Tools for HPC

Apolent Corporation
COREquations Division
Emerging Platforms

- Moore's Law
  - Power Wall
  - Memory Wall
  - ...

- Complex Parallel Systems are here
  - Core-to-Core comm./sync. only through L3 cache in AMD Barcelona and INTEL Nehalem proc. (L1/L2 private)
  - Processor-to-Processor comm./sync. even more costly in multi-socket configurations
  - Heterogenous multi-cores
    Cell BE, Host + GPUs (NVIDIA, AMD), Host + Larrabee
Programming Requirements

- Need to carefully examine
  - Data Partitioning
  - Communication
  - Synchronization

(A standard data-parallel implementation may be insufficient)
Relevant Applications/Domains

- Image/Video Processing, Vision, etc.
- Scientific/Engg./Statistical Computation
  - Computer Simulation of Physics, Signal Proc., SCADA/HMI
- Game Physics Engines
  - (SONY) Bullet Physics, (NVIDIA) Physx, (INTEL) Havoc
- CG Animation
  - Autodesk 3DS Max/Maya/XSI, Houdini, PIXAR PRMan, Mental Images. Also (open-source) Blender, Pixie & Aqsis
- Bioinformatics, RNA Structure Prediction
- Geospatial Information Systems (GIS)
  - ESRI ArcGIS, PCI Geomatica. Also (open-source) GRASS
Application Hotspots

- Focus of Attention *(Vision, Image Processing)*
  - Image filtering, edge detection, feature extraction
- Motion Estimation: H.264 *(Video Processing)*
- Numerical Solutions to PDEs, Filters, Regression, Control & Optimization, Parameter Estimation *(Sc./Engg./Stat.)*
- Dynamics *(Physics Simulation)*
  - Fluid/Cloth/Wire/Body Simulation in Games/Movies
- Rendering *(CG Animation)*
  - Global Illumination/Ray Tracing
- Dynamic Programming Kernels *(Bioinformatics)*
- Orthorectification *(GIS)*
Hotspot Characteristics

- Highly Compute-Intensive
- High Data Volume
- Very well structured kernel fragments

Possible Solutions

- Automatically generate libraries
- Develop highly tuned libraries
Automatic Library Generation
(The COREquations Engine)
Automatic Library Generation

- Domain Specific Compiler Engine
- Class of Problems: The Polyhedral Model
  - Incorporates most parts of four dwarfs in the "Berkeley View*"
    (each was expected to require an independent technology)
    - Structured Grids
    - Dense Matrix
    - Most of Dynamic Programming
    - Some of Graphical Models

* The Berkeley View is a recent study by application/compiler/hardware specialists that outlines the most important problem domains for parallel platforms
The Polyhedral Model

- Most dynamic instr. from iterative kernels
- Manual code optimization is error prone and time consuming
- We wish to develop equivalent high level compiler optimizations
  - Analyze program behavior
  - Design custom transformations
  - Guarantee validity
Parallel Code Generation

- **Host owns data** (partitions, communicates & synchronizes)
- "Compute Engines" process data
- Host reports results back to user
- Heterogeneous threads
Specifications

- Single Assignment Loops (actually equations)
- Matrix Multiplication (The HPC "Hello World")

```plaintext
affine matrix_product {P, Q, R|P>0 && Q>0 && R>0} Program Parameters
given float A {i,k | 0<=i<P && 0<=k<Q };
    float B {k,j | 0<=k<Q && 0<=j<R };

returns float C {i,j,k | 0<=i<P && 0<=j<R && k==Q-1 };

using float temp_C {i,j,k | 0<=i<P && 0<=j<R && -1<=k<Q };

local temp_C[i,j,k] = case
    { | k>=0} : temp_C[i,j,k-1]+(A[i,K]*B[K,j]);
    { | k==-1} : 0;
esac;

C[i,j,k] = temp_C[i,j,k];

.```

Program Parameters

- **Input Variables**
  - float A {i,k | 0<=i<P && 0<=k<Q }
  - float B {k,j | 0<=k<Q && 0<=j<R }

- **Output Variable**
  - float C {i,j,k | 0<=i<P && 0<=j<R && k==Q-1 }

- **Local Variable**
  - float temp_C {i,j,k | 0<=i<P && 0<=j<R && -1<=k<Q }

Equation for Accumulator

\[
\text{temp}_C[i,j,k] = \begin{cases}
\text{temp}_C[i,j,k-1] + (A[i,K] \times B[K,j]) & \text{if } k \geq 0 \\
0 & \text{if } k = -1
\end{cases}
\]

Equation for Result

\[
C[i,j,k] = \text{temp}_C[i,j,k];
\]
source("Setup.bsh");
ConnectServer("localhost");
ReadProgram("matrix_product.alphabets");
SetMemoryMap("matrix_product","C",
"(P,Q,R,i,j,k->P,Q,R,i,j)","0,0");
MPICodeGen(8,"32,32,32");

Memory Maps are only needed for output variables
Tool finds optimal memory layout for local variables
Performance I: Scaling

- Near-Perfect Scaling
Performance II: Raw Comparison

- Compare to Hand-Optimized Sequential Code
- Before, our MPI Code Generator, we wanted to generate Good Sequential Code
Sequential Code Generation

- Case Study: PDE Solver from Atmospheric Sc.
  - 2D-3D stencil. Five-point update
    - Over time, elements on an area update their value based on previous value and North, South, East and West neighbors
  - Periodic boundary conditions
    - West border depends on East border, etc.
    - SW corner depends on SE and NW corners
- Hand-Optimized by Nathan Burnett, a Masters Student at CSU
  - Performed optimizations such as loop interchange, fusion, fission, unrolling, removal of lookup-tables ...
Performance (Hand-Optimization)

- Intel CORE2 Duo (2.3Ghz)
EI PDE Solver

through

\[ h[i,j,k] = \text{case} \]
\[ \begin{cases} 
| k==0 \rangle & : h_{\text{init}}[j,i]; \\
| 0<k \rangle & : h[i,j,k-1] - \text{delt} * H0 * \text{delta}[i,j,k-1]; 
\end{cases} \]
esac;

\[ \text{lap}_h[i,j,k] = \text{case} \]
\[ \begin{cases} 
| i==0 \&\& j==0 \rangle & : (h[i+1,j,k] + h[2im_h-1,j,k] + h[i,j+1,k] + h[i,2jm_h-1,k] - 4h[i,j,k]) * \text{inv}_d_\text{sq}; \\
| i==0 \&\& 0<=j<2jm_h \rangle & : (h[i+1,j,k] + h[2im_h-1,j,k] + h[i,j+1,k] + h[i,j,k] - 4h[i,j,k]) * \text{inv}_d_\text{sq}; \\
| 0<i<2im_h-1 \&\& 0<j<2jm_h-1 \rangle & : (h[i+1,j,k] + h[i-1,j,k] + h[i,j+1,k] + h[i,j-1,k] - 4h[i,j,k]) * \text{inv}_d_\text{sq}; 
\end{cases} \]
esac;

da \[ \text{delta}[i,j,k] = \text{case} \]
\[ \begin{cases} 
| k==0 \rangle & : 0; \\
| 0<k \rangle & : \text{delta}[i,j,k-1] - \text{delt} * g * \text{lap}_h[i,j,k]; 
\end{cases} \]
esac;

\[ \text{results}[j,i] = h[i,j,TMAX]; \]

Main Variable

Key Computation

Auxillary Variable

Output
Performance (Hand-Optimization)

- Intel CORE2 Duo (2.3Ghz)

![Bar chart showing performance comparison between original code and hand-optimized versions for different tests.]
Then, I provided code for his equations

- Intel CORE2 Duo (2.3Ghz)
Why is generated code so good?

- Memory Optimization
  - (Prefetch-friendly) Alignment
  - Storage Minimization
- Loop Optimizations (just much more of them): Automatic loop-generator specializes code for all boundary cases
  - Aggressive Fusion
  - Aggressive Fission
Beyond "Hello World"

P=Q=R (=N say). Add triangular constraints

affine triangular_matrix_product {N|N>0}
given float A {i,k | 0<=i<N && 0<=k<=i };
float B {k,j | 0<=k<N && k<=j<N };
returns float C {i,j,k | 0<=i<=j && 0<=j<N && k==i } ||
{i,j,k | 0<=i<N && 0<=j<i && k==j } ;
using float temp_C {i,j,k | 0<=i<N && 0<=j<N && -1<=k<=(i,j) };
through
temp_C[i,j,k] = case
    { | k>=0} : temp_C[i,j,k-1]+(A[i,k]*B[k,j]);
    { | k==-1} : 0;
esac;
C[i,j,k] = temp_C[i,j,k];
.
### Triangular Matrix Multiplication

#### Sequential Code

```c
for i = 0 to n-1 {
    for j = 0 to n-1 {
        C[i,j] = 0;
        for k = 0 to min(i,j) { 
            C[i,j] += A[i,k]*B[k,j]
        }
    }
}
```

Why write equations when you can write code?

#### Iteration Space

- Iteration Space is a polyhedron
- Now, let us generate MPI code (What is each Compute Engine iterating over?)
Triangular Matrix Multiplication

- Targeting memory-constrained architectures
  - Need to tile all dimensions (not necessarily cubic)
  - Many Partial Tiles
  - All Results are from Partial Tiles
  - Result distributed in block-cyclic along \( i \)

CE 1 requires a partial block of inputs from B and a partial block from A
CE 2 requires a partial block of inputs from B but a full block of inputs from A
In Short ...

- Writing MPI code even for a straightforward specialization of Matrix Multiplication is extremely time consuming and error-prone.
- In COREquations, just run the same script for the new specification.
Performance

- Again, Near-Perfect Scaling
Sequential Code for Matrix Multiplication

```java
for i = 0 to n-1 {
    for j = 0 to n-1 {
        C[i,j] = 0;
        for k = 0 to N-1 {
            C[i,j] += A[i,k] * B[k,j]
        }
    }
}
```

- Subsequent reads of A are along the inner index. Memory accesses to A are prefetch friendly.
- Subsequent reads of B are along the outer index. Memory accesses to B are not prefetch friendly.

What about writes to C?

- Writes to C are to the same location throughout the inner loop
A number of choices in code-generation

- Tiling: (shape and size)
  
  FixedTiledCodeGen("48,32,63");

- Different Memory Layouts of Variables
  
  SetMemoryMap("matrix_product","B", 
     
     
     
     ”(P,Q,R,k,j->P,Q,R,j,k)”,"0,0");

- Permutation of Loops
  
  With j as the inner loop, A and C are accessed only once along the inner loop. B[k,j] becomes Prefetch-friendly.
  
  CoB("matrix_product","C",(P,Q,R,i,j,k->P,Q,R,i,k,j));
  
  CoB("matrix_product","temp_C",(P,Q,R,i,j,k->P,Q,R,i,k,j));
Future Work

- We're going to make it easier to program
  - Higher Level Language
    - Matrix Product should simply be specified as
      \[ C[i,j] = \sum[k] A[i,k] \times B[k,j]; \]
    - Domain Inferencing
  - More Analysis (Scheduling, etc.)
  - Get the Human-Out-of-the-Loop

- Porting to Newer Architectures
  (GPUs, Cell, Larrabee, ...)

28
Summary

- For specifications in Alphabets
  - MPI is easy
  - Design Space Exploration is easy
  - Scalable Performance

- Auto-verification
  - Compiler-assisted programming to avoid double-or incomplete-definitions, out-of-bound references ...
    - Major source of errors is eliminated
A new language for generating HPC code for compute-intensive kernels, but also ... An Intermediate Representation for optimization of loops in conventional languages
Inexpensive Design Space Exploration
Simple Matrix Product

Part of Generated Code

Refer to script on slide 7

Execution Time: 33.002811 ssec.
Tiled Matrix Product

```
2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % source("Setup.bsh");
bsh % ConnectServer("localhost");
bsh % ReadProgram("/.../examples/embedded/matrix_product.alphabets");
bsh % AShow();
affine matrix_product {P,Q,R | P>=1 & Q>=1 & R>=1};
given
  float A {i,k | k>=0 & i>=0 & Q-k>=1 & P-i>=1};
  float B {k,j | j>=0 & k>=0 & R-j>=1 & Q-k>=1};
returns
  float C {i,j,k | Q-k>=1 & j>=0 & i>=0 & R-j>=1 & P-i>=1};
using
  float temp_C {i,j,k | k>=0 & j>=0 & i>=0 & Q-k>=1 & P-i>=1};
through
  temp_C[i,j,k] = case
    { | k>=0} : (temp_C[i,j,k-1]+A[i,k]*B[k,j]);
  { | k==1} : 0;
  esac;
  C[i,j,k] = temp_C[i,j,k];
```

Loaded Program

```
[ggupta@kenshin 01:53:02 temp_dir]
gcc -o3 -o tiled_mm matrix_product-fixedtile.c matrix_product-main.c -lm
[ggupta@kenshin 01:53:08 temp_dir]
./tiled_mm 1024 1024 1024
Execution time : 15.598294 sec.
[ggupta@kenshin 01:53:25 temp_dir]
```
Changing the memory layout of B

- Significant performance upgrade by having a prefetch-friendly layout of the input array B
- Can we obtain similar performance with the standard memory layout for B
By having j as the innermost loop index, accesses (to arrays A and B) preserve locality