Improving Performance of Machine Learning Workloads

Dong Li

Parallel Architecture, System, and Algorithm Lab

Electrical Engineering and Computer Science

School of Engineering

University of California, Merced

Outline

- Running machine learning workloads on CPU
- Running machine learning workloads on GPU
- Recent new hardware to run machine learning workloads
  - Google TPU
  - NVIDIA Volta

Running Machine Learning Workloads on CPU

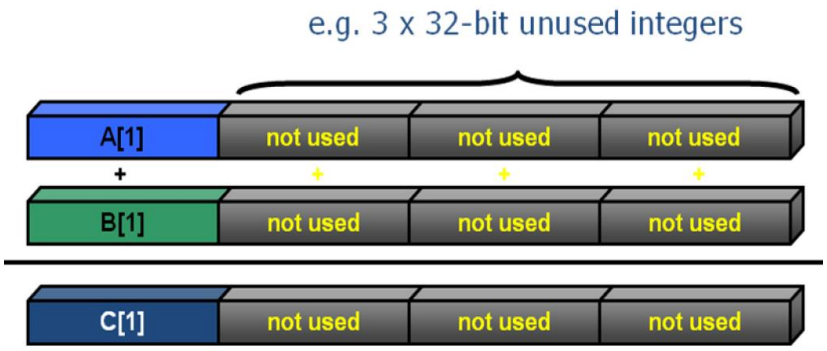
Running ML on CPU: performance improvement tips

- Code vectorization for SIMD vector instructions
  - Ensure that all the key primitives, such as convolution, matrix multiplication, and batch normalization are vectorized to the latest SIMD instructions
- NUMA performance problem
  - Improve data locality and avoid data access bottleneck
- Manage thread-level parallelism
  - Improve system throughput

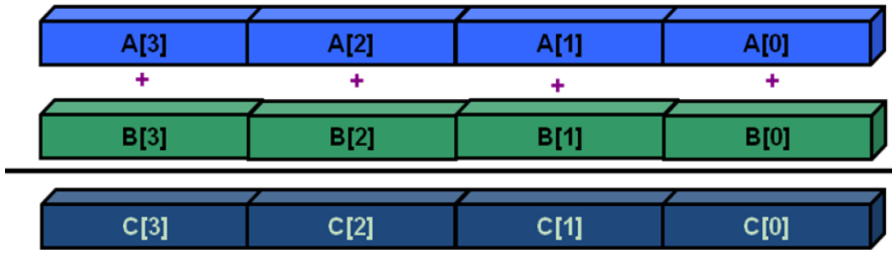
SIMD (Single Instruction Multiple Data) vector instructions in a nutshell

- What are these instructions?
  - Extension of the ISA → Data types and instructions for parallel computation on short (2-16) vectors of integers and floats
  - One operation produces multiple results
- An example

```
for(i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];    //a, b and c are integer arrays
```



Vectorization not enabled



Vectorization enabled

Evolution of Intel vector instructions

- MMX: Multimedia Extensions (1996)
- SSE: Streaming SIMD Extensions
- AVX: Advanced Vector Extension (2010)
  - Intel Xeon Phi (co-processor): 512 bit wide SIMD vector

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- SIMD allows programmer to continue to think sequentially

Obstacles to vectorization

- Non-contiguous Memory Accesses

```
// arrays accessed with stride 2
   for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

// inner loop accesses a with stride SIZE
   for (int j=0; j<SIZE; j++)  {
      for (int i=0; i<SIZE; i++)   b[i] += a[i][j] * x[j];
 }

// indirect addressing of x using index array
   for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[index[i]];
```

Obstacles to vectorization

- Data dependencies

Read after write

```
A[0]=0;

for (j=1; j<MAX; j++)  A[j]=A[j-1]+1;

// this is equivalent to:

A[1]=A[0]+1;   A[2]=A[1]+1;   A[3]=A[2]+1; A[4]=A[3]+1;
```

The above loop cannot be vectorized

Write after read

```
for (j=1; j<MAX; j++)  A[j-1]=A[j]+1;

// this is equivalent to:

A[0]=A[1]+1;   A[1]=A[2]+1;   A[2]=A[3]+1; A[3]=A[4]+1;
```

The above loop can be vectorized

Obstacles to vectorization

- The loop trip count is not known at entry to the loop at runtime

- Data-dependent exit condition

```
void no_vec(float a[], float b[], float c[])
{
      int i = 0.;
      while (i < 100) {
        a[i] = b[i] * c[i];
//    this is a data-dependent exit condition:
        if (a[i] < 0.0)
           break;
        ++i;
      }
}
```

Obstacles to vectorization

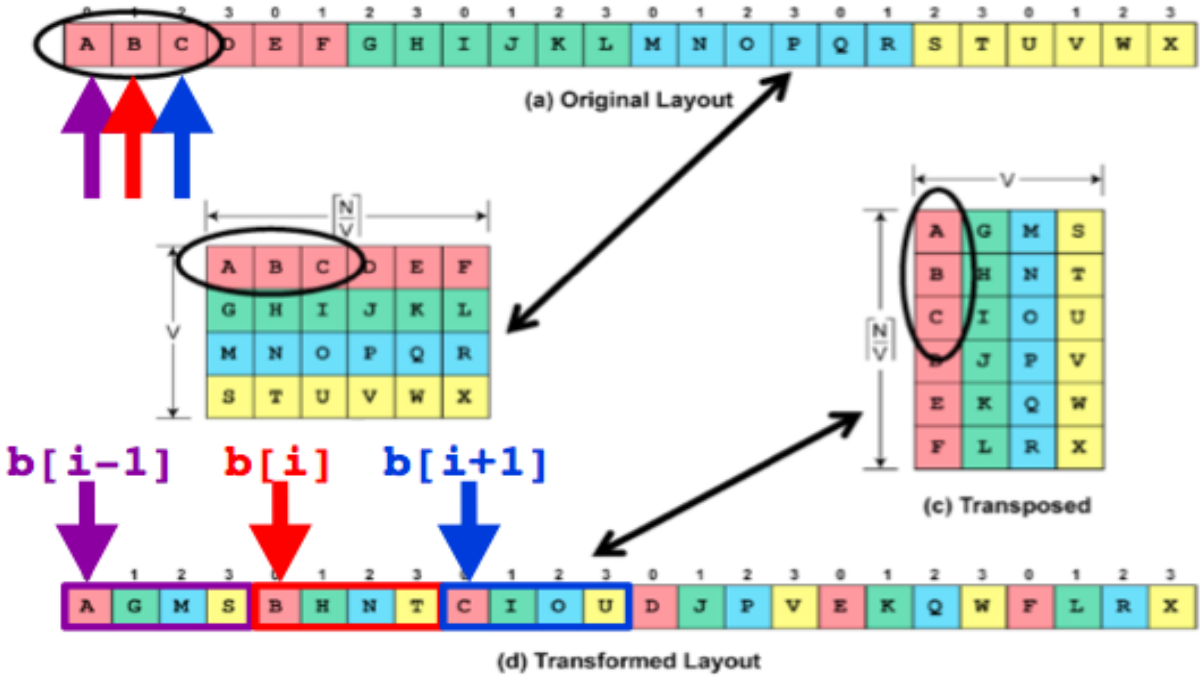
- Not straight-line code
  - There is control flow

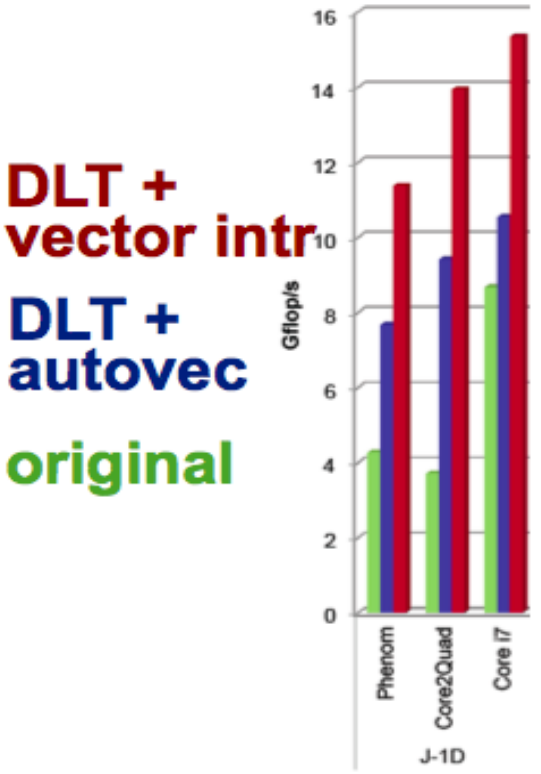
```
#include <math.h>
void quad(int length, float *a, float *b,
          float *c, float *restrict x1, float *restrict x2)
   {
     for (int i=0; i<length; i++) {
       float s = b[i]*b[i] - 4*a[i]*c[i];
       if ( s >= 0 ) {
          s = sqrt(s) ;
          x2[i] = (-b[i]+s)/(2.*a[i]);
          x1[i] = (-b[i]-s)/(2.*a[i]);
       }
       else {
          x2[i] = 0.;
          x1[i] = 0.;
       }
     }
   }
```

Code transformation to enable vectorization

- An example: dimension-lifted Transformation (DLT)

$$a[i] = b[i-1] + b[i] + b[i+1]$$





NUMA performance problem

- NUMA = **Non-uniform memory access**

FIGURE 1: **A MODERN NUMA SYSTEM**

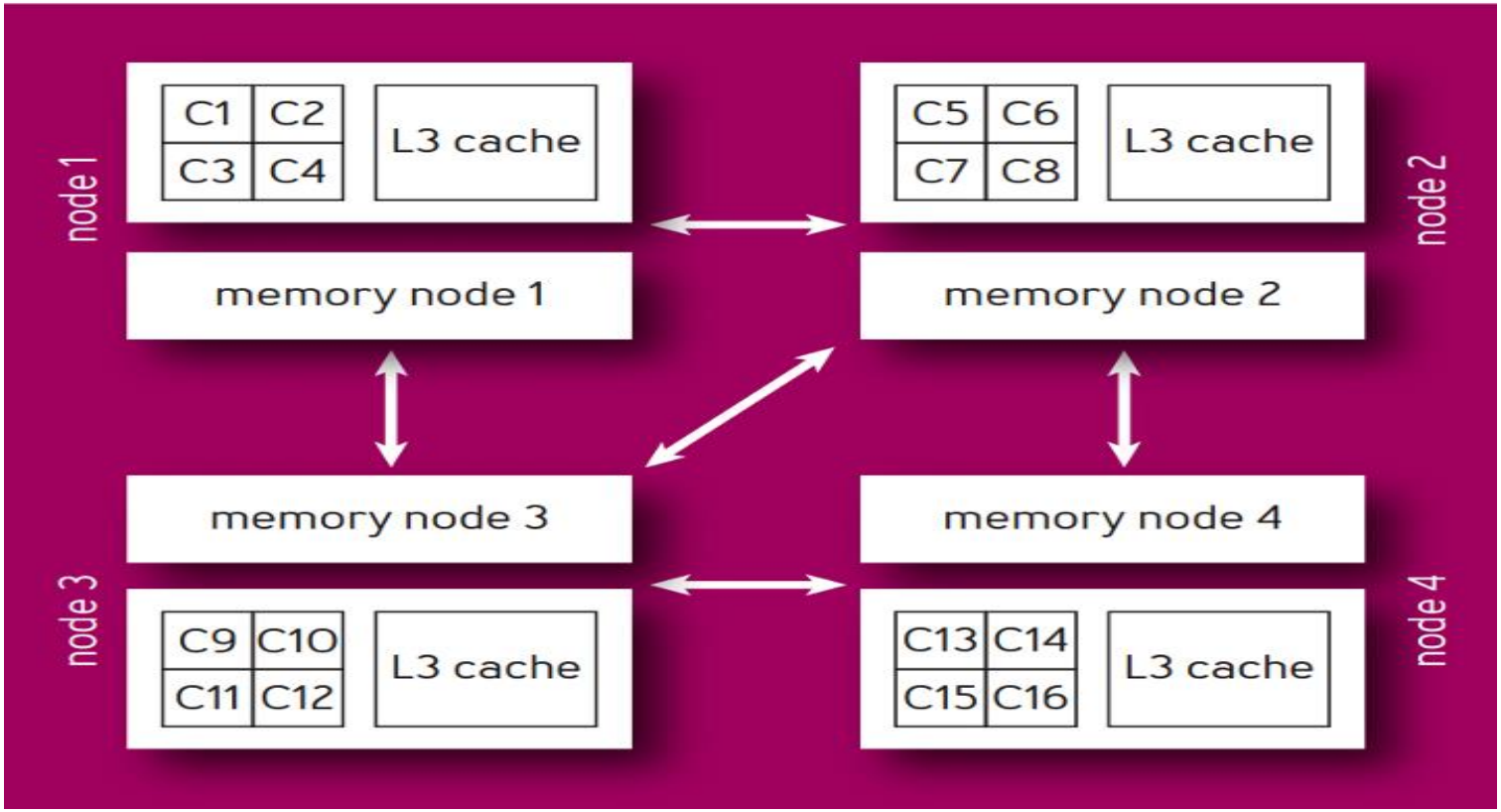
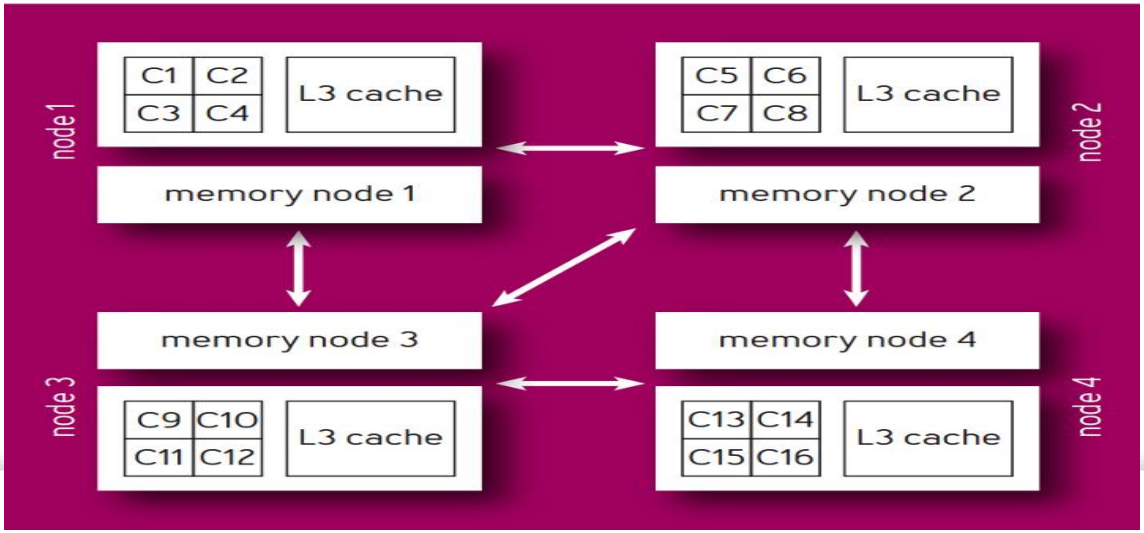


Figure courtesy: http://queue.acm.org/detail.cfm?id=2852078

NUMA performance problem

- Two different common NUMA memory-allocation policies
  - First-touch
    - Memory is allocated on the same node as the thread that first accesses the memory
  - Interleave
    - Distributes memory allocations equally on all nodes regardless of which threads access it
    - Good for memory allocation balance

FIGURE 1: **A MODERN NUMA SYSTEM**



NUMA performance problem: locality and congestion

- No one policy is best for all applications
- NUMA effects beyond the remote-access penalty can indeed severely affect performance
  - e.g., streamcluster

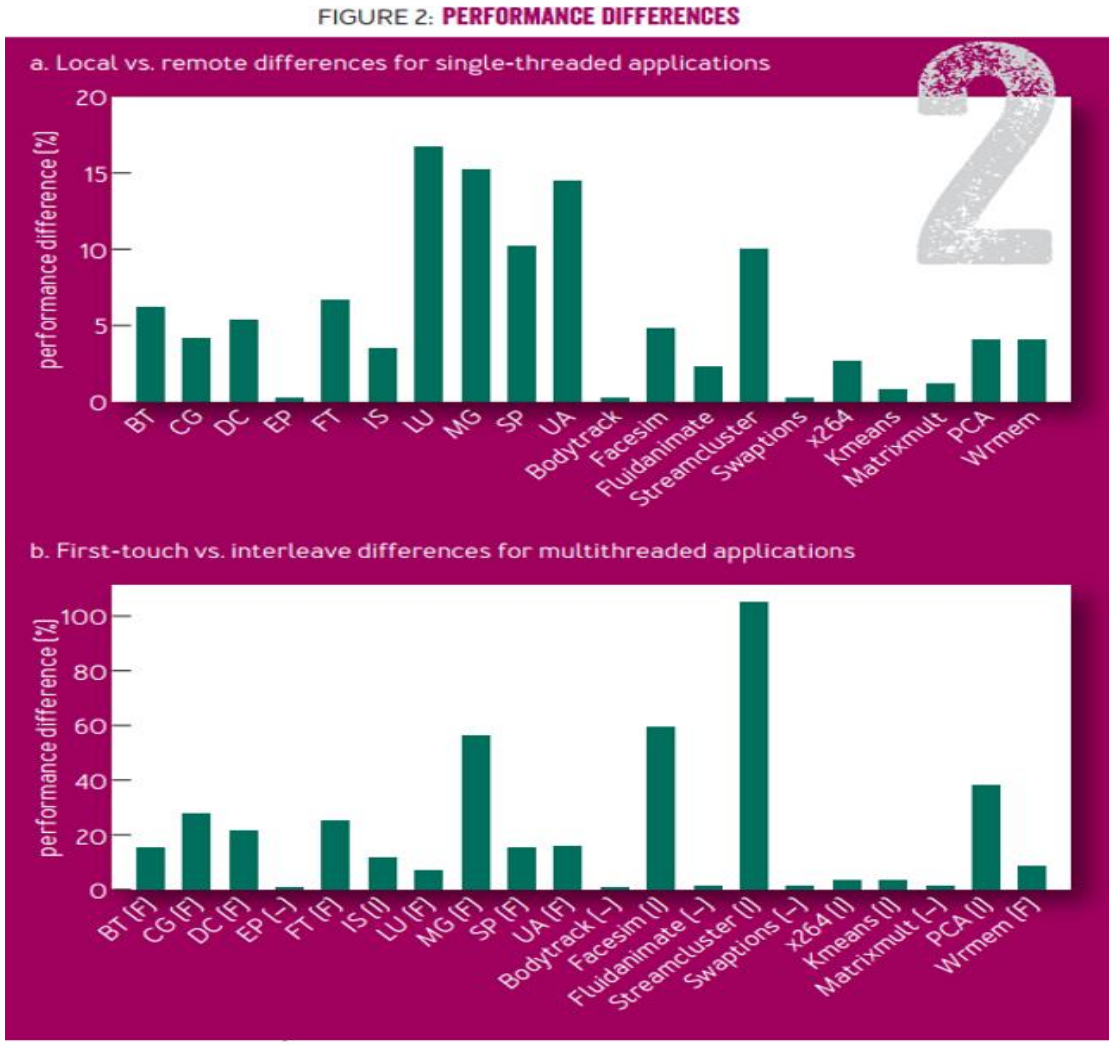


Figure courtesy: http://queue.acm.org/detail.cfm?id=2852078

NUMA performance problem: locality and congestion

- For streamcluster, the first-touch policy creates sever congestion
  - The average memory latency with the first-touch policy is more than double the latency of the interleave policy

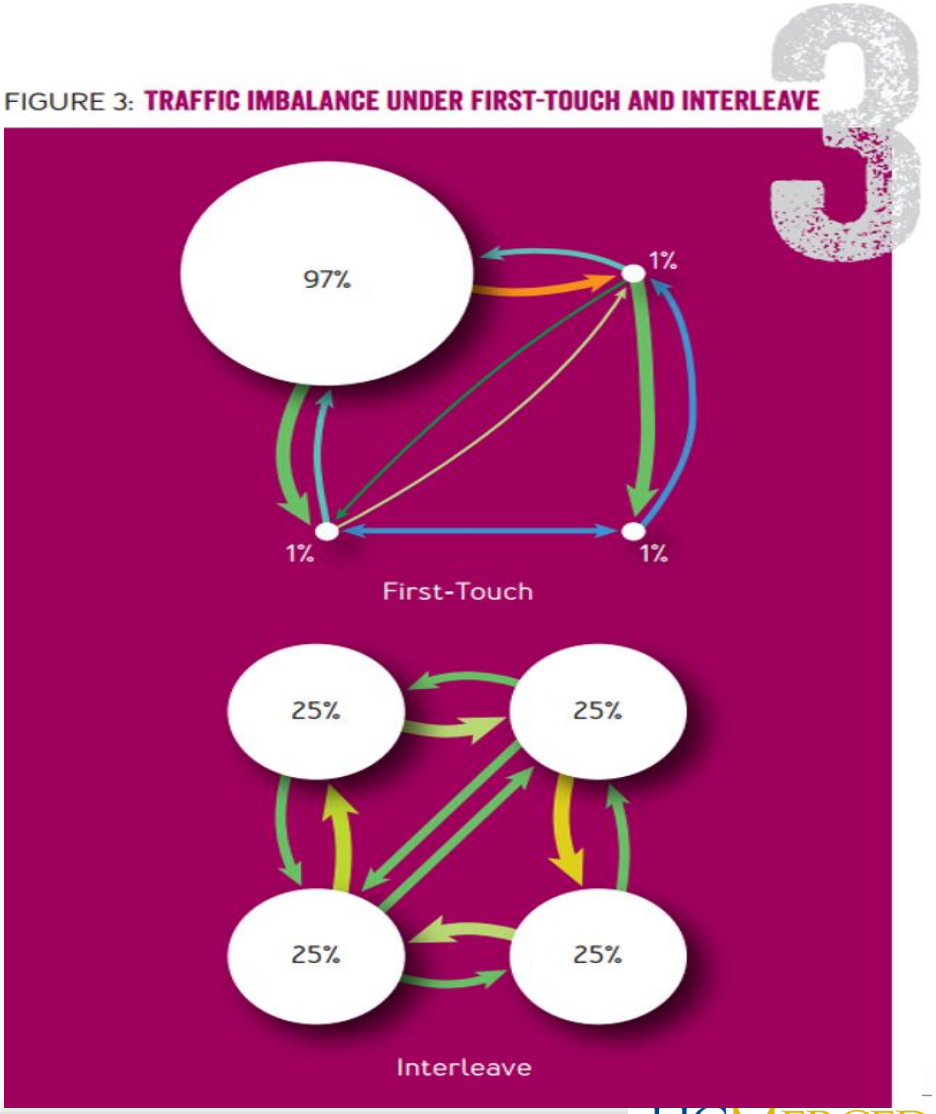


Figure courtesy: http://queue.acm.org/detail.cfm?id=2852078

Manage thread-level parallelism

- Many machine learning frameworks are based on fine-grained operations
  - e.g., tensorflow, caffe2, theano, MXNet, and CNTK
    - An example from inception_v3

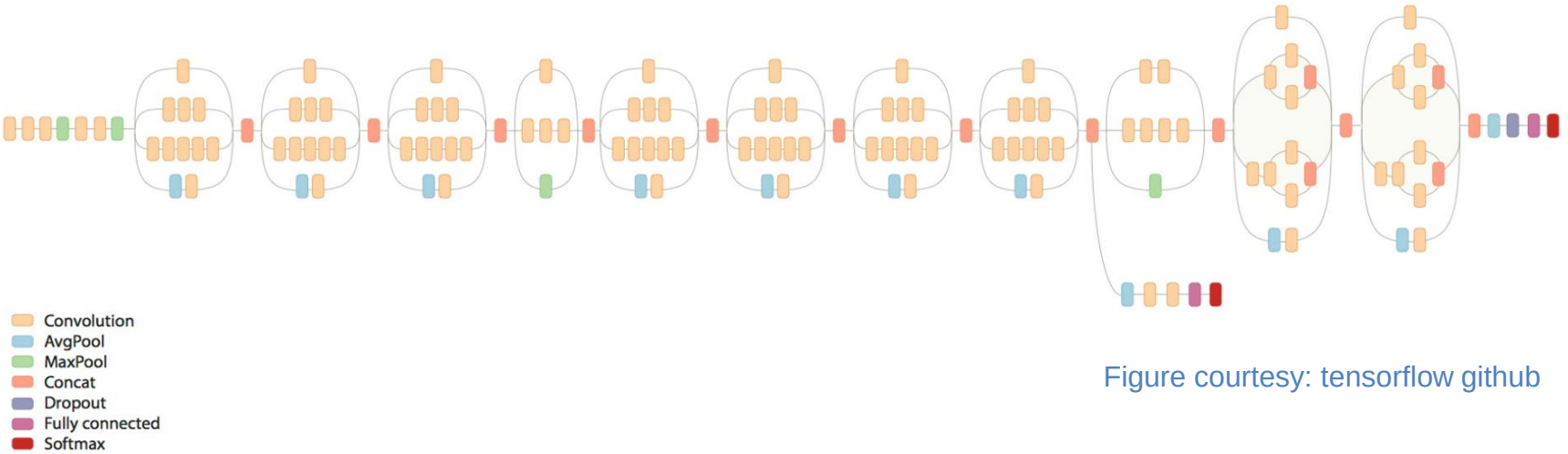
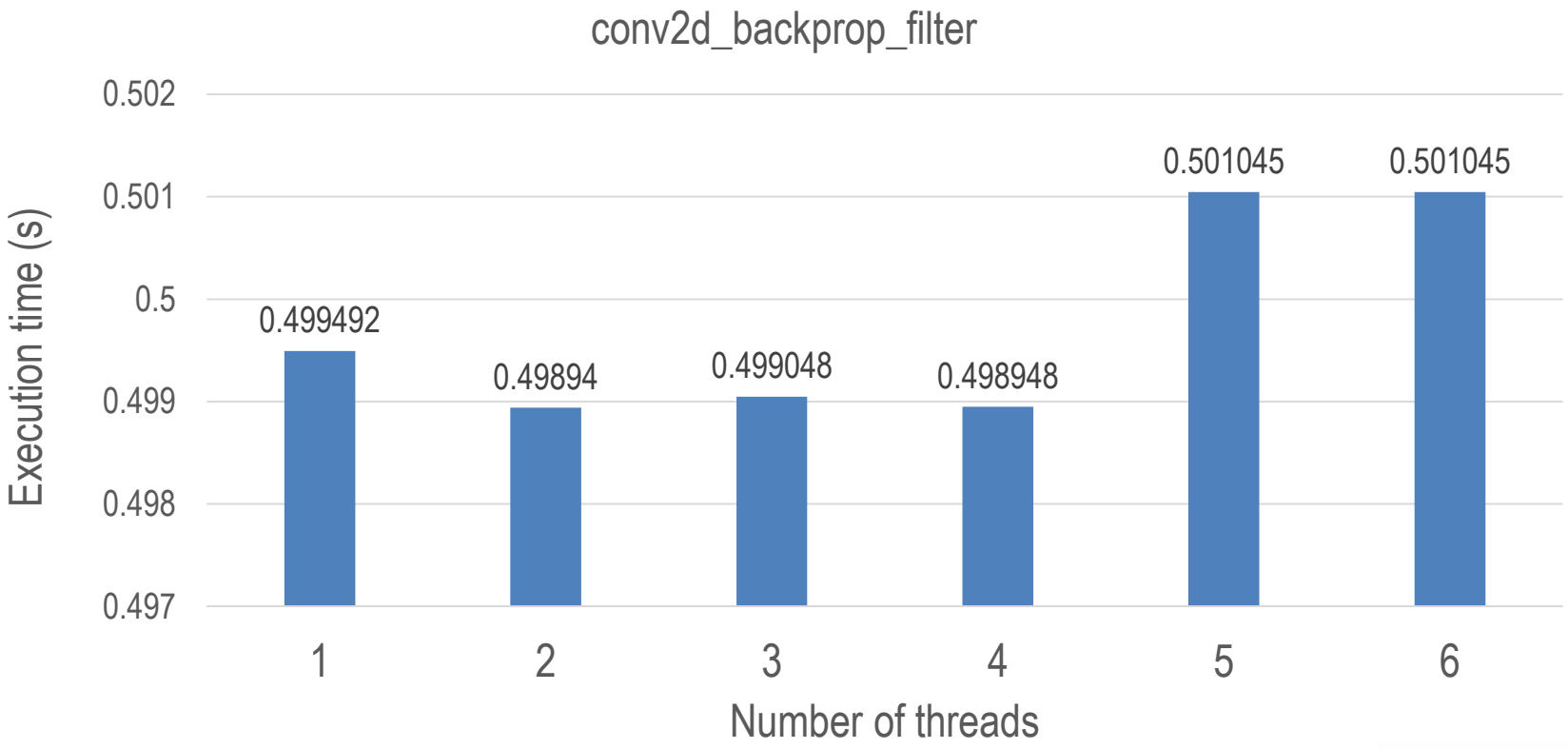


Figure courtesy: tensorflow github

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

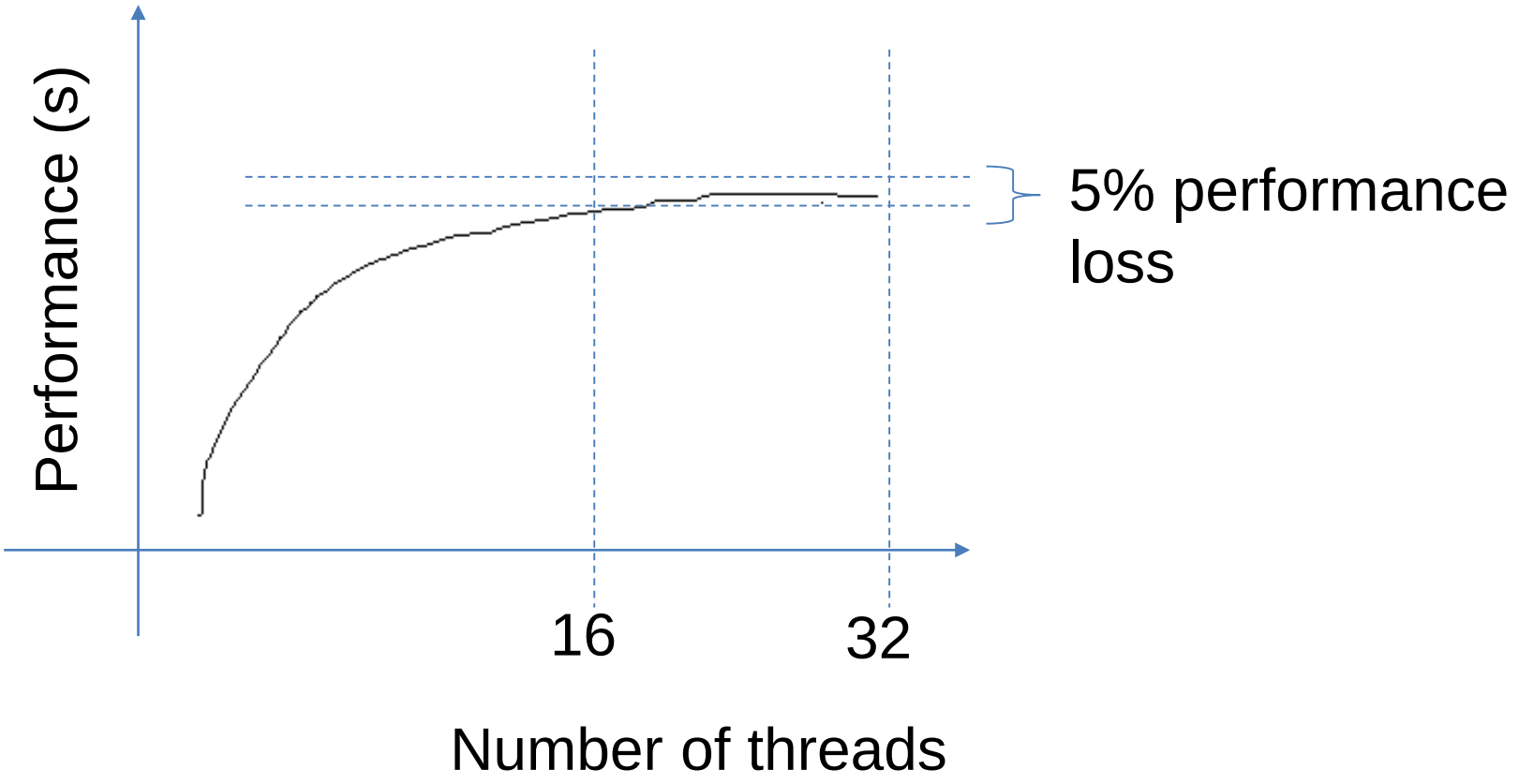
Manage thread-level parallelism

- Intra-op parallelism
  - Using maximum number of threads does not necessarily lead to best performance



Manage thread-level parallelism

- Inter-op parallelism
  - Co-run multiple operations to improve system throughput



Running Machine Learning Workloads on GPU

Running ML on GPU: performance improvement tips

- Optimize memory usage for maximum bandwidth
- Maximize occupancy to hide latency
- Control flow divergence

Optimize memory usage: basic strategies

- Processing data is cheaper than moving it around
  - Especially for GPUs as they devote many more transistors to ALUs than memory
- And will be increasingly so
  - The less memory bound a kernel is, the better it will scale with future GPUs
- So you want to:
  - Maximize use of low-latency, high-bandwidth memory
  - Optimize memory access patterns to maximize bandwidth
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
    - Kernels with high arithmetic intensity (ratio of math to memory transactions)
- Sometimes recompute data rather than cache it

Minimize CPU ↔ GPU data transfers

- CPU ↔ GPU memory bandwidth much lower than GPU memory bandwidth

- Minimize CPU ↔ GPU data transfers by moving more code from CPU to GPU
  - Even if that means running kernels with low parallelism computations
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to CPU memory

- Group data transfers
  - One large transfer much better than many small ones
  - Kernel fusion

Optimize memory access patterns

- Effective bandwidth can vary by an order of magnitude depending on access pattern

- Optimize access patterns to get:
  - *Coalesced* global memory accesses
  - Shared memory accesses with *no or few bank conflicts*
  - *Cache-efficient* texture memory accesses

Coalesced global memory accesses

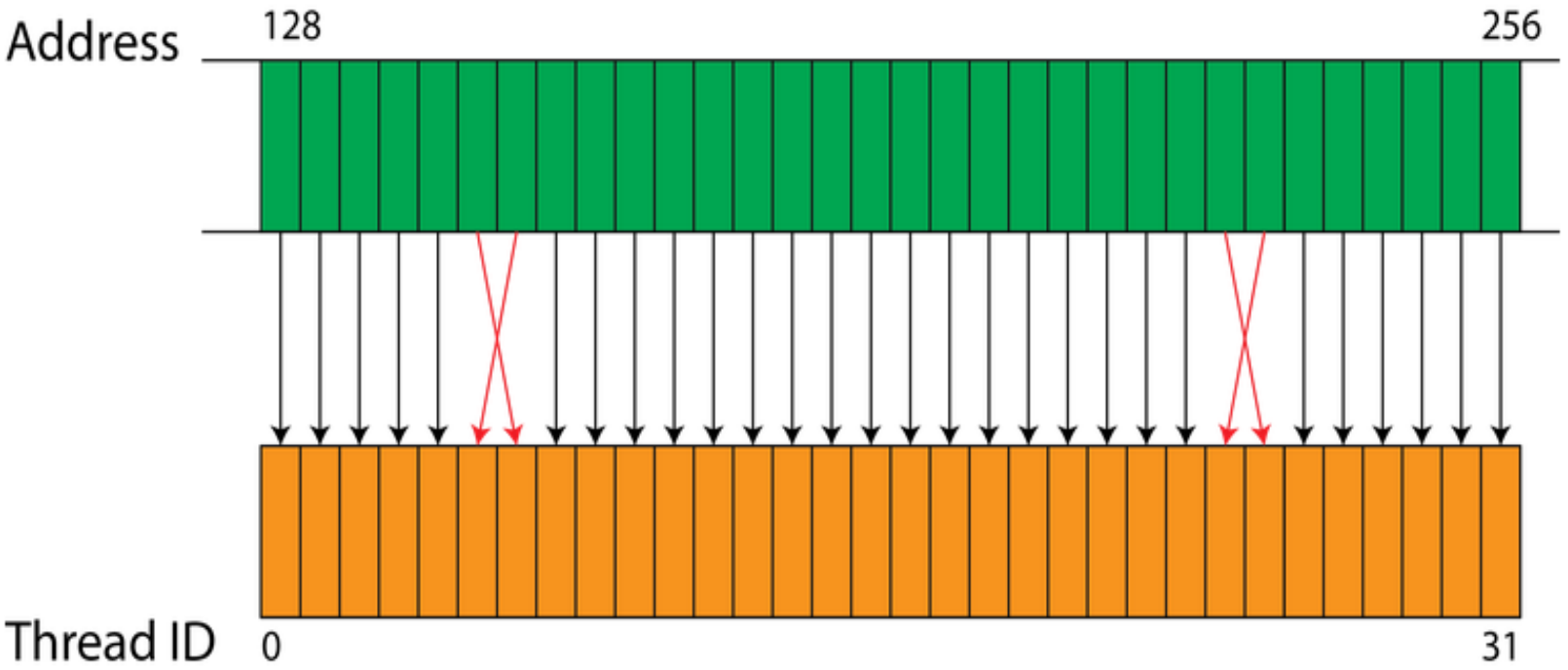
- Memory coalescing refers to combining multiple memory accesses into a single transaction
  - An example: 32 consecutive threads access 32 consecutive words

Sequential and aligned access:

Coalesced memory accesses

Coalesced global memory accesses

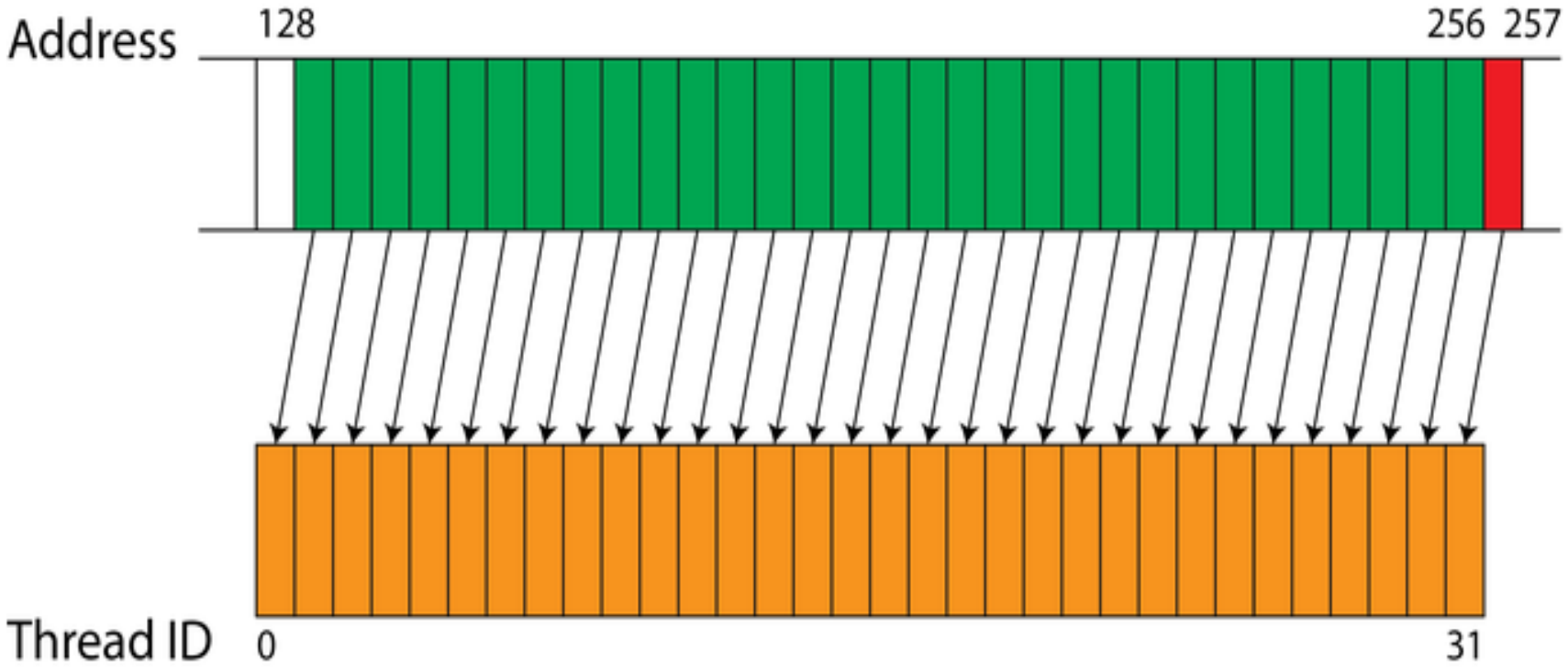
- **Aligned but non-sequential access**



Coalesced memory accesses

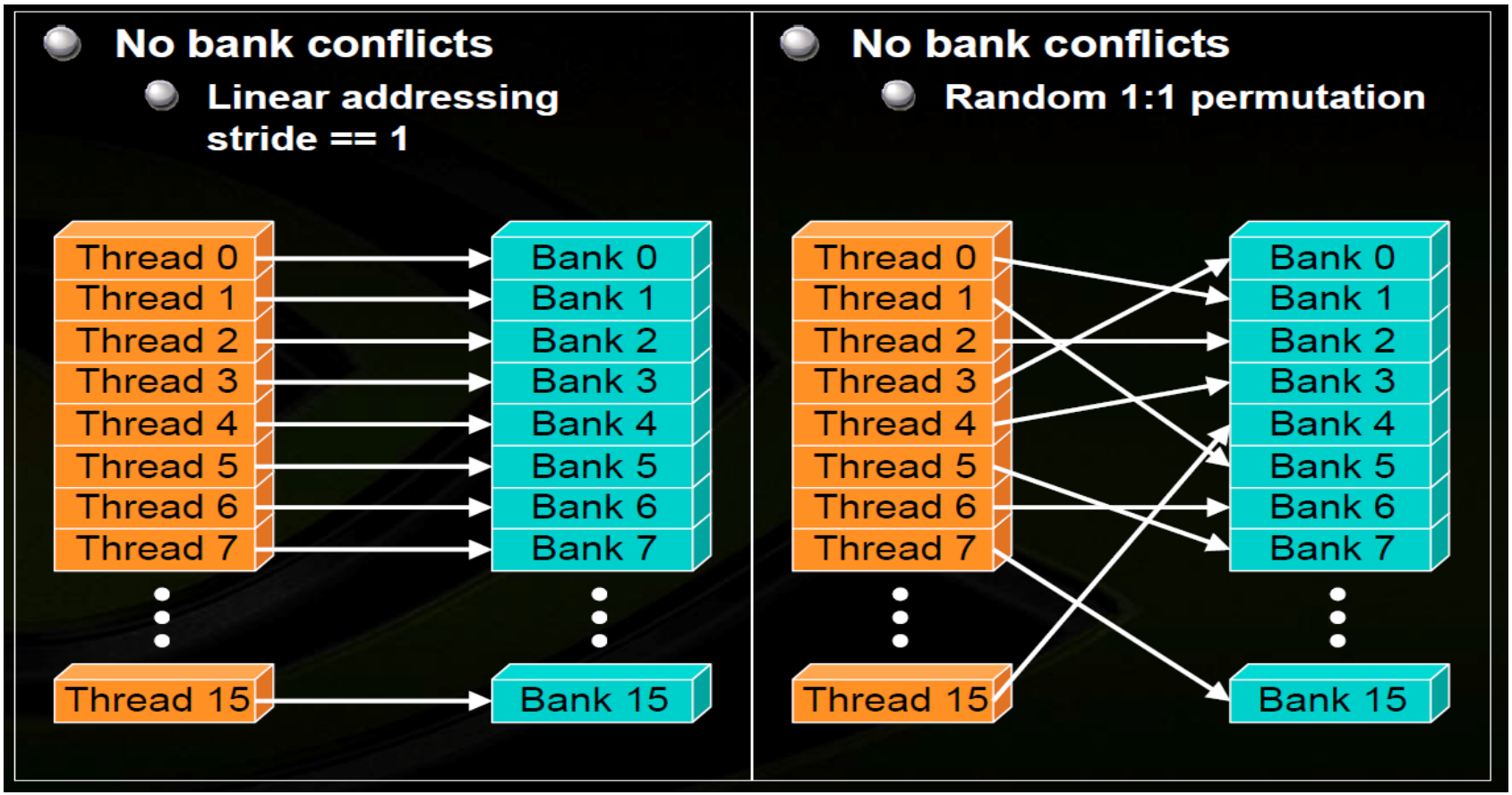
Coalesced global memory accesses

- **Unaligned Memory Access**

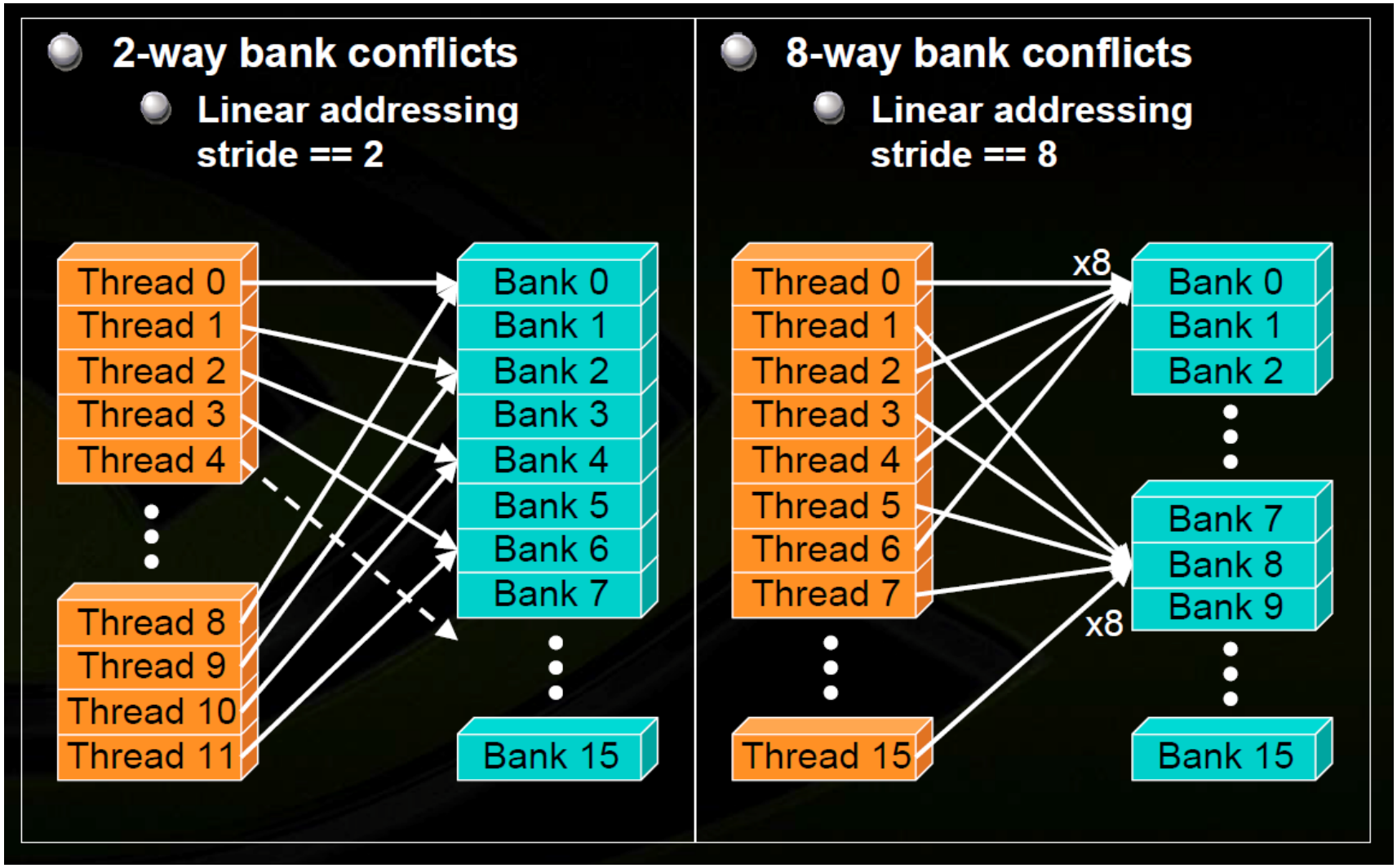


Non-coalesced memory accesses

Bank conflicts on shared memory

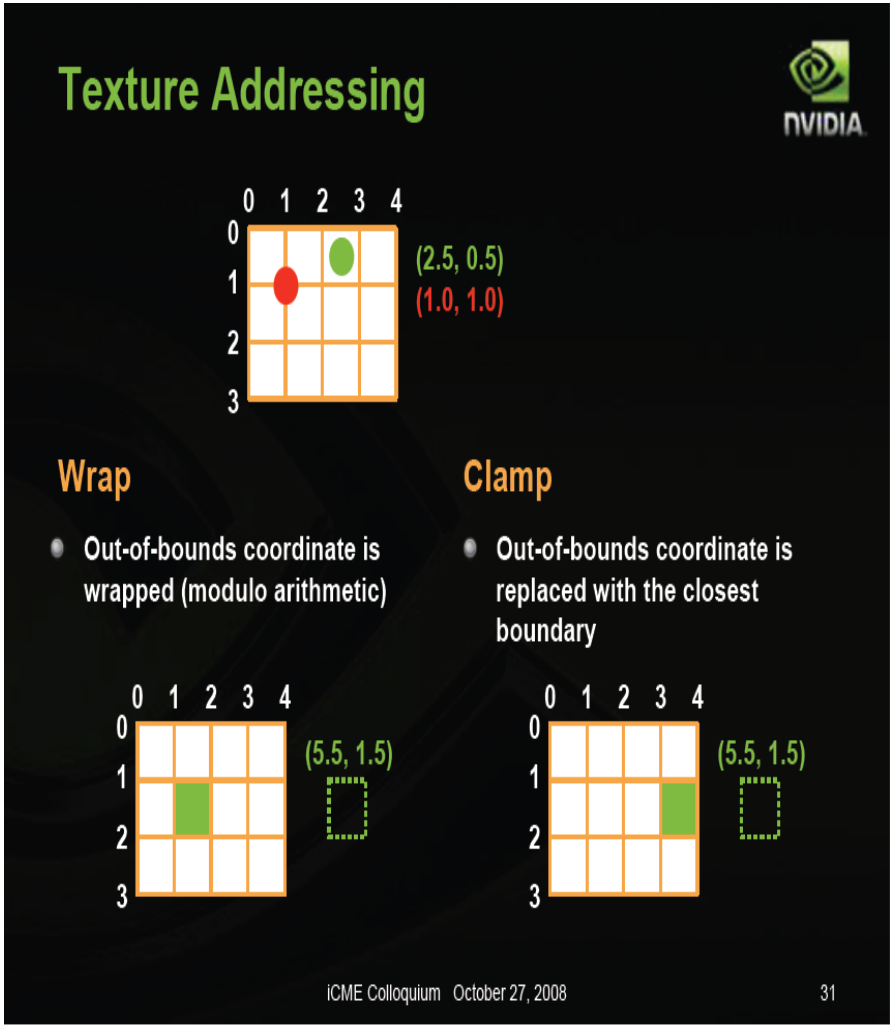


Bank conflicts on shared memory



Cache-efficient texture memory accesses

- Read-only object
- Texture fetches are cached
  - Optimized for 2D locality
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
  (Wrap, clamp)



Maximize occupancy to hide latency

- Sources of latency:
  - Global memory access: 400-600 cycle latency
  - Read-after-write register dependency
    - Instruction's result can only be read 11 cycles later

- Latency blocks dependent instructions in the same thread, but instructions in other threads are not blocked
  - Hide latency by running as many threads per multiprocessor as possible!

- Choose execution configuration to maximize

  occupancy = (# of active warps) / (maximum #of active warps)

Maximize occupancy to hide latency

- occupancy = (# of active warps) / (maximum #of active warps)

- Remember: resources are allocated for the entire block
  - Resource are finite
  - Utilizing too many resources per thread may limit the occupancy

- Potential occupancy limiters
  - Register usage
  - Shared memory usage
  - Block size

Occupancy limiters: registers

- Register usage: compile with --ptxas-options=-v

- Fermi has 32K registers per SM
  - Fermi can have up to 48 active warps per SM (1536 threads)

- Example 1
  - Kernel uses 20 registers per thread (+1 implicit)
  - Active threads = 32K/21 = 1560 threads
  - > 1536 thus an occupancy of 1

- Example 2
  - Kernel uses 63 registers per thread (+1 implicit)
  - Active threads = 32K/64 = 512 threads
  - 512/1536 = .3333 occupancy

Occupancy limiters: registers

- Use –maxrregcount=N flag to *nvcc* to control register usage
  - N = desired maximum registers / kernel
  - At some point “spilling” into local memory may occur
  - Reduces performance –-- local memory is slow
  - Check for LMEM usage
    - Compiler output
      - nvcc option: –Xptxas, –v, –abi=no
      - Will print the number of lmem bytes for each kernel
    - Profiler: e.g, CUPTI

Occupancy limiters: shared memory

- Shared memory usage: compile with --ptxas-options=-v
  - Reports shared memory per block
- Fermi has either 16K or 48K shared memory
- Example 1, 48K shared memory
  - Kernel uses 32 bytes of shared memory per thread
  - 48K/32 = 1536 threads
  - occupancy=1
- Example 2, 16K shared memory
  - Kernel uses 32 bytes of shared memory per thread
  - 16K/32 = 512 threads
  - occupancy=.3333
- Don’t use too much shared memory
- Choose L1/Shared config appropriately.

Occupancy limiter: block size

- Each SM can have up to 8 active blocks

- A small block size will limit the total number of threads

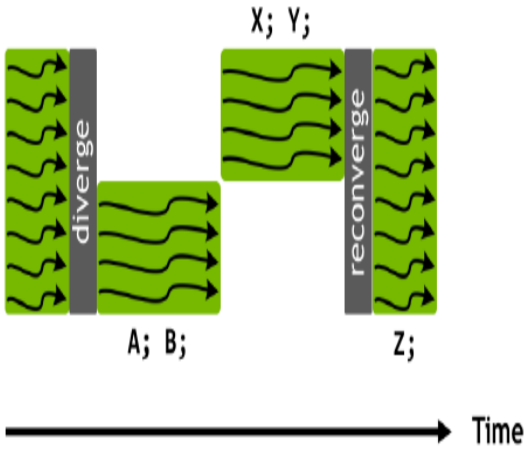
Block Size	Active Threads	Occupancy
32	256	.1666
64	512	.3333
128	1024	.6666
192	1536	1
256	2048 (1536)	1

Control flow instructions

- Main performance concern with branching is *divergence*

- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - If (threadIdx.x > 2) { }
    - Branch granularity < warp size

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

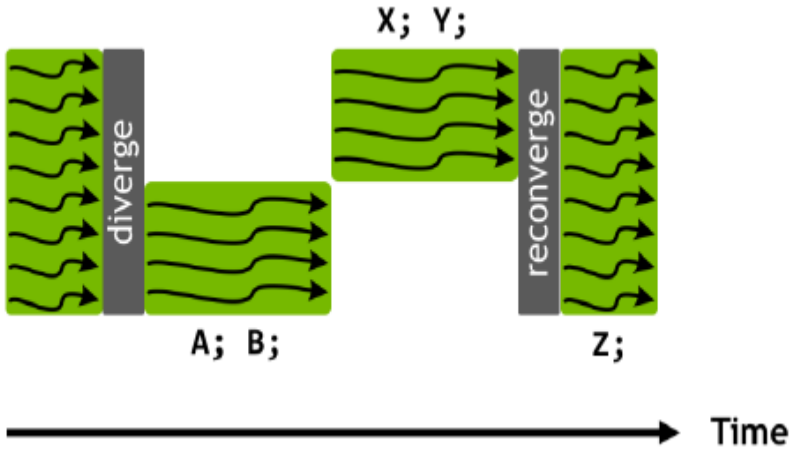


  - Example without divergence:
    - If (threadIdx.x / WARP_SIZE > 2) { }
    - Branch granularity is a whole multiple of warp size

Control flow instructions

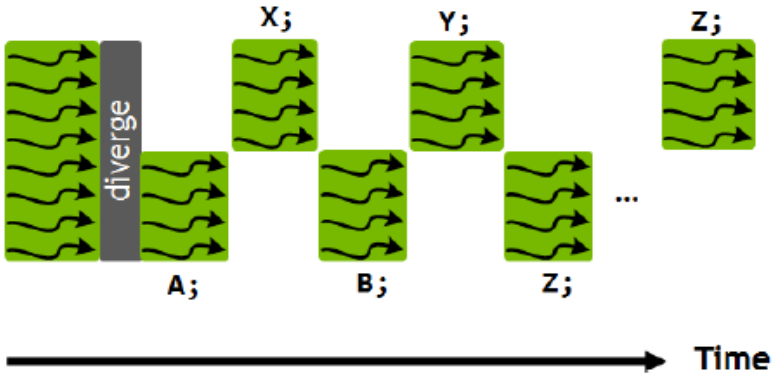
- New change to SIMT model in Volta reduces the negative impact of control flow divergence a little

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



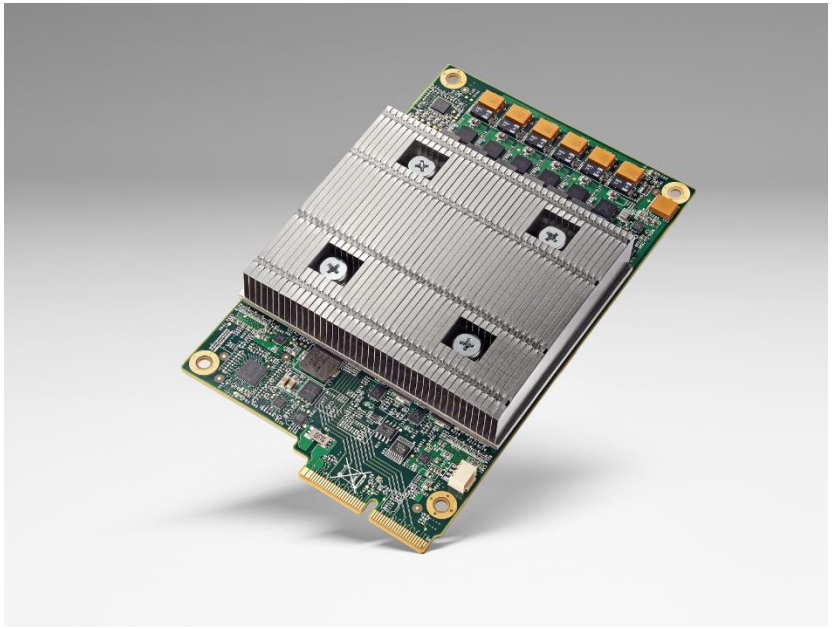
Old model

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



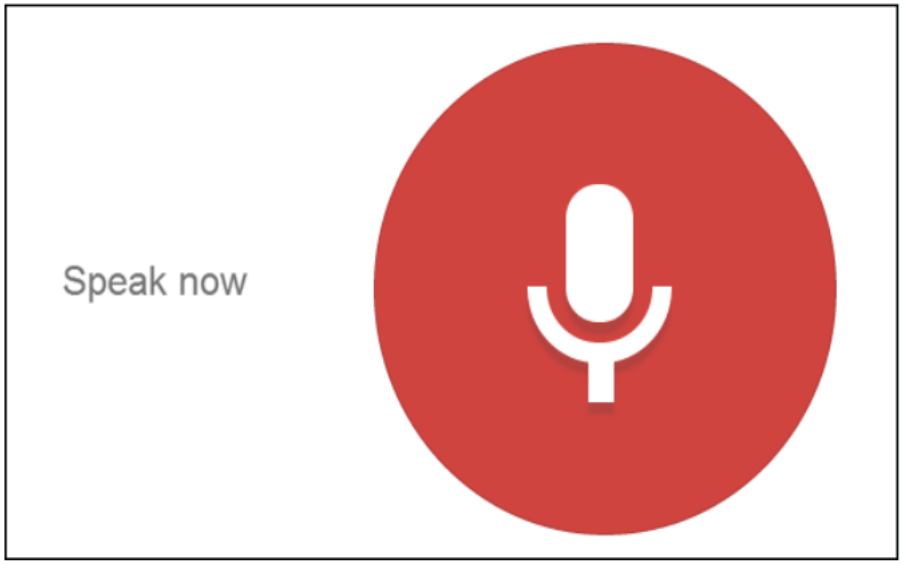
New model

Google Tensorflow Processing Unit (TPU)



Motivation for TPU

- **2006:** "Just run DNNs on our CPU datacenter. It's basically free."
- **2013:** "3 minutes of DNN-based voice search == 2x more datacenter compute."



Tensor Processing Unit (TPU)

- **30-80x** TOPS/watt vs. 2015 CPUs and GPUs.
- 8 GiB DRAM.
- 8-bit fixed point.
- 256x256 MAC unit.
- Support for data reordering, matrix multiply, activation, pooling, and normalization.

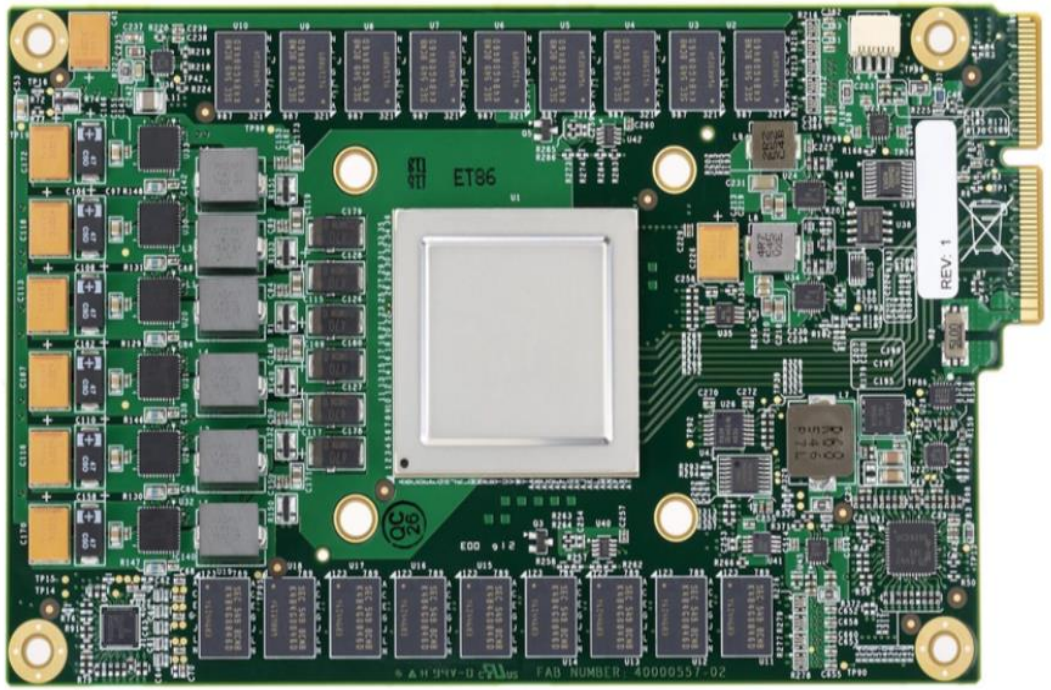


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

TPU is “simple”

- It has none of the sophisticated microarchitectural features
  - Cache, branch prediction, out-of-order execution, multiprocessing, speculative prefetching, address coalescing, multithreading, context switching
- It uses less transistors and consumes less energy than regular CPU
- Aims to improve the average case but not the 99th-percentile case

Application testbed

Name	LOC	Layers					Nonlinear function	Weights	TPU Ops / Weight Byte	TPU Batch Size	% of Deployed TPUs in July 2016
		FC	Conv	Vector	Pool	Total					
MLP0	100	5				5	ReLU	20M	200	200	61%
MLP1	1000	4				4	ReLU	5M	168	168	
LSTM0	1000	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1500	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1000		16			16	ReLU	8M	2888	8	5%
CNN1	1000	4	72		13	89	ReLU	100M	1750	32	

Table 1. Six NN applications (two per NN type) that represent 95% of the TPU's workload. The columns are the NN name; the number of lines of code; the types and number of layers in the NN (FC is fully connected, Conv is convolution, Vector is self-explanatory, Pool is pooling, which does nonlinear downsizing on the TPU; and TPU application popularity in July 2016. One DNN is RankBrain [Cla15]; one LSTM is a subset of GNM Translate [Wu16]; one CNN is Inception; and the other CNN is DeepMind AlphaGo [Sil16][Jou15].

"The unexpected desire for TPUs by many Google services combined with the preference for **low response time** changed the equation, with application writers often **opting for reduced latency** over waiting for bigger batches to accumulate."

“Patterson” Discussion

- Fallacy: NN inference applications in data centers value throughput as much as response time.
  - 10ms in 2014 → 7ms in 2015-2016

- Fallacy: The K80 GPU architecture is a good match to NN inference
  - GPU is a high thoughput architecture that relies on high-bandwidth memory and thousands of threads

- Pitfall: Architects have neglected important NN tasks

> “CNNs constitute only about 5% of the representative NN workload for Google. More attention should be paid to MLPs and LSTMs. Repeating history, **it’s similar to when many architects concentrated on floating-point performance when most mainstream workloads turned out to be dominated by integer operations.”**

“Patterson” Discussion

- Pitfall: For NN hardware, Inferences Per Second (IPS) is an inaccurate summary performance metric.
  - IPS is more of a function of NN than of the underlying hardware
  - For example, the TPU runs the 4-layer MLP at 360,000 IPS but the 89-layer CNN at only 4,700 IPS
- Fallacy: The K80 GPU results would be much better if boost mode were enabled
  - Improvement in performance/Watt is only 1.1x
- Fallacy: After two years of software tuning, the only path left to increase TPU performance is hardware upgrades.

NVIDIA's Rebuttal to the TPU

	K80 2012	TPU 2015	P40 2016
Inferences/Sec <10ms latency	$^1/_{13}$X	1X	2X
Training TOPS	6 FP32	NA	12 FP32
Inference TOPS	6 FP32	90 INT8	48 INT8
On-chip Memory	16 MB	24 MB	11 MB
Power	300W	75W	250W
Bandwidth	320 GB/S	34 GB/S	350 GB/S

NVIDIA New GPU for Machine Learning: Volta GV100 GPU

Figure courtesy: Nividia Volta architecture white paper

Figure 1. NVIDIA Tesla V100 SXM2 Module with Volta GV100 GPU

Key features

- New tensor cores
  - Eight tensor cores per streaming multiprocessor
    - Each tensor core performs 64 floating point FMA operations per clock
  - Each tensor core operatates on 4x4 matrix, and performs

    D= A x B + C

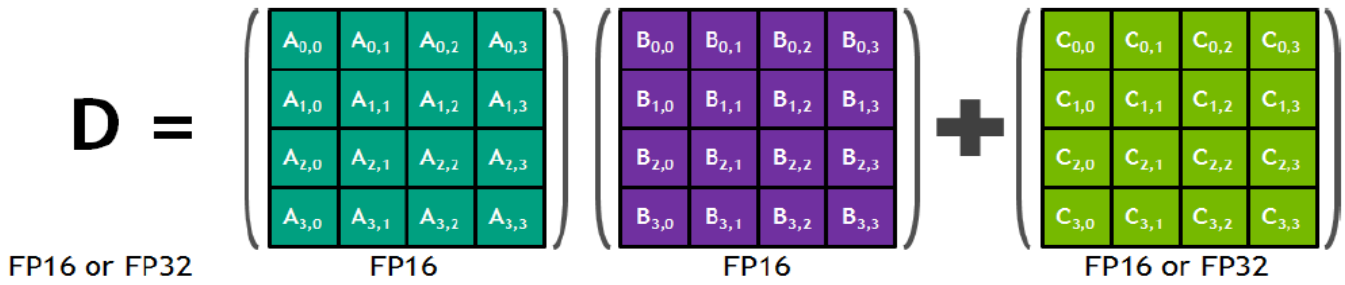


Figure 8. Tensor Core 4x4 Matrix Multiply and Accumulate

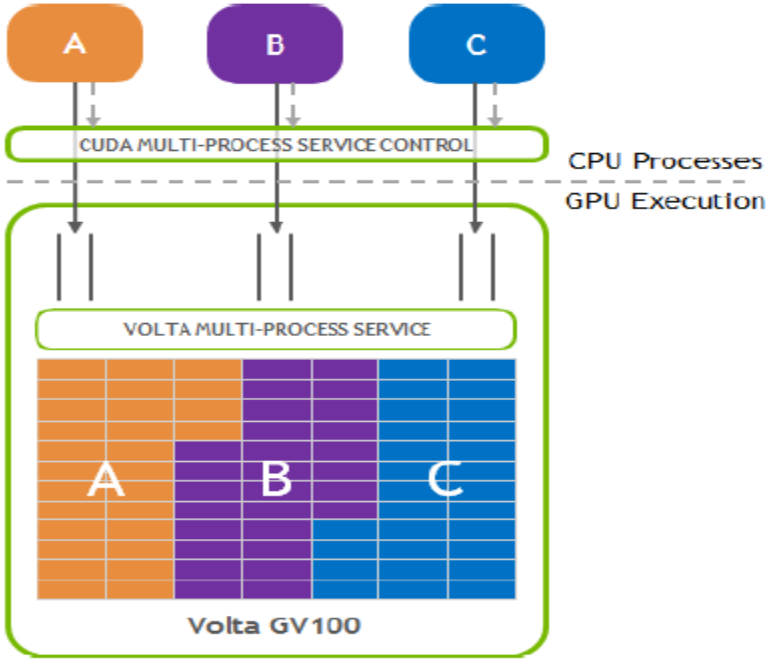
Key features

- NVLink 2.0
  - High bandwidth between CPU and GPU and between GPU and GPU: 300GB/s
  - CPU to regular main memory bandwidth on Intel Xeon Phil (knights landing): ~90 GB/s

- High-bandwidth memory
  - Memory stacks located on the same physical package as GPU
  - 900GB/s, 1.5x delivered memory bandwidth versus Pascal GP100 (the last version)
  - CPU to high bandwidth memory on Intel Xeon Phil (Knights Landing): ~475-490 GB/s

Key features

- Multi-Process Service
  - Allow multiple applications to simultaneously share GPU execution resources
    - Permitting many individual inference jobs to be submitted concurrently to GPU and improve overall GPU utilization



Key features

- Maximum Performance and Maximum Efficiency Modes
  - A not-to-exceed power cap can be set
- Cooperative Groups and New Cooperative Launch APIs
  - Pascal and Volta include support for new cooperative launch APIs that support synchronization amongst CUDA thread blocks

References

- A Guide to Vectorization with Intel® C++ Compilers,
  - https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers
- TPU slides on ISCA'17
  - https://homes.cs.washington.edu/~cdel/presentations/TPUPaperISCA2017.pdf
- NVIDIA white paper (NVIDIA Tesla V100 GPU architecture)
  - http://www.nvidia.com/object/volta-architecture-whitepaper.html
- B. Wilkinson, "GPU Memories", ITCS 6/8010 CUDA programming, UNC-Charlotte, 2011
- Justin Luitjens and Steven Rennich, "CUDA Warps and Occupancy", GPU Computing Webinar, July 12, 2011