The End of Denial Architecture and The Rise of Throughput Computing

Bill Dally
Chief Scientist & Sr. VP of Research, NVIDIA
Bell Professor of Engineering, Stanford University
May 19, 2009
Async Conference UNC
Outline

- Performance = Parallelism
- Efficiency = Locality
- Single-thread processors are in denial about these two facts
- We are entering the era of *Throughput Computing*
- Stream Processors
  - Parallel arithmetic units
  - Exposed storage hierarchy
- Stream Programming
  - Bulk operations
- GPU Computing
In 1965 Gordon Moore predicted the number of transistors on an integrated circuit would double every year.

- Later revised to 18 months
- Also predicted $L^3$ power scaling for constant function
- No prediction of processor performance
- Advancements in architecture turn device scaling into performance scaling

Moore, Electronics 38(8) April 19, 1965
The Computer Value Chain

- Moore’s law gives us more transistors
- Architects turn these transistors into more performance
- Applications turn this performance into value for the user
Discontinuity 1
The End of ILP Scaling

Dally et al. The Last Classical Computer, ISAT Study, 2001
Future potential of novel architecture is large (1000 vs 30)

Dally et al. The Last Classical Computer, ISAT Study, 2001
Single-Thread Processor Performance vs Calendar Year

Source: Hennessy & Patterson, CAAQA, 4th Edition
Technology Constraints
CMOS Chip is our Canvas
4,000 64b FPUs fit on a chip

- 64b FPU
- 0.1mm²
- 50pJ/op
- 1.5GHz
200,000 16b MACs fit on a chip

64b FPU
0.1mm$^2$
50pJ/op
1.5GHz

20mm

16b MAC
0.002mm$^2$
1pJ/op
1.5GHz
Moving a word across die = 124 MACs, 10 FMAs
Moving a word off chip = 250 MACs, 20 FMAs

64b FPU
0.1mm$^2$
50pJ/op
1.5GHz

16b MAC
0.002mm$^2$
1pJ/op
1.5GHz

64b 1mm Channel
25pJ/word

16b 1mm Channel
6pJ/word

64b Off-Chip Channel
1nJ/word

16b Off-Chip Channel
250pJ/word

64b Floating Point

16b Fixed Point
Discontinuity 2
The End of $L^3$ Power Scaling

Gordon Moore, ISSCC 2003
Performance = Parallelism

Efficiency = Locality
Single thread processors are in denial about parallelism and locality

They provide two illusions:

Serial execution
- Denies parallelism
- Tries to exploit parallelism with ILP – limited scalability

Flat memory
- Denies locality
- Tries to provide illusion with caches – very inefficient when working set doesn’t fit in the cache
We are entering the era of Throughput Computing
Throughput computing is what matters now

- Latency optimized processors (most CPUs) are improving very slowly
  - Little value is being delivered from the evolution of these processors
- Throughput optimized processors (like GPUs) are still improving at >70% per year
  - This drives new throughput applications that convert this performance to value
- Going forward throughput processors matter, not latency processors
Applications
Scientific Applications

- Large data sets
  - Lots of parallelism
- Increasingly irregular (AMR)
  - Irregular and dynamic data structures
  - Requires efficient gather/scatter
- Increasingly complex models
  - Lots of locality
- Global solution sometimes bandwidth limited
  - Less locality in these phases
Embedded Applications

- Codecs, modems, image processing, etc…
- Lots of data (pixels, samples, etc…)
  - Lots of parallelism
- Increasingly complex
  - Lots of parallelism
  - Lots of data dependent control
- High arithmetic intensity
  - Lots of locality
Performance = Parallelism

Efficiency = Locality

Fortunately, most applications have lots of both.

Amdahl’s law doesn’t apply to most future applications.
Stream Processor Architecture
Organize computation to

- Optimize use of scarce bandwidth
  - Minimize expensive data movement
  - Keep scarce bandwidth resources busy

- Take advantage of plentiful arithmetic
  - Operate in parallel when data is local

- Avoid denial architecture
  - Don’t hide parallelism or locality
    - Flat, serial model inhibits optimization
Optimize use of scarce bandwidth

- Provide rich, exposed storage hierarchy
Optimize use of scarce bandwidth

- Provide rich, exposed storage hierarchy
- Explicitly manage data movement on this hierarchy
- Reduces demand, increases utilization
Optimize use of scarce bandwidth

- Provide rich, exposed storage hierarchy
- Explicitly manage data movement
- Makes execution predictable
- Enables more arithmetic per unit bandwidth
Summary so far

- Arithmetic is cheap, bandwidth is expensive
  - Performance = parallelism
  - Efficiency = Locality
  - And most applications have lots of both
- Optimize use of scarce bandwidth
  - Rich, exposed storage hierarchy
  - Explicit, bulk transfers
  - Reduces demand, increases utilization
  - Enables more arithmetic and predictable execution
What is a Stream Processor?

- Lots of arithmetic units
  - To exploit parallelism
- Rich, exposed storage hierarchy
  - Makes frequent communication inexpensive
  - Makes kernel execution predictable
- Explicit bulk transfers (streams) and bulk operations (kernels)
  - Load/Store machine for streams
- Simple control
  - SIMD x VLIW x Threads
  - Simple synchronization
  - Low Overhead
Some Example Stream Processors
Example Stream Processors

Imagine

Merrimac

Storm-1

Cell

Nvidia GT200
Imagine 1995-2002

- 48 32b FPUs
  - 8 lanes of 6 32b FPUs each
- Exposed storage hierarchy
  - LRFs, SRF, DRAM
- Bulk operations
  - Including gather/scatter
- Conditional streams
- Applications
  - Graphics
    - OpenGL, RTSL, Reyes, Ray Tracing
  - Image processing
  - Codecs
  - Modems
  - Network processing
- 20x performance/W of efficient DSPs in same technology
- Parity with GPUs in same technology
Imagine Architecture (cont.)
Producer-Consumer Locality in the Depth Extractor

Memory/Global Data
- row of pixels
- previous partial sums
- new partial sums
- blurred row
- previous partial sums
- new partial sums
- sharpened row
- filtered row segment
- filtered row segment
- previous partial sums
- new partial sums
- depth map row segment

SRF/Streams

Clusters/Kernels
- Convolution (Gaussian)
- Convolution (Laplacian)
- SAD

1 : 23 : 317
Applications match the bandwidth hierarchy
Merrimac 2001-2005

- 64 64b FP MADD units
- 16 clusters of 4 units each
- Stream cache
- Shared memory
  - Via network
- Scatter-add
## Merrimac Application Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Developed</th>
<th>Compiled</th>
<th>Sustained GFLOP/s</th>
<th>Speedup over Pentium4</th>
<th>Efficiency vs. Pentium4</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamFEM</td>
<td>Brook</td>
<td>StreamC</td>
<td>69</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StreamMD</td>
<td>C</td>
<td>StreamC</td>
<td>46</td>
<td>16.1</td>
<td>34.5</td>
</tr>
<tr>
<td>StreamFLO</td>
<td>Brook</td>
<td>Metacompiled</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StreamCDP</td>
<td>Fortran</td>
<td>StreamC</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGEMM</td>
<td>StreamC</td>
<td>StreamC</td>
<td>117</td>
<td>17.7</td>
<td>37.9</td>
</tr>
<tr>
<td>CONV2D</td>
<td>StreamC</td>
<td>StreamC</td>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT3D</td>
<td>StreamC</td>
<td>StreamC</td>
<td>37</td>
<td>18.7</td>
<td>40.1</td>
</tr>
<tr>
<td>StreamSPAS</td>
<td>C</td>
<td>StreamC</td>
<td>3.1</td>
<td>6</td>
<td>12.6</td>
</tr>
</tbody>
</table>
## Merrimac Bandwidth Hierarchy

<table>
<thead>
<tr>
<th>Application</th>
<th>Sustained GFLOPS</th>
<th>FP Ops / Mem Ref</th>
<th>LRF Refs</th>
<th>SRF Refs</th>
<th>Mem Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamFEM3D (Euler, quadratic)</td>
<td>43.9</td>
<td>19 – 22</td>
<td>153.0M (95.0%)</td>
<td>6.3M (3.9%)</td>
<td>1.8M (1.1%)</td>
</tr>
<tr>
<td>StreamFEM3D (MHD, constant)</td>
<td>39.2</td>
<td>13.8</td>
<td>186.5M (99.4%)</td>
<td>7.7M (0.4%)</td>
<td>2.8M (0.2%)</td>
</tr>
<tr>
<td>GROMACS (StreamMD)</td>
<td>50.1*</td>
<td>11 –16*</td>
<td>108M (95.0%)</td>
<td>4.2M (2.9%)</td>
<td>1.5M (1.3%)</td>
</tr>
<tr>
<td>StreamFLO</td>
<td>12.9*</td>
<td>7.4*</td>
<td>234.3M (95.7%)</td>
<td>7.2M (2.9%)</td>
<td>3.4M (1.4%)</td>
</tr>
<tr>
<td>StreamSPAS</td>
<td>6.2 (80% BW)</td>
<td>0.6 – 0.9</td>
<td>3.84M (57.6%)</td>
<td>1.9M (28.5%)</td>
<td>0.93M (13.9%)</td>
</tr>
<tr>
<td>Dense mat-mul</td>
<td>1.16 (90% FLOPS)</td>
<td>190</td>
<td>239.0M (93.1%)</td>
<td>17.1M (6.6%)</td>
<td>0.6M (0.2%)</td>
</tr>
<tr>
<td>StreamCDP</td>
<td>8.9</td>
<td>1.8 – 2.9</td>
<td>4.3M (58.9%)</td>
<td>1.9M (25.8%)</td>
<td>1.1M (15.3%)</td>
</tr>
</tbody>
</table>

* Dominated by divide and square-root operations
A CUDA-Enabled GPU is
The Ultimate Throughput Computer

GeForce GTX 280 / Tesla T10

- 240 scalar processors
- 30SMs x 8 each
- > 1TFLOPS peak
- Exposed, hierarchical memory
- 10-200x performance & efficiency of a latency-optimized processor
GPU Throughput Computer

GeForce GTX 280 / Tesla T10

Fixed Function Acceleration

Communication Fabric

Memory & I/O
Stream Programming
Stream Programming:
Parallelism, Locality, and Predictability

- **Parallelism**
  - Data parallelism across stream elements
  - Task parallelism across kernels
  - ILP within kernels

- **Locality**
  - Producer/consumer
  - Within kernels

- **Predictability**
  - Enables scheduling
Evolution of Stream Programming

1997 StreamC/KernelC
Break programs into kernels and streams
Kernels operate only on input/output streams and locals
Communication scheduling and stream scheduling

2002 Brook
Continues the construct of streams and kernels
Hides underlying details
Too “one-dimensional”

2007 Sequoia
Generalizes kernels to “tasks”
Tasks operate on local data
Local data “gathered” in an arbitrary way
“Inner” tasks subdivide, “leaf” tasks compute
Machine-specific details factored out
Stream Programming (Sequoia view)

- Express computation as a tree of tasks
  - Inner tasks (streams) move data to subdivide the problem
  - Leaf tasks (kernels) solve local instance of the problem

- Makes leaf tasks completely predictable
  - Enables static optimization

- Enables strategic optimization
  - Stream scheduling
  - Tuning of data movement

Diagram:
- Node memory
- Aggregate LS
- LS 0
- LS 7
- FU
Sequoia – Generalize Kernels into Leaf Tasks

- Perform actual computation
- Analogous to kernels
- “Small” working set

```c
void __task matmul::leaf(__in float A[M][P],
                       __in float B[P][N],
                       __inout float C[M][N])
{
    for (int i=0; i<M; i++) {
        for (int j=0; j<N; j++) {
            for (int k=0; k<P; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```
Inner tasks

- Decompose to smaller subtasks
  - Recursively
- “Larger” working sets

```c
void __task matmul::inner(__in float A[M][P],
                          __in float B[P][N],
                          __inout float C[M][N])
{
    tunable unsigned int U, X, V;
    blkset Ablks = rchop(A, U, X);
    blkset Bblks = rchop(B, X, V);
    blkset Cblks = rchop(C, U, V);

    mappar (int i=0 to M/U, int j=0 to N/V)
        mapreduce (int k=0 to P/X)
            matmul(Ablks[i][k],Bblks[k][j],Cblks[i][j]);
}
```
CUDA as a Stream Language

- Explicit control of the memory hierarchy with shared memory
  ```
  __shared__ float a[SIZE] ;
  ```
- Also enables communication between threads of a CTA
- Transfer data up and down the hierarchy
- Operate on data with kernels
- But does allow access to arbitrary data within a kernel
Examples

- 146X: Interactive visualization of volumetric white matter connectivity
- 36X: Ionic placement for molecular dynamics simulation on GPU
- 19X: Transcoding HD video stream to H.264
- 17X: Fluid mechanics in Matlab using .mex file CUDA function
- 100X: Astrophysics N-body simulation

- 149X: Financial simulation of LIBOR model with swaptions
- 47X: GLAM@elab: an M-script API for GPU linear algebra
- 20X: Ultrasound medical imaging for cancer diagnostics
- 24X: Highly optimized object oriented molecular dynamics
- 30X: Cmatch exact string matching to find similar proteins and gene sequences
Stream loads/stores (bulk operations) hide latency
(1000s of words in flight)
Explicit storage enables simple, efficient execution

All needed data and instructions on-chip no misses
Caches lack predictability
(controlled via a “wet noodle”)
Caches are controlled via a “wet noodle”

99% hit rate, 1 miss costs 100s of cycles, 10,000s of ops
Predictability enables efficient static scheduling

One iteration

SW Pipeline

ComputeCellInt kernel from StreamFem3D

Over 95% of peak with simple hardware

Depends on explicit communication to make delays predictable
Bulk operations enable strategic optimization

- Compute Flux
- States

- Compute Numerical Flux

- Gather Cell

- Compute Cell Interior

- Advance Cell

- Read-Only Table Lookup Data
  (Master Element)

- Element Faces

- Face Geometry

- Numerical Flux

- Cell Geometry

- Cell Orientations

- Gathered Elements

- Elements (Current)

- Elements (New)
## Results – Horizontal portability

<table>
<thead>
<tr>
<th></th>
<th>Scalar</th>
<th>SMP</th>
<th>Disk</th>
<th>Cluster</th>
<th>Cell</th>
<th>PS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAXPY</td>
<td>0.3</td>
<td>0.7</td>
<td>0.007</td>
<td>1.4</td>
<td>3.5</td>
<td>3.1</td>
</tr>
<tr>
<td>SGEMV</td>
<td>1.1</td>
<td>1.7</td>
<td>0.04</td>
<td>3.8</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>SGEMM</td>
<td>6.9</td>
<td>45</td>
<td>5.5</td>
<td>91</td>
<td>119</td>
<td>94</td>
</tr>
<tr>
<td>CONV2D</td>
<td>1.9</td>
<td>7.8</td>
<td>0.6</td>
<td>24</td>
<td>85</td>
<td>62</td>
</tr>
<tr>
<td>FFT3D</td>
<td>1.5</td>
<td>7.8</td>
<td>0.1</td>
<td>7.5</td>
<td>54</td>
<td>31*</td>
</tr>
<tr>
<td>GRAVITY</td>
<td>4.8</td>
<td>40</td>
<td>3.7</td>
<td>68</td>
<td>97</td>
<td>71</td>
</tr>
<tr>
<td>HMEER</td>
<td>0.9</td>
<td>11</td>
<td>0.9</td>
<td>12</td>
<td>12</td>
<td>7.1*</td>
</tr>
</tbody>
</table>

*Problem size reduced to fit

* Reduced dataset size to fit in memory
Resource Utilization – IBM Cell

- Bandwidth utilization
- Compute utilization

---

**Resource Utilization (%)**

- **SAXPY**: Bandwidth utilization
- **SCBMV**: Bandwidth utilization
- **FFT3D**: Bandwidth utilization
- **SCGEMM**: Bandwidth utilization
- **CONV2D**: Bandwidth utilization
- **GRAVITY**: Bandwidth utilization

---

**Resource Utilization (%)**

- **SAXPY**: Compute utilization
- **SCBMV**: Compute utilization
- **FFT3D**: Compute utilization
- **SCGEMM**: Compute utilization
- **CONV2D**: Compute utilization
- **GRAVITY**: Compute utilization
Search Spaces

Execution Time of Matrix Multiplication for Unrolling and Tiling

T. Kisuki and P. M. W. Knijnenburg and Michael F. P. O’Boyle
Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation.
### Performance of Auto-tuner

<table>
<thead>
<tr>
<th></th>
<th>Conv2D</th>
<th>SGEMM</th>
<th>FFT3D</th>
<th>SUmb</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cell</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auto</td>
<td>96.4</td>
<td>129</td>
<td>57</td>
<td>10.5</td>
</tr>
<tr>
<td>Hand</td>
<td>85</td>
<td>119</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td><strong>Cluster</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auto</td>
<td>26.7</td>
<td>91.3</td>
<td>5.5</td>
<td>1.65</td>
</tr>
<tr>
<td>Hand</td>
<td>24</td>
<td>90</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td><strong>Cluster of PS3s</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auto</td>
<td>19.5</td>
<td>32.4</td>
<td>0.55</td>
<td>0.49</td>
</tr>
<tr>
<td>Hand</td>
<td>19</td>
<td>30</td>
<td>0.23</td>
<td></td>
</tr>
</tbody>
</table>

Measured Raw Performance of Benchmarks: auto-tuner vs. hand-tuned version in GFLOPS.

For FFT3D, performances is with fusion of leaf tasks.

SUmb is too complicated to be hand-tuned.
Conclusion
The Road Ahead

- 20x improvement in GPU performance by 2015
- <2x improvement in CPU performance by 2015
  - Most of the value delivered to end user comes from the GPU, the throughput computer
- But not all things scale at the same rate
  - Memory bandwidth scales by <5x
  - Energy per bit-mm on chip nearly constant
  - Energy per op only improves by 5x
- Each of these presents an interesting challenge
Conclusion – Parallelism and Locality for efficient computation

- Denial architecture is at an end
- We are entering an era of Throughput Computing
  - Heterogeneous computers
  - Value comes from Throughput Applications running on Throughput Processors (like a GPU)
- Performance = parallelism, Efficiency = locality
  - Applications have lots of both
- Stream processing
  - Many ALUs exploit parallelism
  - Rich, exposed storage hierarchy enables locality
  - Simple control and synchronization reduces overhead
- Stream programming - explicit movement, bulk operations
  - Enables strategic optimization
  - Exposes parallelism and locality
- Result: performance and efficiency
  - TOPs on a chip
  - 20-30x efficiency of conventional processors.
  - Performance portability
- GPUs are the ultimate stream processors
Application Performance (cont.)

Execution time (%):
- Depth
- MPEG
- RTSL
- Average

Bar chart showing execution time distribution across different categories:
- Host bandwidth stalls
- Stream controller overhead
- Memory stalls
- Cluster stalls
- Kernel non main loop
- Kernel main loop overhead
- Operations