Many-core Computing
Can compilers and tools do the heavy lifting?

Wen-mei Hwu

FCRP GSRC, Illinois UPCRC, Illinois CUDA CoE, IACAT, IMPACT
University of Illinois, Urbana-Champaign
Outline

• Parallel application outlook

• Heavy lifting in “simple” parallel applications

• Promising tool strategies and early evidence

• Challenges and opportunities

SoC specific opportunities and challenges?
The Energy Behind Parallel Revolution

- GPU in every PC – massive volume and potential impact

3 year shift

Graph showing the performance growth of AMD (GPU), NVIDIA (GPU), and Intel (CPU) over the years with a 3 year shift. Courtesy: John Owens
My Predictions

• Mass market parallel apps will focus on many-core GPUs in the next three to four years
  • NVIDIA GeForce, ATI Radon, Intel Larrabee
  • “Simple” (vector) parallelism
  • Dense matrix, single/multi-grids, stencils, etc.

• Even “simple” parallelism can be challenging
  • Memory bandwidth limitation
  • Portability and scalability
  • Heterogeneity and data affinity
**DRAM Bandwidth Trends**

- Random access BW 1.2% of peak for DDR3-1600, 0.8% for GDDR4-1600 (and falling)
- 3D stacking and optical interconnects will unlikely help.
Dense Matrix Multiplication Example (G80)

Impact of optimizations:
- Memory bandwidth limited
- Instruction throughput limited

Register tiling allows 200 GFLOPS

Volkov and Demmel, SC’08

Ryoo, et al, PPoPP 2008

GFLOPS

optimizations

unroll 1
unroll 2
unroll 4
complete unroll

cannot run

8x8 tiles

1x1
1x2
1x4

16x16 tiles

1x1
1x2
1x4

8x8 tiles
Example: Convolution - Base Parallel Code

- Each parallel task calculates an output element
- Figure shows
  - 1D convolution with K=5 kernel
  - Calculation of 3 output elements
- Highly parallel but memory bandwidth inefficient
  - Uses massive threading to tolerate memory latency
  - Each input element loaded up to K times

Input elements in main memory
Example: convolution using on-chip caching

- Output elements calculated from cache contents
  - Each input element loaded only once
  - Cache pressure - \((K-1+N)\) input elements needed for \(N\) output elements
    - \(7/3 = 2.3\), \(7^2/3^2 = 5.4\), \(7^3 / 3^3 = 12\)
    - For small caches, the benefit can be significantly reduced due to the high-ratio of additional elements loaded.

Input elements first loaded into cache
Example: Streaming for Reduced Cache Pressure

- Each input element is loaded into cache in turn
  - Or a (n-1)D slice in nD convolution
- All threads consume that input element
  - “loop skewing” needed to align the consumption of input elements
  - This stretches the effective size of the on-chip cache
Many-core GPU Timing Results

- Time to compute a 3D $k^3$-kernel convolution on 4 frames of a 720X560 video sequence
  - All times are in milliseconds
  - Timed on a Tesla S1070 using one G280 GPU

<table>
<thead>
<tr>
<th>$k$</th>
<th>Baseline (3.1)</th>
<th>Shared Memory (3.2)</th>
<th>Streaming (3.3)</th>
<th>3D Fourier (3.4)</th>
<th>Hybrid Fourier (3.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>16</td>
<td>11</td>
<td>4</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>44</td>
<td>15</td>
<td>8</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>48</td>
<td>16</td>
<td>39</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>180</td>
<td>77</td>
<td>27</td>
<td>44</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>295</td>
<td>45</td>
<td>45</td>
<td>74</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>454</td>
<td>75</td>
<td>56</td>
<td>56</td>
<td>26</td>
</tr>
</tbody>
</table>
Multi-core CPU Timing Results

- Time to compute a 3D $k^3$-kernel convolution on 4 frames of a 720X560 video sequence
  - All times are in milliseconds
  - Timed on a Dual-Socket Duo-Core 2.4 GHz Opteron system, all four cores used

<table>
<thead>
<tr>
<th>$k$</th>
<th>Baseline (3.1)</th>
<th>Shared Memory (3.2)</th>
<th>Streaming (3.3)</th>
<th>3D Fourier (3.4)</th>
<th>Hybrid Fourier (3.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>136</td>
<td>117</td>
<td>140</td>
<td>128</td>
<td>133</td>
</tr>
<tr>
<td>7</td>
<td>362</td>
<td>289</td>
<td>317</td>
<td>235</td>
<td>152</td>
</tr>
<tr>
<td>9</td>
<td>1018</td>
<td>597</td>
<td>614</td>
<td>208</td>
<td>213</td>
</tr>
<tr>
<td>11</td>
<td>1954</td>
<td>1065</td>
<td>1135</td>
<td>238</td>
<td>237</td>
</tr>
<tr>
<td>13</td>
<td>3590</td>
<td>1733</td>
<td>1771</td>
<td>267</td>
<td>271</td>
</tr>
<tr>
<td>15</td>
<td>6453</td>
<td>2676</td>
<td>2633</td>
<td>338</td>
<td>356</td>
</tr>
</tbody>
</table>
Application Example: Up-resolution of Video

Nearest & bilinear interpolation:
+ Fast but low quality

Bicubic interpolation:
+ Higher quality but computational intensive
Implementation Overview

• **Step 1:** Find the coefficients of the shifted B-Splines.
  - Two single pole IIR filters along each dimension
  - Implemented with recursion along scan lines

• **Step 2:** Use the coefficients to interpolate the image
  - FIR filter for bicubic interpolation implemented as a k=4 2D convolution with \((2+16+2)^2\) input tiles with halos
  - Streaming not required due to small 2D kernel, on-chip cache works well as is.

• **Step 3:** DirectX displays from the GPU
Upconversion Results

- Parallelize bicubic B-spline interpolation
  - Interpolate QCIF (176x144) to nearly HDTV (1232x1008)
  - Improved quality over typical bilinear interpolation
  - Improved speed over typical CPU implementations
    - Measured 350x speedup over un-optimized CPU code
    - Estimated 50x speedup over optimized CPU code from inspection of CPU code
- Real-time!

<table>
<thead>
<tr>
<th>Hardware</th>
<th>IIR</th>
<th>FIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>5 ms</td>
<td>1689 ms</td>
</tr>
<tr>
<td>nVidia GeForce 8800 GTX</td>
<td>1 ms</td>
<td>4 ms</td>
</tr>
</tbody>
</table>
Application Example: Depth-Image Based Rendering

• Three main steps:
  • Depth propagation
  • Color-based depth enhancement
  • Rendering
Color-based depth enhancement

Propagated depth image at color view

- Occlusion removal
- Depth-color bilateral filtering
- Directional disocclusion filling
- Depth edge enhancement

Propagated depth image

Enhanced depth image
**Depth - color bilateral filtering**

\[
d_A = \frac{1}{W_A} \sum_{B \in S_A} G_{\sigma_s^2}(|\vec{x}_A - \vec{x}_B|) \cdot G_{\sigma_r^2}(|I_A - I_B|) \cdot d_B
\]

\[
W_A = \sum_{B \in S_A} G_{\sigma_s^2}(|\vec{x}_A - \vec{x}_B|) \cdot G_{\sigma_r^2}(|I_A - I_B|)
\]

- \(d_A\) : depth value of point \(A\).
- \(I_A\) : color value of point \(A\).
- \(\vec{x}_A = [u_A, v_A]\) : 2D coordinate of point \(A\).
- \(S_A\) : set of \(A\) neighboring points.
- \(G_{\sigma_s}(|\vec{x}|) = \exp\left(-\frac{|\vec{x}|^2}{2\sigma^2}\right)\) : Gaussian kernel.
- \(W_A\) : normalizing term.
DIBR Visual Results

Left view

Right view

Middle view

Rendered view
DIBR Time results

- Depth propagation.
  - Not computationally intensive but hard to parallelize
  - Each pixel in the depth view is be copied to the corresponding pixel in a different color view.
  - 3D-to-2D projection, many-to-one mapping.
  - Atomic functions are used, current work to improve with sort-scan and binning algorithms.

- Depth-color bilateral filter (DCBF)
  - Computational expensive.
  - Similar to 2D convolution. Similar parallelism techniques work well

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Depth propagation</th>
<th>DCBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Intel Core 2 Duo E8400 3.0GHz</td>
<td>38 ms</td>
<td>1041 ms</td>
</tr>
<tr>
<td>GPU NVIDIA GeForce 9800 GT</td>
<td>24 ms</td>
<td>14 ms</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.6x</td>
<td>74.4x</td>
</tr>
</tbody>
</table>
Some upcoming tools
Gluon - specification information enables robust co-parallelization. (Illinois)

• Developers specify pivotal information at function boundaries
  • Heap data object shapes and sizes
  • Object access guarantees
  • Some can be derived from global analyses but others can be practically infeasible to extract from source code.

• Compilers leverage the information to
  • Expose and transform parallelism
  • Perform code and layout transformations for locality
Gluon Parallelism Exposure

Example

```c
struct data {
    float x; float y; float z;
};

int cal_bin(struct data *a, struct data *b) {
    1. __spec(*a: r, (data)[1]);
    2. __spec(*b: r, (data)[1]);
    3. __spec(ret_v: range(0,SZ));

    int bin = . ; /* use *a and *b*/
    return(bin);
}

int *tpacf(int len, struct data *d) {
    4. __spec(d: r, (int)[len]);

    int *hist = malloc(SZ*sizeof(int));
    __spec(hist: (int)[SZ]);

    for (i=0; i < len; i++) {
        for (j = 0; j < len; j++) {
            6. int bin = cal_bin(&d[i],&d[j]);

            7. hist[bin] += 1;
        }
    }
}
```

No side effect on d elements

hist safe to privatize

Data layout can be done safely
Program Dependence Graph Based Application Performance Prediction (Illinois)

Predicting the performance effect of compiler transformations.

Baghsorkhi and Hwu, EPHAM 2009
Automating Memory Coalescing using Gluon and PDG

```c
#define ASIZE 3000
#define TPB 32

void
kernel (float *a, float *b)
{
    __annotation (L"__global__ TPB 1");
    __annotation (L"garray a 2 4 ASIZE ASIZE");
    __annotation (L"garray b 2 4 ASIZE ASIZE");

    int thi = threadIdx.x;
    int bki = blockIdx.x;
    float t = (float) thi + bki;
    int i;

    __annotation (L"BoundChk");
    if (bki * TPB + thi >= ASIZE)
        return;

    for (i = 0; i < ASIZE; i++)
    {
        __annotation (L"loop i 0 ASIZE 1");
        b[(bki*TPB+thi)*ASIZE + i] =
            a[(bki*TPB+thi)*ASIZE + i] * t;
    }
}
```

```c
#define ASIZE 3000
#define TPB 32

__global__ void
kernel (float *a, float *b)
{
    int thi = threadIdx.x;
    int bki = blockIdx.x;
    float t = (float) thi + bki;
    int i;

    int j, End;

    End = ASIZE % TPB == 0 ? ASIZE / TPB : (ASIZE/TPB+1);
    for (j = 0; j < End; j++)
    {
        __syncthreads();
        /* Coalesce loads */
        for (k = 0; k < TPB; k++)
        {
            if (((i+TPB + thi < ASIZE) &&
                (bki*TPB+k)*ASIZE + i + TPB + thi < ASIZE * ASIZE))
                a_shared[i][ti] = a[(bki*TPB + k)*ASIZE + i+TPB + thi];
        }
        __syncthreads();

        /* Conditions:
        * TPB is obey original end && (early exit condition)
        */
        for (i = 0; i < TPB && (j*TPB+i < ASIZE) && (bki + TPB + thi >= ASIZE); i++)
        {
            b_shared[thi][i] = a_shared[thi][i] * t;
        }
        __syncthreads();

        /* Coalesce stores */
        for (k = 0; k < TPB; k++)
        {
            if (((i+TPB + thi < ASIZE) &&
                (bki*TPB+k)*ASIZE + i + TPB + thi < ASIZE * ASIZE))
                b[(bki*TPB + k)*ASIZE + i+TPB + thi] = b_shared[k][thi];
        }
        __syncthreads();
    }
```
Memory Layout Transformation
Lattice-Boltzmann Method Example

Array of Structure: [z][y][x][e]
F(z, y, x, e) = z \cdot |Y| \cdot |X| \cdot |E| + y \cdot |X| \cdot |E| + x \cdot |E| + e

Structure of Array: [e][z][y][x]
F(z, y, x, e) = e \cdot |Z| \cdot |Y| \cdot |X| + z \cdot |Y| \cdot |X| + y \cdot |X| + x
4X faster than AoS on GTX280
The best layout is neither SoA nor AoS

* Tiled Array of Structure, using lower bits in x and y indices, i.e. $x_{3:0}$ and $y_{3:0}$ as lowest dimensions: $[z][y_{31:4}][x_{31:4}][e][y_{3:0}][x_{3:0}]$
  - $F(z, y, x, e) = z \cdot \left| Y \right|/2^i \cdot \left| X \right|/2^i \cdot \left| E \right| \cdot 2^i \cdot 2^j + y_{31:4} \cdot \left| X \right|/2^i \cdot \left| E \right| \cdot 2^i \cdot 2^j + x_{31:4} \cdot \left| E \right| \cdot 2^i \cdot 2^j + e \cdot 2^i \cdot 2^j + y_{3:0} \cdot 2^i + x_{3:0}$

* 6.4X faster than AoS, 1.6X faster than SoA on GTX280:
  - Better utilization of data by neighboring cells
  - This is a scalable layout: same layout works for very large objects.
Summary

• Tools must understand and manage data accesses
  • Partnership between developers and tools
  • Key to “good” parallelism
  • Must balance between developer specification and program analysis
  • Key to portability and productivity

• “Simple” many-core programming tools within reach
  • Memory bandwidth optimizations
  • Parallel execution granularity adjustments
  • Well-known algorithm changes
  • Heterogeneous computing mapping and data transfers
  • Haves and Have-Nots of many-core computing

• http://www.parallel.illinois.edu/
  • Courses, seminars, publications, tools,
  • UPCRC. CUDA Center of Excellence. IACAT. ...
Current Challenges

• Execution Models
  • Currently single kernel execution
  • Moving to multiple kernel steaming

• Irregular Algorithms and Data Structures
  • Data layout and tiling transformations for sparse matrices and spatial data structures need to be developed and automated
  • Graph algorithms lack conceptual foundation for locality

• Usability
  • Tools and interfaces may be still too tedious and confusing for application developers
Thank you! Any questions?
### Applications Entry Timeframes

App developers want at least 3X-5X for user perceived value-add.

<table>
<thead>
<tr>
<th></th>
<th>50 GF</th>
<th>100 GF</th>
<th>200 GF</th>
<th>400 GF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-core</td>
<td>G80</td>
<td>G280</td>
<td>G380 Larrabee</td>
<td></td>
</tr>
<tr>
<td>16-cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500 GF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1TF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 TF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 TF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Time: 24-month generations
- Apps entry point (2008)
- Apps entry point (2011)
FIR implementation

Cubic interpolation for 1D case

\[ k = x - \lfloor \frac{x}{R} \rfloor \cdot R \]

\[ g[x] = c[x-1]w0[k] + c[x]w1[k] + c[x+1]w2[k] + c[x+2]w3[k] \]
Depth propagation

- Propagate depth information from the depth camera to each color camera.

- 2D point to 3D ray mapping relation:
  \[ \vec{r} = \begin{bmatrix} s_{ijk} & t_{ijk} & f & w_{ijk} \end{bmatrix} \begin{bmatrix} x_s & x_t & x_w \end{bmatrix}^T = P \vec{x} \]

- Warping equation: (L. McMillan, 1997)
  \[ \vec{x}_d = P_d^{-1} \left( \frac{P_r \vec{x}_r}{d(\vec{x}_r)} (\vec{C}_r - \vec{C}_d) + P_r \vec{x}_r \right) \]

- Compute new depth values:
  \[ d_d(\vec{x}_d) = \left| \frac{\vec{C}_2 \vec{X}}{\vec{C}_1 + \vec{C}_1 \vec{X}} \right| \]

A form of 2D “histogram” challenging for GPUs

**Notation:**
- \( \{s, t, w\} \) = local view coordinates.
- \( \{i, j, k\} \) = global coordinates.
- \( f \) = focal length of the camera.
- \( P \) = point-to-ray projection matrix.
- \( \vec{r} \) = 3D ray.
- \( \vec{x} \) = 2D coordinate of a pixel.
- \( \vec{X} \) = 3D projection of \( \vec{x} \).
- \( \vec{C} \) = camera center.

Subscript \( r \) = reference view.
Subscript \( d \) = desired view.
Illinois Vision Video (ViVid) Framework

- Constructed by vision experts with parallel programming expertise
- For video analysis, enhancement, and synthesis apps
- Python module bindings for seamless CPU/GPU deployment
  - MPEG2 Video Decoder and file I/O - C++ (through OpenCV)
  - 2D Convolution - C++, Python, CUDA
  - 3D Convolution - C++, Python, CUDA
  - 2D Fourier Transform - C++, Python, CUDA
  - 3D Fourier Transform - C++, Python, CUDA
  - Optical Flow Computation - C++ (through OpenCV)
  - Motion Feature Extraction - C++, Python, CUDA
  - Pairwise distance between 2 collections of vectors - C++, Python, CUDA
- Domain knowledge capture for optimization and auto-tuning

M. Dikman, et al, University of Illinois, Urbana-Champaign
GMAC Heterogeneous Computing Runtime (UPC/Illinois)

- Software-Based Unified CPU/GPU Address Space
  - Same address/pointer used by CPU and GPU
  - No explicit data transfers
- Data reside mainly in GPU memory
  - Close to compute power
  - Occasional CPU access for legacy libraries and I/O
- Customizable automatic data transfers:
  - Transfer everything (safe mode)
  - Transfer dirty data before kernel execution
  - Transfer data as being produced (default)
- Multi-process / Multi-thread support
- CUDA compatible, Linux alpha version available soon.