Board 3 images/second
- IBM QS20 347.2 images/second

Maximum achieved performance on IBM QS20

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 SPEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>112.9 GFLOPS</td>
</tr>
<tr>
<td>Corner Turn</td>
<td>12.88 GB/s</td>
</tr>
<tr>
<td>Azimuth</td>
<td>128.4 GFLOPS</td>
</tr>
<tr>
<td>TOTAL SAR</td>
<td>97.94 GFLOPS</td>
</tr>
</tbody>
</table>
Synthetic Aperture Radar Algorithm

Tailoring to the Cell/B.E.
Large images do not fit in LS
- Size of each SPE’s LS: 256 kB
- Size of example image: 2048 x 512 x 4 B/w = 4 MB

Store large data sets in system memory
Coordinate movement of data between system memory and LS

System Memory
- 256 MB on Sony Playstation 3
- 1 GB on IBM QS20

25GB/s
Strip Mining

- Strip mine data from system memory
  - DMA pieces of image (rows or tiles) to LS of SPE
  - Process pieces of image
  - DMA result back to system memory
Gedae Automated Implementation of Strip Mining

- **Unmapped memory type**
  - Platform independent of specifying memory outside of the PE’s address space, such as system memory

- **Gedae can adjust the granularity**
  - Up to increase vectorization
  - Down to reduce memory use

- **Specify rowwise processing of a matrix as vector operations**
  - Use matrix-to-vector and vector-to-matrix boxes to convert
  - Gedae can adjust the implementation to accommodate the processor
Synthetic Aperture Radar Algorithm

Range Processing
Range Processing

- Break matrix into sets of rows
- Triple buffering used so new data is always available for processing

<table>
<thead>
<tr>
<th>Time</th>
<th>DMA to LS</th>
<th>Process</th>
<th>DMA from LS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Subarray 0 → Buf 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Subarray 1 → Buf 1</td>
<td>Buf 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Subarray 2 → Buf 2</td>
<td>Buf 1</td>
<td>Buf 0 → Subarray 0</td>
</tr>
<tr>
<td>3</td>
<td>Subarray 3 → Buf 0</td>
<td>Buf 2</td>
<td>Buf 1 → Subarray 1</td>
</tr>
<tr>
<td>4</td>
<td>Subarray 4 → Buf 1</td>
<td>Buf 0</td>
<td>Buf 2 → Subarray 2</td>
</tr>
</tbody>
</table>

Repeat pattern

<table>
<thead>
<tr>
<th></th>
<th>DMA to LS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Buf 1</td>
<td>Buf 0 → Subarray N-2</td>
</tr>
<tr>
<td>N+1</td>
<td>Buf 1</td>
<td>Buf 1 → Subarray N-1</td>
</tr>
</tbody>
</table>
Implementation of Range

Get rows from system memory

Range processing

Put rows into system memory
Trace Table for Range Processing

- **Vector routines**
  - FFT (2048) 5.62us
  - Real/complex vector multiply (2048) 1.14us

- **Communication**
  - Insert 0.91us
  - Extract 0.60us

- **Total**
  - 8.27us per strip
  - 529us per frame
  - 832us measured

- **Scheduling overhead**
  - 303us per frame
  - 256 primitive firings
Scheduling Overhead

- Gaps between black boxes are
  - Static scheduling overhead: determine next primitive in current thread
  - Dynamic scheduling overhead: determine next thread
- Static scheduling overhead will be removed by automation
Synthetic Aperture Radar Algorithm

Corner Turn
Distributed Corner Turn

- Break matrix into tiles
- Assign each SPU a set of tiles to transpose
- Four buffers in LS, two inputs and two outputs

<table>
<thead>
<tr>
<th>Time</th>
<th>DMA to LS</th>
<th>Process</th>
<th>DMA from LS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,0 → Buf0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0,1 → Buf1</td>
<td>Buf0 → Buf2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0,2 → Buf0</td>
<td>Buf1 → Buf3</td>
<td>Buf2 → 0,0</td>
</tr>
<tr>
<td>3</td>
<td>0,3 → Buf1</td>
<td>Buf0 → Buf2</td>
<td>Buf3 → 1,0</td>
</tr>
<tr>
<td>4</td>
<td>0,4 → Buf0</td>
<td>Buf1 → Buf3</td>
<td>Buf2 → 2,0</td>
</tr>
<tr>
<td>R*C-1</td>
<td></td>
<td>Repeat pattern</td>
<td>Buf2 → R-1,C-1</td>
</tr>
</tbody>
</table>
Implementation of Corner Turn

- Get tiles from system memory
- Transpose tiles on SPUs
- Put tiles into system memory
Vector routine
- Matrix transpose (32x32) 0.991us

Transfer
- Vary greatly due to contention

Total
- 1661us measured

Scheduling overhead
- 384 primitive firings
Synthetic Aperture Radar Algorithm

Azimuth Processing
Azimuth Processing

- Double buffering of data in system memory provides $M(i-1)$ and $M(i)$ for azimuth processing.
- Triple buffering in LS allows continuous processing.
- Output DMA’ed to separate buffer in system memory.
Implementation of Azimuth

Get tiles from system memory

Azimuth processing

Put tiles into system memory
Trace Table for Azimuth Processing

- **Vector routines**
  - FFT/IFFT (1024) 2.71us
  - Complex vector multiply (1024) 0.618us

- **Communication**
  - Insert 0.229us
  - Get 0.491us

- **Total**
  - 6.76us per strip
  - 1731us per frame
  - 2058us measured

- **Scheduling overhead**
  - 327us per frame
  - 1280 primitive firings
Synthetic Aperture Radar Algorithm

Implementation Settings
Partition Table is used to group primitives

Map Partition Table is used to assign partitions to processors

Group primitives by partition name

Map to processor numbers
Implementation of Strip Mining

- Subscheduling is Gedae’s method of applying strip mining
- Gedae applies maximum amount of strip mining
  - Low granularity
  - Low memory use
- User can adjust the amount of strip mining to increase the vectorization

Result of automated strip mining
Synthetic Aperture Radar Algorithm

Efficiency Considerations
Distributed Control

- SPEs are very fast compared to the PPE
  - SPEs can perform 25.6 GFLOPS at 3.2 GHz
  - PPE can perform 6.4 GFLOPS at 3.2 GHz
- PPE can be a bottleneck
- Minimize use of PPE
  - Do not use the PPE to control the SPEs
  - Distribute control amongst the SPEs
- Gedae automatically implements distributed control
Alignment Issues

- Misalignment can make a large impact in performance
- Input and output of DMA transfers must have same alignment

Gedae automatically enforces proper alignment to the extent possible
Destination of DMA List transfers are
- Contiguous
- On 16 byte boundaries

Not Possible with DMA List

Possible but not Always Useful
Implications to Image Partitioning

- **Rowwise partitioning**
  - Rows should be 16 byte multiples

  ![Rowwise Partitioning Example]

- **Tile partitioning**
  - Tile dimensions should be 16 byte multiples

  ![Tile Partitioning Example]

Inefficient – source not 16 B aligned

Inefficient – source and destination for each row have different alignments
Evidence of Contention

- Performance of 8 SPE implementation is only 50% faster than 4 SPE implementation
- Sending tiles between system memory and LS is acting like a bottleneck
- Histogram of tile get/insert shows more variation in 8 SPE execution
Great performance and speedup can be achieved by moving algorithms to the Cell/B.E. processor.

That performance cannot be achieved without knowledge and a plan of attack on how to handle:
- Streaming processing through the SPE’s LS without involving the PPE
- Using the system memory and the SPEs’ LS in concert
- Use of all the SIMD ALU on the SPEs
- Compensating for alignment in both vector processing and transfers

Gedae can help mitigate the risk of moving to the Cell/B.E. processor by automating the plan of attack for these tough issues.