Energy and Parallelism
The Challenges of Future Computing

Bill Dally
Chief Scientist & Sr VP of Research, NVIDIA
Professor of Engineering, Stanford University
Exascale Computing Will Enable Transformational Science
Comprehensive Earth System Model at 1KM scale, enabling modeling of cloud convection and ocean eddies.
First-principles simulation of combustion for new high-efficiency, low-emission engines.
Coupled simulation of entire cells at molecular, genetic, chemical and biological levels.
Predictive calculations for thermonuclear and core-collapse supernovae, allowing confirmation of theoretical models.
Exascale Computing Will Enable Transformational Science

High-Performance Computers are Scientific Instruments
Titan: World’s #1 Open Science Supercomputer

18,688 NVIDIA Tesla K20X GPUs
27 Petaflops Peak: 90% of Performance from GPUs
17.59 Petaflops Sustained Performance on Linpack
Titan & Kepler

18,688 NVIDIA Kepler GK110
27 PF peak (90% from GPUs)
17.6PF HP Linpack
2.12 GF/W

GK110 is 7GF/W
The Road to Exascale

2012
20PF
18,000GPUs
10MW
2GFLOPs/W
~10^7 Threads

You are Here

2020
1000PF (50x)
72,000HCNs (4x)
20MW (2x)
50GFLOPs/W (25x)
~10^{10} Threads (1000x)
Technical Challenges on The Road to Exascale

1. Energy Efficiency

- 2012
  - 20PF
  - 18,000GPUs
  - 10MW
  - 2GFLOPs/W
  - $\sim 10^7$ Threads

- 2020
  - 1000PF (50x)
  - 72,000HCNs (4x)
  - 20MW (2x)
  - 50GFLOPs/W (25x)
  - $\sim 10^{10}$ Threads (1000x)
Technical Challenges on The Road to Exascale

1. Energy Efficiency
2. Parallel Programmability

2012
20PF
18,000GPUs
10MW
2GFLOPs/W
~10^7 Threads

2020
1000PF (50x)
72,000HCNs (4x)
20MW (2x)
50GFLOPs/W (25x)
~10^{10} Threads (1000x)
Technical Challenges on The Road to Exascale

1. Energy Efficiency
2. Parallel Programmability
3. Resilience

2012
- 20PF
- 18,000GPUs
- 10MW
- 2GFLOPs/W
- $\sim 10^7$ Threads

2020
- 1000PF (50x)
- 72,000HCNs (4x)
- 20MW (2x)
- 50GFLOPs/W (25x)
- $\sim 10^{10}$ Threads (1000x)
50x performance in 8 years, Moore’s Law will take care of that, right?
50x performance in 8 years, Moore’s Law will take care of that, right?

Wrong!
Moore’s Law gives us transistors
Which we used to turn into scalar performance

Moore, Electronics 38(8) April 19, 1965
But ILP was ‘mined out’ in 2000

And $L^3$ energy scaling ended in 2005.
Result: The End of Historic Scaling

C. Moore, *Data Processing in ExaScale-Class Computer Systems*, Salishan, April 2011
Historic scaling is at an end!

To continue performance scaling of all sizes of computer systems requires addressing two challenges:

Power and Parallelism

Much of the economy depends on this
The Power Challenge
In the past we had constant-field scaling

\[ L' = \frac{L}{2} \]
\[ V' = \frac{V}{2} \]
\[ E' = CV^2 = \frac{E}{8} \]
\[ f' = 2f \]
\[ D' = \frac{1}{L^2} = 4D \]
\[ P' = P \]

Halve L and get 8x the capability for the same power
Now voltage is held nearly constant

\[ L' = \frac{L}{2} \]
\[ V' = V \]
\[ E' = CV^2 = \frac{E}{2} \]
\[ f' = 2f^* \]
\[ D' = \frac{1}{L^2} = 4D \]
\[ P' = 4P \]

Halve \( L \) and get 2x the capability for the same power in \( \frac{1}{4} \) the area

*\( f \) is no longer scaling as \( 1/L \), but it doesn’t matter, we couldn’t power it if it did
Performance = Efficiency

Efficiency = Locality
Locality
The High Cost of Data Movement

Fetching operands costs more than computing on them

- 64-bit DP: 20 pJ
- 256-bit buses
- 256-bit access: 8 kB SRAM
- DRAM Rd/Wr: 16 nJ
- Efficient off-chip link: 500 pJ
- 26 pJ
- 256 pJ
- 1 nJ
- 50 pJ
Scaling makes locality even more important

<table>
<thead>
<tr>
<th>Process technology</th>
<th>2010</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdd (nominal)</td>
<td>0.9 V</td>
<td>0.75 V</td>
</tr>
<tr>
<td>Frequency Target</td>
<td>1.6 GHz</td>
<td>2.5 GHz</td>
</tr>
<tr>
<td>DFMA energy</td>
<td>50 pJ</td>
<td>8.7 pJ</td>
</tr>
<tr>
<td>64-bit read from 8KB SRAM</td>
<td>14 pJ</td>
<td>2.4 pJ</td>
</tr>
<tr>
<td>Wire energy (per transition)</td>
<td>240 fJ/bit/mm</td>
<td>150 fJ/bit/mm</td>
</tr>
<tr>
<td>Wire energy (256 bits, 10mm)</td>
<td>310 pJ</td>
<td>200 pJ</td>
</tr>
<tr>
<td>DRAM process technology</td>
<td>45 nm</td>
<td>16 nm</td>
</tr>
<tr>
<td>DRAM Interface Pin Bandwidth</td>
<td>4 Gbps</td>
<td>50 Gbps</td>
</tr>
</tbody>
</table>

Table 1: Technology and circuit projections.
Its not about the FLOPS

Its about data movement

Algorithms should be designed to perform more work per unit data movement.

Programming systems should further optimize this data movement.

Architectures should facilitate this by providing an exposed hierarchy and efficient communication.
Move Bits More Efficiently

![Graph showing energy per bit per mm (fJ) vs maximum frequency (GHz)](image)

- **FSI**
- **LSI (200 mV)**
- **LSI (400 mV)**
- **CDI**
- **SCI**
forall cells in set {
    compute_x_flux(cell) ;
}
forall cells in set {
    compute_y_flux(cell) ;
}
forall cells in set {
    compute_z_flux(cell) ;
}
forall cells in set {
    compute_p(cell) ;
}
forall cells in set {
    compute_x_flux(cell) ;
    compute_y_flux(cell) ;
    compute_z_flux(cell) ;
    compute_p(cell) ;
}
forall blocks in set {// hierarchically
    localize(block)
    forall cells in block {
        compute_x_flux(cell);
        compute_y_flux(cell);
        compute_z_flux(cell);
        compute_p(cell);
    }
}
System Sketch
Echelon Chip Floorplan

NOC

L2 Banks

XBAR

17mm

10nm process

290mm$^2$
An Out-of-Order Core

Spends 2nJ to schedule a 25pJ FMUL (or an 0.5pJ integer add)
SM Lane Architecture

Control Path

Thread PCs

Active PCs

Scheduler

L0 I$

Inst

64 threads
4 active threads
2 DFMAs (4 FLOPS/clock)
ORF bank: 16 entries (128 Bytes)
L0 I$: 64 instructions (1KByte)
LM Bank: 8KB (32KB total)

Data Path

RF
L0Addr
L1Addr
Net

LM Bank 0

RF
L0Addr
L1Addr
Net

LM Bank 3

To LD/ST

ORF

FP/Int

ORF

FP/Int

ORF

LS/BR

Net

To LD/ST

Thread PCs

Active PCs

Scheduler

L0 I$

Inst

64 threads
4 active threads
2 DFMAs (4 FLOPS/clock)
ORF bank: 16 entries (128 Bytes)
L0 I$: 64 instructions (1KByte)
LM Bank: 8KB (32KB total)
Solving the Power Challenge – 1, 2, 3
Solving the ExaScale Power Problem
Parallelism
Parallel programming is not inherently any more difficult than serial programming

However, we can make it a lot more difficult
forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { // nested
        forall force in forces { // doubly nested
            molecule.force =
                reduce_sum(force(molecule, neighbor))
        }
    }
}
Why is this easy?

forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { // nested
        forall force in forces { // doubly nested
            molecule.force =
                reduce_sum(force(molecule, neighbor))
        }
    }
}

No machine details
All parallelism is expressed
Synchronization is semantic (in reduction)
We could make it hard

```c
pid = fork(); // explicitly managing threads

lock(struct.lock); // complicated, error-prone synchronization
// manipulate struct
unlock(struct.lock);

code = send(pid, tag, &msg); // partition across nodes
```
Programmers, tools, and architecture
Need to play their positions

Programmer

Tools

Architecture

Celsius Lecture
2/14/13

45
Programmers, tools, and architecture
Need to play their positions

Programmer

Algorithm
All of the parallelism
Abstract locality

Tools

Combinatorial optimization
Mapping
Selection of mechanisms

Architecture

Fast mechanisms
Exposed costs
Programmers, tools, and architecture
Need to play their positions

forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { //
        forall force in forces { // doubly nested
            molecule.force = reduce_sum(force(molecule, neighbor))
        }
    }
}

Map foralls in time and space
Map molecules across memories
Stage data up/down hierarchy
Select mechanisms

Exposed storage hierarchy
Fast comm/sync/thread mechanisms
Abstract description of Locality – not mapping

```cpp
compute_forces::inner(molecules, forces) {
    tunable N ;
    set part_molecules[N] ;
    part_molecules = subdivide(molecules, N) ;

    forall(i in 0:N-1) {
        compute_forces(part_molecules[i]) ;
    }
}
```
Abstract description of **Locality** – not mapping

```c
compute_forces::inner(molecules, forces) {
    tunable N;
    set part_molecules[N];
    part_molecules = subdivide(molecules, N);

    forall(i in 0:N-1) {
        compute_forces(part_molecules);
    }
}
```

**Autotuner picks number and size of partitions - recursively**

**No need to worry about “ghost molecules” with global address space, it just works**
Autotuning Search Spaces

Architecture enables simple and effective autotuning

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.
Fundamental and Incidental Obstacles to Programmability

**Fundamental**
- Expressing $10^9$ way parallelism
- Expressing locality to deal with $>100:1$ global:local energy
- Balancing load across $10^9$ cores

**Incidental**
- Dealing with multiple address spaces
- Partitioning data across nodes
- Aggregating data to amortize message overhead
The fundamental problems are hard enough. We must eliminate the incidental ones.
Execution Model

Global Address Space

Object Thread

Abstract Memory Hierarchy

Active Message

Load/Store

Bulk Xfer
Thread array creation, messages, block transfers, collective operations – at the “speed of light”
Hardware thread-array creation
Fast syncthreads();
Shared memory
Scalar ISAs don’t matter

Parallel ISAs – the mechanisms for threads, communication, and synchronization make a huge difference.
A Prescription
Research

- Need a research vehicle (experimental system)
  - Co-design architecture, programming system, applications
- Productive parallel programming
  - Express all the parallelism and locality
  - Compiler and run-time map to the target machine
  - Leverage an existing eco-system
- Mechanisms – for: threads, comm, sync
  - Eliminate ‘incidental’ programming issues
  - Enable fine-grain execution
- Power
  - Locality – exposed memory hierarchy and software to use it
  - Overhead – move scheduling to compiler
- Others are investing, if we don’t invest we will be left behind.
We need parallel programmers
- But we are training serial programmers
- and serial thinkers

Parallelism throughout the CS curriculum
- Programming
- Algorithms
  - Parallel algorithms
  - Analysis focused on communications, not counting ops
- Systems

Models need to include locality
A Bright Future from Supercomputers to Cellphones

Eliminate overhead and exploit locality to get 100x power efficiency

Easy parallelism with a coordinated team

Programmer Tools
Architecture
More Fundamentally

Both

are *power limited*

get performance from *parallelism*

need *100x performance* increase in 10 years
More Fundamentally

Both are *power limited*

get performance from *parallelism*

need *100x performance* increase in 10 years
Granularity
#Threads increasing faster than problem size.
Number of Threads increasing faster than problem size

The graph shows the growth of threads and bytes from 1995 to 2020. The number of threads is increasing exponentially faster than the amount of bytes.
Number of Threads increasing faster than problem size
Number of Threads increasing faster than problem size

- Weak Scaling
- Strong Scaling

Graph showing the increase in threads and bytes from 1995 to 2020.
Smaller sub-problem per thread
Smaller sub-problem per thread
Smaller sub-problem per thread

More frequent comm, sync, and thread operations
Smaller sub-problem per thread

More frequent comm, sync, and thread operations
This fine-grain parallelism is multi-level and irregular
To support this requires fast mechanisms for

- **Thread arrays** – create, terminate, suspend, resume
  - Hardware allocation of resources to a thread array
    - threads, registers, shared memory
    - With locality
- **Communication**
  - Data movement up and down the hierarchy
  - Fast active messages (message-driven computing)
- **Synchronization**
  - Collective operations (e.g., barrier, reduce)
  - Pairwise (producer-consumer)
Execution Model

- Global Address Space
- Abstract Memory Hierarchy
- Load/Store
- Bulk Xfer
- Active Message

Object Thread
J-Machine Speedup with Strong Scaling

J-Machine Speedup with Strong Scaling