Productive Software and The Path to ExaScale

Bill Dally | Chief Scientist and SVP, Research NVIDIA | Professor (Research), EE&CS, Stanford
18,688 NVIDIA Tesla K20X GPUs
27 Petaflops Peak: 90% of Performance from GPUs
17.59 Petaflops Sustained Performance on Linpack
2013

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

2020

1,000PF (50x)
72,000 HCNs (4x)
20MW (2x)
50 GFLOPs/W (25x)
~10^{10} Threads (1000x)
Long-Term Vision: Compute Node & System

Efficiency:
- Malleable memory hierarchy
- Hierarchical register files
- Hierarchical thread scheduling
- Place coherency/consistency
- Temporal SIMT & scalarization

Programmability:
- Global address space (PGAS)
- AMOs and collectives
- Efficient work creation
- Fast synchronization
- Streamlined LOC/TOC interaction
- Active messages
Programming Systems

Our Goals

High-level performance portability across diverse machines. With tools to automate/assist mapping to...

Foundations permitting experts to write the code they want and others to build upon that cleanly.

Runtime environment that provides a unified memory model to the application.

### Application

<table>
<thead>
<tr>
<th>Collection-oriented programming framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide higher-level abstractions for performance portability.</td>
</tr>
<tr>
<td>reduce(combine, map(update, split(x, size)))</td>
</tr>
</tbody>
</table>

### Core programming model

Provide foundation for managing heterogeneity, parallelism, & locality.

```c
auto where = placementof(ptr);
int *more = allocate<int>(where, n);
wait(async(where, n)(some_function, x, more));
```

### Runtime environment

Provide system-level support for unified virtual memory.

```c
setup_array(A);
launch_kernel(A, x, y);
inspect_results(A);
```
Overhead Dominates Power in CPUs

In-order Embedded
- Instruction Supply: 42%
- Instruction Supply: 42%
- Clock + Control Logic: 24%
- Data Supply: 17%
- ALU: 6%
- Register File: 11%

OOOO Hi-perf
- Clock + Pins: 45%
- Fetch: 11%
- Rename: 10%
- Issue: 11%
- RF: 14%
- ALU: 4%
- Data Supply: 5%

Dally [2008] (Embedded in-order CPU)
Natarajan [2003] (Alpha 21264)
Much of the Rest is Communication

- 64-bit DP: 20pJ
- 256-bit buses
- 256-bit access: 8 kB SRAM: 50 pJ
- 256-bit access: 8 kB SRAM: 50 pJ
- DRAM Rd/Wr: 500 pJ
- Efficient off-chip link: 1 nJ
for all molecule in set { // launch a thread array
    for all neighbor in molecule.neighbors { // nested
        for all 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 driven by data dependencies
Communication is captured in the reduction
We could make it hard

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
Programmers, Tools, and Architecture Need to Play Their Positions

Programmer

Tools

Architecture

Algorithm
All of the parallelism
Abstract locality

Combinatorial optimization
Mapping
Selection of mechanisms

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
Target-Independent Source → Mapping Tools → Target-Dependent Adaptation → Compile → Target-Dependent Executable

Profiling & Visualization → Mapping Directives
We need Power Tools for Parallel Programming. Conventional Programming Systems are Hand Tools.
Power
25x Efficiency with 2.2x from process

Programming
Parallelism
Heterogeneity
Hierarchy

![Graph showing the trend of power and efficiency over years]

- **Needed**: Green line indicating the required power trend.
- **Process**: Blue line showing the process efficiency trend.

- **Architecture** increase by 4x from 2013 to 2020.
- **Circuits** increase by 3x from 2013 to 2020.
- **Process** increase by 2.2x from 2013 to 2020.

- **Tools** connected to **Programmer** and **Architecture**.
- **Programmer** connected to **Tools** and **Architecture**.
“Super” Computing
From Super Computers to Super Phones