The Future of Computing: Domain-Specific Architecture

Chesapeake Large Scale Analytics Conference
October 5, 2021

Bill Dally
Chief Scientist and SVP of Research, NVIDIA Corporation
Adjunct Professor of EE and CS, Stanford University
Domain-Specific Hardware Accelerators

By William J. Dally, Yatish Turakhia, Song Han
Communications of the ACM, July 2020, Vol. 63 No. 7, Pages 48-57
10.1145/3381682

From the simple embedded processor in your washing machine to powerful processors in data center servers, most computing today takes place on general-purpose programmable processors or CPUs. CPUs are attractive because they are easy to program and because large code bases exist for them. The programmability of CPUs stems from their execution of sequences of simple instructions, such as ADD or BRANCH; however, the energy required to fetch and interpret an instruction is ten to a hundred more than that required to perform a simple operation such as ADD. This high overhead was acceptable when processor performance and efficiency were scaling according to Moore's Law. One could simply wait and an existing application would run faster and more efficiently. Our economy has become dependent on these increases in computing performance and efficiency to enable new features and new applications. Today, Moore's Law has largely ended, and we must look to alternative architectures with lower overhead, such as domain-specific accelerators, to continue scaling of performance and efficiency. There are several ways to realize domain-specific accelerators as discussed in the sidebar on accelerator options.

Credit: Matt Herring, Bet_Noire / Getty Images

Key Insights
- Most speedups come from parallelism
- Accelerator design and implementation drivers

Article Contents:
- Introduction
- Key Insights
- Sources of Acceleration
- Codeign in Needed
- Memory Dominates
- Accelerators
- Balancing Specialization and General
- Total Cost of Ownership (TCO)
- Accelerator Design
- Conclusion
- References
- Authors
- Footnotes
- Sidebar: Acceleration Options

MORE NEWS & OPINIONS
- Japan Post Delivery Robot Debuts in Tokyo
- Al Authorship?
Faster Computing

Better Algorithms

More Data

Value
We need to continue delivering improved performance and perf/W
But Process Technology isn’t Helping us Anymore

Moore’s Law is Dead
Accelerators can continue scaling perf and perf/W
Fast Accelerators since 1985


• **EIE**: Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A. and Dally, W.J., 2016, June. EIE: efficient inference engine on compressed deep neural network, ISCA 2016


• **Darwin**: Turakhia, Bejerano, and Dally, “Darwin: A Genomics Co-processor provides up to 15,000 × acceleration on long read assembly”, ASPLOS 2018.

• **SATiN**: Zhuo, Rucker, Wang, and Dally, “Hardware for Boolean Satisfiability Inference,” Under Review.
Accelerators Employ:

• Special **Data Types and Operations**
  • Do in 1 cycle what normally takes 10s or 100s – **10-1000x efficiency gain**

• Massive **Parallelism** – >1,000x, not 16x – with **Locality**
  • This gives performance, not efficiency

• Optimized **Memory**
  • High bandwidth (*and low energy*) for specific data structures and operations

• Reduced or Amortized **Overhead**
  • **10,000x efficiency gain** for simple operations

• Algorithm-Architecture **Co-Design**
Specialized Operations
Orders of Magnitude Efficiency
Moderate Speedup
Dynamic programming for gene sequence alignment (Smith-Waterman)

\[
I(i, j) = \max \{H(i, j - 1) - o, I(i, j - 1) - e\}
\]
\[
D(i, j) = \max \{H(i - 1, j) - o, D(i - 1, j) - e\}
\]
\[
H(i, j) = \max \begin{cases} 0 \\ I(i, j) \\ D(i, j) \\ H(i - 1, j - 1) + W(r_i, q_j) \end{cases}
\]

On 14nm CPU
- 35 ALU ops, 15 load/store
- 37 cycles
- 81nJ

On 40nm Special Unit
- 1 cycle (37x speedup)
- 3.1pJ (26,000x efficiency)
- 300fJ for logic (270,000x efficiency)
Why is a Specialized PE 26,000x More Efficient?

Area is proportional to energy – all 28nm

16b Int Add, 32fJ


Specialization -> Efficiency
Efficiency -> Parallelization
Parallelization -> Speedup
\[
I(i, j) = \max \{ H(i, j - 1) - o, I(i, j - 1) - e \}
\]
\[
D(i, j) = \max \{ H(i - 1, j) - o, D(i - 1, j) - e \}
\]
\[
H(i, j) = \max \begin{cases} 
0 & \\
I(i, j) & \\
D(i, j) & \\
H(i - 1, j - 1) + W(r_i, q_j) & 
\end{cases}
\]

**Specialization**  
37x speedup  26,000x efficiency

**Efficiency**  
Parallelism 64 PE arrays x 64 PEs per array, 4,096x total

**Speedup**  
37 (Specialization)  4,034 (Parallelism)  150,000x total
\[ I(i, j) = \max \{ H(i, j - 1) - o, I(i, j - 1) - e \} \]
\[ D(i, j) = \max \{ H(i - 1, j) - o, D(i - 1, j) - e \} \]

\[ H(i, j) = \max \begin{cases} 
0 \\
I(i, j) \\
D(i, j) \\
H(i - 1, j - 1) + W(r_i, q_j) 
\end{cases} \]

**Specialization**  37x speedup  26,000x efficiency

**Efficiency**  Parallelism 64 PE arrays x 64 PEs per array, 4,096x total

**Speedup**  37 (Specialization)  4,034 (Parallelism)  150,000x total
The Algorithm Often Has to Change
To Avoid Being Global Memory Limited
Algorithm-Architecture Co-Design for Darwin
Start with Graphmap

Graphmap

- ~10K seeds
- ~440M hits

Filtration
- ~3 hits

Alignment
- ~1 hits

1. Graphmap (software)
Algorithm-Architecture Co-Design for Darwin
Replace Graphmap with Hardware-Friendly Algorithms
Speed up Filtering by 100x, but 2.1x Slowdown Overall

1. Graphmap (software)
2. Replace by D-SOFT and GACT (software)

Graphmap
~10K seeds
~440M hits

Filtration
~3 hits

Alignment
~1 hits

Darwin
~2K seeds
~1M hits

Filtration (D-SOFT)
~1680 hits

Alignment (GACT)
~1 hits
Algorithm-Hardware Co-Design for Darwin Accelerate Alignment – 380x Speedup

1. Graphmap (software)
2. Replace by D-SOFT and GACT (software)
3. GACT hardware-acceleration
Algorithm-Hardware Co-Design for Darwin
4x Memory Parallelism – 3.9x Speedup

1. Graphmap (software)
2. Replace by D-SOFT and GACT (software)
3. GACT hardware-acceleration
4. Four DRAM channels for D-SOFT
Algorithm-Hardware Co-Design for Darwin
Specialized Memory for D-Soft Bin Updates – 15.6x Speedup

1. Graphmap (software)
2. Replace by D-SOFT and GACT (software)
3. GACT hardware-acceleration
4. Four DRAM channels for D-SOFT
5. Move bin updates in D-SOFT to SRAM (ASIC)

![Graph showing speedup and time/read in ms]

**Time/read (ms)**

- **Filtration**
- **Alignment**

- **0.1**
- **1**
- **10**
- **100**
- **1000**
- **10000**
- **100000**

- **Time**
- **Count**

- **1.0** 2.1X slowdown
- **2.0** 380X speedup
- **3.0** 3.9X speedup
- **4.0** 15.6X speedup
- **5.0**
Algorithm-Hardware Co-Design for Darwin
Pipeline D-Soft and GACT – now completely D-Soft limited – 1.4x
Overall 15,000x

1. Graphmap (software)
2. Replace by D-SOFT and GACT (software)
3. GACT hardware-acceleration
4. Four DRAM channels for D-SOFT
5. Move bin updates in D-SOFT to SRAM (ASIC)
6. Pipeline D-SOFT and GACT
Memory Dominates
Memory dominates power and area
<table>
<thead>
<tr>
<th></th>
<th>Unit</th>
<th>Area (mm²)</th>
<th>(%)</th>
<th>Power (W)</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GACT</td>
<td>Logic</td>
<td>17.6</td>
<td>20.5%</td>
<td>1.04</td>
<td>23.6%</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>68.0</td>
<td>79.5%</td>
<td>3.36</td>
<td>76.4%</td>
</tr>
<tr>
<td>D-SOFT</td>
<td>Logic</td>
<td>6.2</td>
<td>1.8%</td>
<td>0.41</td>
<td>4.4%</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>320.3</td>
<td>98.2%</td>
<td>8.80</td>
<td>95.6%</td>
</tr>
<tr>
<td>EIE</td>
<td>Logic</td>
<td>2.8</td>
<td>6.9%</td>
<td>0.23</td>
<td>40.3%</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>38.0</td>
<td>93.1%</td>
<td>0.34</td>
<td>59.7%</td>
</tr>
</tbody>
</table>
Communication is Expensive, Be Small, Be Local

LPDDR DRAM
GB

On-Chip SRAM
MB

Local SRAM
KB

640pJ/word
50pJ/word
5pJ/word
Small, Local Memories

Traceback RAM
2KB, 4b wide
2.8pJ/access
vs nJ to DRAM
MAGNet System

Global Controller

Global Buffer

DRAM

MAGNet System

AXI-In

Weight Buffer

Input Buffer

AXISlave

Address Generator

Buffer Manager

Address Generator

Buffer Manager

Vector MAC

Arbitrated Xbar

Pooling

ReLU

Bias Addition

Scaling

Rounding

PPU

ControlOut

CrossPE-AccumOut

OutputActivation

Processing Element (PE)

Weights

InputActivation

CrossPE-AccumOut

VectorSize

Collector

Accumulation Collector

Vector MAC unit

Algorithms must be memory optimized

Minimize global memory accesses

Keep local memory footprint small
GACT Alignment

- 15M Reads, 10k bases each, ~2k hits each
  - ~300T Alignments to be done
  - Additional parallelism within each alignment
- But long reads have large (10M) memory footprint
- Solution: GACT (Tiling)
Complex Memory Ops
Not just Load/Store
Hash, Atomic Functions, Side Effects
Make the Most Use of One Communication
One traversal of network:

- Access hash table
- Increment bin (RMW)
- If it was zero, append to NZ bins
- If over threshold, append to output queue
Sparsity and Compression multiply Memory Bandwidth and Capacity
EIE HARDWARE

- Traverse CSC Sparse matrix
- Decode scalar quantization
- Little overhead

Platforms for Acceleration
GPUs Provide:

• High-Bandwidth, Hierarchical **Memory** System
  • Can be configured to match application

• Programmable **Control** and **Operand Delivery**

• Simple places to bolt on **Domain-Specific Hardware**
  • As instructions or memory clients
## Specialized Instructions Amortize Overhead

*Overhead is instruction fetch, decode, and operand fetch – 30pJ

**Energy numbers from 45nm process

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ops</th>
<th>Energy**</th>
<th>Overhead*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vs op</td>
</tr>
<tr>
<td>HFMA</td>
<td>2</td>
<td>1.5pJ</td>
<td>20x</td>
</tr>
<tr>
<td>HDP4A</td>
<td>8</td>
<td>6.0pJ</td>
<td>5x</td>
</tr>
<tr>
<td>HMMA</td>
<td>128</td>
<td>130pJ</td>
<td>0.23x</td>
</tr>
<tr>
<td>IMMA</td>
<td>1024</td>
<td>230pJ</td>
<td>0.13x</td>
</tr>
</tbody>
</table>
Program

(map force
(pairs
particles))

Mapping Directives

Mapper & Runtime

Synthesis

Data & Task Placement

Custom Compute Blocks
(Instructions or Clients)

SMs

Configurable Memory

Efficient NoC

GPU
Implementation Alternatives

- GDDR6
- LPDDR4
- DPSTEP

Bar chart comparing performance of different technologies in Deep Learning and Genomics.
Accelerator Design as Programming
With Hardware Costs
**Algorithm**

```plaintext
Algorithm

```

**Mapping**

<table>
<thead>
<tr>
<th>STRIPES ← TS / AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor_array p (AS)</td>
</tr>
<tr>
<td>memory_array t bm (AS)[ST RIPES, TS ]</td>
</tr>
<tr>
<td>Map h (i,j) → p (i % AS)</td>
</tr>
<tr>
<td>at t = (i % AS) · TS + j - i / AS</td>
</tr>
<tr>
<td>Map tb [i,j] → t bm (i % AS) [i / AS, j]</td>
</tr>
</tbody>
</table>
```
Hardware Costs

Global Memory 640pJ/word
On Chip Comm 3.2pJ/word-mm
Local Memory 1.6pJ/word
Arithmetic
- 5pJ (FP64 FMA)
- 1.2pJ (FP32 FMA)
- 260fJ (int16 mul)
- 10fJ (int8 add)

*Energy numbers for 14nm
**A word is 32 bits
Accelerating Deep Learning
NVIDIA DLA

Command Interface

Tensor Execution Micro-controller

Memory Interface

Input DMA (Activations and Weights) — Unified 512KB Input Buffer — Activations and Weights

Sparse Weight Decompression

Native Winograd Input Transform

MAC Array
- 2048 Int8
- 1024 Int16
- 1024 FP16

Output Accumulators

Output Postprocessor (Activation Function, Pooling etc.)

Output DMA
EIE Hardware

- Traverse CSC Sparse matrix
- Decode scalar quantization
- Little overhead

Sparse Convolution Engine

Sparse Weight Buffer
Sparse Input Buffer

Output Addr Computation

MxW multiplier array

Banked Output Buffer
Scatter-Add Unit

RC18: A 36-die MCM Architecture

Connected via Ground-Referencing Signaling (GRS)

- GRS: 4 data bumps + 1 clock bump, 25Gbps/pin, 1.6pJ/bit*.
- 8 GRS links per die connected in mesh (NESW TX/RX).
- 100GB/s per chiplet.
- 105fJ/op

*1.2pJ/bit at 8b width, 1pJ/bit at 16b width
RC18: Partitioning of Weights and Activations

Scaling DL Inference across NoP and NoC
MAGNET

DataFlow Options

Temporal weight reuse

Weight Stationary (WS)

Input Buffer

Weight Collector

Weight Buffer

Vector MAC

Vector MAC

Vector MAC

Accum. Buffer

Temporal partial sum reuse

Output Stationary (OS)

Input Buffer

Vector MAC

Vector MAC

Vector MAC

Accum. Collector

Accum. Buffer

Energy consumption

Less-frequent access

More-frequent access

Weight Stationary

Input Buffer

Weight Buffer

Accumulation Buffer

Datapath

29%

11%

5%

55%

Output Stationary

Input Buffer

Weight Buffer

Accumulation Buffer

Datapath

29%

7%

54%

10%
Multi-Level DataFlows

Weight Stationary (WS)
- Reduce accum. buffer accesses
- Temporal weight reuse
- Temporal partial sum reuse
- Reduce weight buffer accesses

Output Stationary (OS)

Weight Stationary – Local Weight Stationary (OS-LWS)
- Less-frequent access
- More-frequent access
Multi-Level DataFlows

VectorSize=16, IAPrecision=8, WPrecision=8

70 fJ/MAC
35 fJ/OP
29 TOPS/W
Conclusion
Summary

• Moore’s Law is over, but we must continue scaling perf/W.

• Accelerators are the future
  – Specialization, Customized Memories -> Efficiency
  – Parallelism -> Speedup
  – Co-Design: The algorithm has to change
  – Memory dominates

• GPUs as accelerator platforms
  – GPUs – efficient memory, communication and control
  – Custom blocks – instructions or clients

• DSA design is programming – with a hardware cost model