Simple, Efficient, Portable Decomposition of Large Data Sets

William Lundgren (wlundgren@gedae.com, Gedae), David Erb (IBM), Max Aguilar (IBM), Kerry Barnes (Gedae), James Steed (Gedae)

HPEC 2008
Introduction

The study of High Performance Computing is the study of
– How to move data into fast memory
– How to process data when it is there

Multicores like Cell/B.E. and Intel Core2 have hierarchical memories
– Small, fast memories close to the SIMD ALUs
– Large, slower memories offchip

Processing large data sets requires decomposition
– Break data into pieces small enough for the local storage
– Stream pieces through using multibuffering
Each SPE core has a 256 kB local storage
Each Cell/B.E. chip has a large system memory
Intel Quad Core Memory Hierarchy

- Caching on Intel and other SMP multicores also creates memory hierarchy
Optimization of Data Movement

- Optimize data movement using software
- **Upside**
  - Higher performance possibilities
- **Downside**
  - Complexity beyond the reach of many programmers
- **In analogy, introduction of Fortran and C**
  - The CPU was beyond the reach of many potential software developers
  - Fortran and C provide automatic compilation to assembly
  - Spurred the industry

*Multicores require the introduction of fundamentally new automation.*
Gedae Background

We can understand the problem by considering the guiding principles of automation that effectively addresses the problem.
Structure of Gedae

- Developer
- Functional Model
- Implementation Specification
- Compiler
- Analysis Tools
- Threaded Application
- Thread Manager
- SW / HW System
- Hardware Model
Guiding Principle for Evolution of Multicore SW Development Tools

- Functional model
- Architecture-specific details
- Implementation specification
- Implementation
- Compiler
- Complexity
- Libraries
**Language – Invariant Functionality**

- **Functionality must be free of implementation policy**
  - C and Fortran freed programmer from specifying details of moving data between memory, registers, and ALU
  - Extend this to multicore parallelism and memory structure

- **The invariant functionality does not include multicore concerns like**
  - Data decomposition/tiling
  - Thread and task parallelism

- **Functionality must be easy to express**
  - Scientist and engineers want a thinking tool

- **Functional expressiveness must be complete**
  - Some algorithms are hard if the language is limited
Language Features for Expressiveness and Invariance

- Stream data (time based data) *
- Stream segments with software reset on segment boundaries *
- Persistent data – extends from state* to databases ‡
- Algebraic equations (HLL most similar to Mathcad) ‡
- Conditionals †
- Iteration ‡
- State behavior †
- Procedural *

* These are mature language features
† These are currently directly supported in the language but will continue to evolve
‡ Support for directly expressing algebraic equations and iteration, while possible to implement in the current tool, will be added to the language and compiler in the next major release. Databases will be added soon after.
Library Functions

- Black box functions hide essential functionality from compiler
- Library is a vocabulary with an implementation
  \[ \text{conv}(\text{float } \ast \text{in}, \text{float } \ast \text{out}, \text{int } R, \text{int } C, \text{float } \ast \text{kernel}, \text{int } KR, \text{int } KC); \]
- Algebraic language is a specification
  \[ \text{range } i=0..R-1, j=0..C-1, i1=0..KR-1, j1=0..KC-1; \]
  \[ \text{out}[i][j] += \text{in}[i+i1][j+j1] \ast \text{kernel}[i1][j1]; \]

Other examples:
\[
\begin{align*}
\text{As}[i][j] &\ += \text{B}[i+i1][j+j1]; /* \text{kernel of ones} */ \\
\text{Ae}[i][j] &\ |= \text{B}[i+i1][j+j1]; /* \text{erosion} */ \\
\text{Am}[i][j] &\ = \text{As}[i][j] > (Kz/2); /* \text{majority operation} */
\end{align*}
\]
Library Functions

A simple example of hiding essential functionality is tile extraction from a matrix
- Software structure changes based on data size and target architecture
- Library hides implementation from developer and compiler
Features Added to Increase Automation of Example Presented at HPEC 2007
New Features

- New language features and compiler functionality provide increased automation of hierarchical memory management

Language features
- Tiled dimensions
- Iteration
- Pointer port types

Compiler functions
- Application of stripmining to iteration
- Inclusion of close-to-the-hardware List DMA to get/put tiles
- Multibuffering
- Accommodation of memory alignment requirements of SPU and DMA
Matrix Multiplication Algorithm
Distributed Algorithm

- **Symbolic Expression**
  \[ A[i][j] += B[i][k] * C[k][j] \]

- **Tile operation for distribution and small memory**
  \[ i->p, i2; j->j1, j2; k->k1, k2 \]
  \[ [p][j1]A[i2][j2] += [p][k1]B[i2][k2] * [k1][j1]C[k2][j2] \]

- **Process p sum spatially and k1 and j1 sums temporally**

- **Accumulate in local store, then transfer result tiles back to system memory**
Data Partitioning by Processor

- Each processor computes different set of rows of “a”

Blue translucent boxes indicate these boxes will migrate to implementation and compiler

System Memory

Data for Processor 0

Data for Processor 1

Data for Processor 2

...
Temporal Data Partitioning

- Fetch tiles from system memory
  - Automatically incorporate DMA List transfer
- Compute the sum of the tile matrix multiplies
- Reconstitute result in system memory
Stripmining and Multibuffering

- Stripmining this algorithm will process the matrix tile-by-tile instead of all at once
  - Enabling automated stripmining adds this to the compilation
- Multibuffering will overlap DMA of next tile with processing of current tile
  - Multibuffering table allows this to be turned off and on
13 kernels
- Each kernel has 10 lines of code or less in its Apply method
- Future version will be one kernel with 1 line of code defined using algebraic expression

Automation ratio (internal kernels added / original kernels / processors)
- Internal kernels added: 276
- Current ratio: 2.65
- Future ratio: 36

Runs at 173.3 GFLOPs on 8 SPEs for large matrices
- Higher rates possible using block data layout, up to 95% max throughput of processor
Polar Format Synthetic Aperture Radar Algorithm
Algorithm

complex out[j][i] pfa_sar(complex in[i][j],
    float Taylor[j], complex Azker[i2]) {
    t1[i][j] = Taylor[j] * in[i][j]
    rng[i] = fft(t1[i]); /* FFT of rows */
    cturn[j][i] = rng[i][j];
    adjoin[j][i2](t) = i2 < R ? cturn[i2][i](t) : cturn[j][i2-R](t-1) ;
    t2[j] = ifft(adjoin[j]);
    t3[j][i2] = Azker[i2] * t2[j][i2];
    azimuth[j] = fft(t3[j]);
    out[j][i] = azimuth[j][i];
}
Analysis of Code Complexity for Benchmark From HPEC 2007

- 33 kernels
  - 7 tiling kernels specially crafted for this application
  - 5 data allocation kernels specially crafted for this application
- DMA transfers between system memory and SPE local storage coded by hand using E library
- Multibuffering is incorporated into the kernels by hand
- The tiling kernels are very complex
  - 80 to 150 lines of code each
  - 20 to 100 lines of code in the Apply method

*a productivity tool should do better!*
Analysis of Code Complexity and Performance

- 23 kernels
- Each kernel has 10 lines of code or less in its Apply method
- Automation ratio
  - Internal kernels added: 1308
  - Current ratio: 7.11
  - Future ratio: 20.67
- Performance:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>GFLOPS</th>
<th>GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2007</td>
<td>2008</td>
</tr>
<tr>
<td>TOTAL SAR</td>
<td>81.1</td>
<td>86.4</td>
</tr>
</tbody>
</table>
Backprojection Synthetic Aperture Radar Algorithm
Backprojection – the Technique

2D Image

(x,y) /* image element indices */

r[p][x][y] /* range from observation point to image element*/

in[p][f] /* pulse returns*/

(Flight path)

(xo[p],yo[p],zo[p]) /* observation point indices */
Algorithm

complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X-1, y = Y-1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
    in2[p] = ifft(in1[p])
    rng[p][y][x] = sqrt((xo[p]-x[x])^2 + (yo[p]-y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p]* rng[p][y][x]);
    rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / C;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
                  in2[p][irbin[p][y][x]+1]*w1[p][y][x]) * phase[p][y][x];
}
Comparison of Manual vs. Automated Implementation

- **Automation ratio**
  - Internal kernels added: 1192
  - Current ratio: 2.26
  - Future ratio: 4.13

- **Processing time for manual results were reported at IEEE RADAR 2008 conference**

<table>
<thead>
<tr>
<th>Npulses</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (mSec)</td>
<td>35.7</td>
<td>285.1</td>
<td>2,368.8</td>
<td>18,259.4</td>
</tr>
</tbody>
</table>

- **Processing time for automated memory transfers with tiling**

<table>
<thead>
<tr>
<th>Npulses</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (mSec)</td>
<td>35.1</td>
<td>280.6</td>
<td>2242.3</td>
<td>17,958.2</td>
</tr>
</tbody>
</table>
Summary and Roadmap

Gedae’s tiling language allows the compiler to manage movement of data through hierarchical memory
  – Great reduction in code size and programmer effort
  – Equivalent performance

Gedae Symbolic Expressions will take the next step forward in ease of implementation
  – Specify the algorithm as algebraic code
  – Express the data decomposition (both spatially and temporally)
  – The compiler handles the rest
End of Presentation
Appendix: Gedae’s Future Direction
Towards Full Automation
Implementation Tools – Automatic Implementation

- Rule Based Engine
- Developer
- Functional Model
- Compiler
- Threaded Application
- Hardware Model with Characterization
- SW / HW System
- Thread Manager
- Implementation Specification
- Software Characterization on HW Model
- Analysis Tools++
Appendix: Cell/B.E. Architecture Details and Tiled DMA Characterization
Cell/B.E Compute Capacity and System Memory Bandwidth

- Maximum flop capacity - 204.8 Gflop/sec 32 bit (4 byte data)
  - 3.2 GHz * 8 flop/SPU * 8 SPU

- Maximum memory bandwidth – 3.2 GWords/sec
  - 25.6 / 4 / 2 words / function / second
    - 25.6 GB/sec
    - 4 bytes/word
    - 2 words/function (into and out-of memory)

- Ideal compute to memory ratio – 64 flops per floating point data value
  - 204.8 / 3.2
Practical Issues

Degradation of memory bandwidth
- Large transfer alignment and size requirements
  - Need 16 byte alignment on source and destination addresses
  - Transfer size must be multiple of 16 bytes
- DMA transfers have startup overhead
  - Less overhead to use list DMA than to do individual DMA transfers

Degradation of compute capacity
- Compute capacity is based on:
  - add:multiply ratio of 1:1
  - 4 wide SIMD ALU
- Filling and emptying ALU pipe
- Pipeline latency
- Data shuffling using SIMD unit
Effect of Tile Size on Throughput

Throughput vs Tile Row Length

(Times are measured within Gedae)
Appendix: Polar Format SAR Description and Flow Graph Specification
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
Simplifications

- **Tile dimensions specify data decomposition**
  - Input: stream float in\([Rt:R][Ct:C]\]
  - Output: stream float out\([Rt/N:R][Ct/N:C]\]
- **This is all the information the compiler needs**
  - User specifies tile size to best fit in fast local storage
  - Compiler stripmines the computation to stream the data through the coprocessors
Simplified Specification: Range

\[ \text{range } p = 0..P-1 \]
Simplified Specification: Corner Turn
Kernels

- Kernels are no more complex than the partitioning kernel shown in the Matrix Multiply example.
- Only difference is it partitions split complex data!
Appendix: Backprojection SAR Symbolic Expression Code Analysis
complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X-1, y = Y-1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
    in2[p] = ifft(in1[p])
    rng[p][y][x] = sqrt((xo[p]-x[x])^2 + (yo[p]-y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p] * rng[p][y][x]);
    rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / C;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
                  in2[p][irbin[p][y][x]+1]*w1[p][y][x]) * phase[p][y][x];
}
Algorithm

complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {

  range f = f0 * 4, x = X-1, y = Y-1;
  { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; } 
  in2[p] = ifft(in1[p])
  rng[p][y][x] = sqrt((xo[p] - x)^2 + (yo[p] - y)^2 + zo[p]^2);
  dphase[p][y][x] = exp(i*4*PI/C*f0[p]* rng[p][y][x]);
  rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / C;
  irbin[p][y][x] = floor(rbin[p][y][x]);
  w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
  w0[p][y][x] = 1 - w1[p][y][x];
  out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
               in2[p][irbin[p][y][x]+1]*w1[p][y][x]) * phase[p][y][x];
}

Declare range variable (iteraters) needed.
complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X-1, y = Y-1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; } 
    in2[p] = ifft(in1[p])
    rng[p][y][x] = sqrt((xo[p]-x[x])^2 + (yo[p]-y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p]*rng[p][y][x]);
    rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / C;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] + in2[p][irbin[p][y][x]+1]*w1[p][y][x])* phase[p][y][x];
}
complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X-1, y = Y-1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
    in2[p] = ifft(in1[p])
    rng[p][y][x] = sqrt((xo[p]-x[x])^2 + (yo[p]-y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p]*rng[p][y][x]);
    rbin[p][y][x] = int((sqrt[p]) * 2 * DF / C;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
                  in2[p][irbin[p][y][x]+1]*w1[p][y][x]) * phase[p][y][x];
}
complex out[x][y] backprojection(complex in[p][f0], float xo[p],
float yo[p], float zo[p], float sr[p], int X, int Y, float DF,
int Nbins) {
range f = f0 * 4, x = X-1, y = Y-1;
{ in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
in2[p] = ifft(in1[p])
rng[p][y][x] = sqrt((xo[p]-x[x])^2 + (yo[p]-y[y])^2 + zo[p]^2);
dphase[p][y][x] = exp(i*4*PI/C*f0[p] * rng[p][y][x]);
rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / C;
irbin[p][y][x] = floor(rbin[p][y][x]);
w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
w0[p][y][x] = 1 - w1[p][y][x];
out[y][x] += (in2[p][irbin[p][y][x]+1]*w1[p][y][x]) * phase[p][y][x];
}

Calculate the range from every point in the output image to every pulse observation point.
Algorithm

complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
  range f = f0 * 4, x = X-1, y = Y-1;
  { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
  in2[p] = ifft(in1[p])
  rng[p][y][x] = sqrt((xo[p]-x[x])^2 + (yo[p]-y[y])^2 + zo[p]^2);
  dphase[p][y][x] = exp(i*4*PI/C*f0[p]* rng[p][y][x]);
  rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF * 4;
  irbin[p][y][x] = floor(rbin[p][y][x]);
  w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
  w0[p][y][x] = 1 - w1[p][y][x];
  out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
    in2[p][irbin[p][y][x]+1]*w1[p][y][x])*phase[p][y][x];}

Calculate the phase shift from every point in the output image to every pulse observation point.
complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X--1, y = Y--1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
    in2[p] = ifft(in1[p]);
    rng[p][y][x] = sqrt((xo[p] - x[x])^2 + (yo[p] - y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p]* rng[p][y][x]);
    rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / 4;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
                     in2[p][irbin[p][y][x]+1]*w1[p][y][x])* phase[p][y][x];
}

Calculate the range bin corresponding to the range of the image point from the observation point.
Algorithm

complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X-1, y = Y-1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
    in2[p] = ifft(in1[p]);
    rng[p][y][x] = sqrt((xo[p] - x[x])^2 + (yo[p] - y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p]*rng[p][y][x]);
    rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / C;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
                   in2[p][irbin[p][y][x]+1]*w1[p][y][x])* phase[p][y][x];
}

Calculate the linear interpolation weights since range will not in center of bin.
complex out[x][y] backprojection(complex in[p][f0], float xo[p], float yo[p], float zo[p], float sr[p], int X, int Y, float DF, int Nbins) {
    range f = f0 * 4, x = X-1, y = Y-1;
    { in1[p][f] = 0.0; in1[p][f0] = in[p][f0]*W[f0]; }
    in2[p] = ifft(in1[p])
    rng[p][y][x] = sqrt((xo[p] - x[x])^2 + (yo[p] - y[y])^2 + zo[p]^2);
    dphase[p][y][x] = exp(i*4*PI/C*f0[p]*rng[p][y][x]);
    rbin[p][y][x] = Nbins * (rng[p][y][x] - rstart[p]) * 2 * DF / 4;
    irbin[p][y][x] = floor(rbin[p][y][x]);
    w1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
    w0[p][y][x] = 1 - w1[p][y][x];
    out[y][x] += (in2[p][irbin[p][y][x]]*w0[p][y][x] +
        in2[p][irbin[p][y][x]+1]*w1[p][y][x])*phase[p][y][x];
}
Analysis of Code Complexity and Performance

The graphs for the backprojection algorithm – while much simpler than the corresponding C code – are relatively complex compared with the data movement. The complexity of the graph is compounded by the 2 sources of complexity. There is great benefit to using symbolic expressions to replace block diagrams as the input. The comparison is shown in an example in the next chart.
Comparison of Symbolic Expression and Block Diagram

\[
w_1[p][y][x] = rbin[p][y][x] - irbin[p][y][x];
\]
\[
w_0[p][y][x] = 1 - w_1[p][y][x];
\]
\[
out[y][x] += (in2[p][irbin[p][y][x]]*w_0[p][y][x] + in2[p][irbin[p][y][x]+1]*w_1[p][y][x]) * phase[p][y][x];
\]