Multicore Programmers Need Help

• Parallel programming is decades old...
• But parallel programming is new to most programmers (and programs)
• Libraries optimized across multiple cores are quickest route to performance
Outline

• Cell processor refresher
• Multicore matrix multiplication library
  ▪ Design and performance
• Bonus slides on multicore FFT library performance
• Where a multicore offload library fits into the bigger picture of multicore programming
Cell BE Processor Block Diagram

- **Cell BE processor boasts nine processors on a single die**
  - 1 Power processor
  - Up to 8 vector processors
- **Computational performance**
  - 205 GFLOPS @ 3.2 GHz
  - 410 GOPS @ 3.2 GHz
- **High-speed data ring connects everything**
  - 205 GB/s maximum sustained bandwidth
- **High-performance chip interfaces**
  - 25.6 GB/s XDR main memory bandwidth
Programming the Cell Processor

• **Easiest aspects of programming the Cell processor**
  - Very deterministic SPE performance
  - Generous ring bandwidth
  - Standards-compliant Power core

• **Getting top performance**
  - 256KB SPE local store for code and data:
    - Minimize code
    - Decompose algorithm to work on chunks that fit in local store
  - Explicit DMAs of code and data between local store and main memory
    - Performance best with 128 byte aligned data in granularity of 128 bytes
  - 128 bit vector engine:
    - Vectorize inner loops
  - Design data decomposition that:
    - Optimizes DMA alignment constraints and
    - Presents data in chunks that can be processed in parallel by vector engine
Multicore Matrix Multiplication Library

A * B → C
Multicore Matrix Multiplication Library

\[ \text{nr} \left\{ \begin{array}{c}
\text{A} \\
d\text{pl} \\
\text{(dot product length)}
\end{array} \right\} \times \left\{ \begin{array}{c}
\text{B} \\
\text{dpl}
\end{array} \right\} \rightarrow \left\{ \begin{array}{c}
\text{C} \\
\text{nc} \\
\text{nr}
\end{array} \right\} \]
Multicore Matrix Multiplication Library

- **Supports**
  - Rectangular matrices
  - Sizes in increments of 32 row or columns
  - Optional accumulation $C = C + A \times B$
  - Optional pre-transposition of A or B or both
  - Selectable parallelism (number of SPEs)
- **Part of MultiCore SAL (Scientific Algorithm Library)**
Matrix Multiplication Implementation

- Different algorithm mappings for different matrix sizes
- Rest of talk covers sizes between 32 and 1024 rows or columns
Problem Decomposition

$A \times B \rightarrow C$

$A [nr=512,dpl=1024] \times B [dpl=1024,nc=512] \Rightarrow C [nr=512,nc=512]$

Each SPE processes entire matrix $A$.

Each SPE processes $nc/p = 64$ column partition of matrix $B$.

Each SPE computes $nc/p = 64$ column partition of matrix $C$. 

$X$
Problem Decomposition

\[ A \text{[nr=512,dpl=1024]} \times B \text{[dpl=1024,nc=512]} \rightarrow C \text{[nr=512,nc=512]} \]

Each SPE processes entire matrix A.

Each SPE processes \( nc/p = 64 \) column partition of matrix B.

Each SPE computes \( nc/p = 64 \) column partition of matrix C.

Inner loop multiplies 8 x \( dpl \) element tile from A with \( dpl \times 32 \) tile from B to produce 8 x 32 tile of C.
Why These Sizes?

• **Chose to store entire dot product in an SPE**
  - Want to maximize dot product length for efficient inner loop
  - But also want to process enough columns at once to make strided transfers of B and C tiles efficient
    • 32 columns: 128 byte DMAs
  - Multiple columns also make vectorization easier

• **Local store usage**
  - Two A buffers: $2 \times 8 \times 1024 \times 4 \text{ bytes/float} = 64K$
  - B buffer: $32 \times 1024 \times 4 \text{ bytes/float} = 128K$
  - C buffer: $8 \times 32 \times 4 \text{ bytes/float} = 1K$
  - Total: $193K$
Streaming Matrix A into Local Store

• Each SPE reads all of matrix A eight rows at a time
  ▪ XDR bandwidth can be the bottleneck
Streaming Matrix A into Local Store

• Each SPE reads all of matrix A eight rows at a time
  ▪ XDR bandwidth can be the bottleneck

• Idea: Some SPEs stream data to other SPEs
## What’s the Best Streaming Strategy?

<table>
<thead>
<tr>
<th>SPEs pulling A from XDR</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 through 7</td>
<td>159.0</td>
</tr>
<tr>
<td>0, 2, 4 and 6</td>
<td>170.2</td>
</tr>
<tr>
<td>0 and 4</td>
<td>169.0</td>
</tr>
<tr>
<td>0</td>
<td>169.7</td>
</tr>
</tbody>
</table>
## Matrix Multiply Library Performance

<table>
<thead>
<tr>
<th>Matrix Dimensions</th>
<th>GFLOPS</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>nr</td>
<td>nc</td>
<td>dpl</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>768</td>
<td>768</td>
<td>768</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>

- As part of a library, implementation must satisfy additional goals beside raw performance
  - Flexibility in data size and organization
  - Options such as accumulation
  - Selectable parallelism
  - Compatible calling sequence
### Performance Comparison (GFLOPS)

<table>
<thead>
<tr>
<th>Matrix Dimensions</th>
<th>Mercury (row major)</th>
<th>IBM SDK (block layout)</th>
<th>Hackenberg (row major)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nr</td>
<td>nc</td>
<td>dpl</td>
<td>Reported</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>512</td>
<td>149</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>1024</td>
<td>162</td>
</tr>
<tr>
<td>768</td>
<td>768</td>
<td>768</td>
<td>163</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>170</td>
</tr>
</tbody>
</table>

- IBM SDK 2.1 matrix multiplication example
  - Square matrices, power of two sizes
  - Block layout only
  - * 174 GFLOPS for 512x512 achieved only for 1000 iterations of same matrix

- Daniel Hackenberg, TU Dresden, May 2007
  - [http://www.fz-juelich.de/zam/datapool/cell/Performance_Measurements_on_Cell.pdf](http://www.fz-juelich.de/zam/datapool/cell/Performance_Measurements_on_Cell.pdf)
  - Square matrices, size increments of 64
  - Row major and block layout
  - Accumulation option
More MC-SAL Performance: Large 2D FFTs

- MC-SAL API called from PPE
- Each FFT performed in parallel on up to 8 SPEs
- Each FFT is too large to fit in the aggregate of the SPE local stores, but small enough that the row and column FFTs fit within local store

<table>
<thead>
<tr>
<th># rows</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td>51</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>45</td>
<td>52</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>44</td>
<td>52</td>
<td>59</td>
<td>63</td>
</tr>
<tr>
<td>2048</td>
<td>44</td>
<td>52</td>
<td>59</td>
<td>63</td>
<td>67</td>
</tr>
<tr>
<td>4096</td>
<td>51</td>
<td>55</td>
<td>63</td>
<td>67</td>
<td>61</td>
</tr>
</tbody>
</table>

MC-SAL 2D FFT performance (GFLOPS) on 8 SPEs (called from PPE, data starts and ends in XDR)
MC-SAL Performance: Streaming Small FFTs

- MC-SAL API performs a batch of 1D FFTs
- Each FFT executed on a single SPE
- Up to 8 SPEs used in parallel

<table>
<thead>
<tr>
<th>N</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>58</td>
</tr>
<tr>
<td>1024</td>
<td>65</td>
</tr>
<tr>
<td>2048</td>
<td>70</td>
</tr>
<tr>
<td>4096</td>
<td>77</td>
</tr>
</tbody>
</table>

Performance for 1000 FFTs using 8 SPEs (called from PPE, data starts and ends in XDR)
1. Compile/run application on general-purpose single core (Cell’s PPE)
2. Introduce function-offload model
   - Replace compute-intensive calls with calls to offload library (MultiCore-SAL)
3. For further improvement, selectively develop custom offload functions to replace offload library calls
   - E.g., fuse functions on SPE to reduce number of SPE-XDR transfers
   - Use SPE-local library (SPE-SAL) and data movement middleware (MultiCore Framework)
MultiCore Scientific Algorithm Library

- Large FFTs, fast convolutions and matrix operations
- Batch operations for smaller sizes
- Also compatible with single-core SAL API
- Compatible with MultiCore Framework
  - For explicit data movement and SPE computation
  - Example or template data-flow code provided for common algorithms
  - User can insert appropriate math (SPE-SAL)
MultiCore Framework Data Movement

PPE

input data

output data

input tile channel

SPE 0

compute

output tile channel

SPE 1

SPE 3
Summary

- Demonstrated superb performance for matrix multiplication on Cell processor
- Function offload libraries provide easiest path to good performance on multicore processors
  - No new languages to learn
  - Also provide portability between diverse multicore architectures
- Need ability to develop custom offload functions to extract maximum performance