P&S Heterogeneous Systems

Parallel Patterns: Convolution

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2021
25 November 2021

Parallel Patterns

Reduction Operation

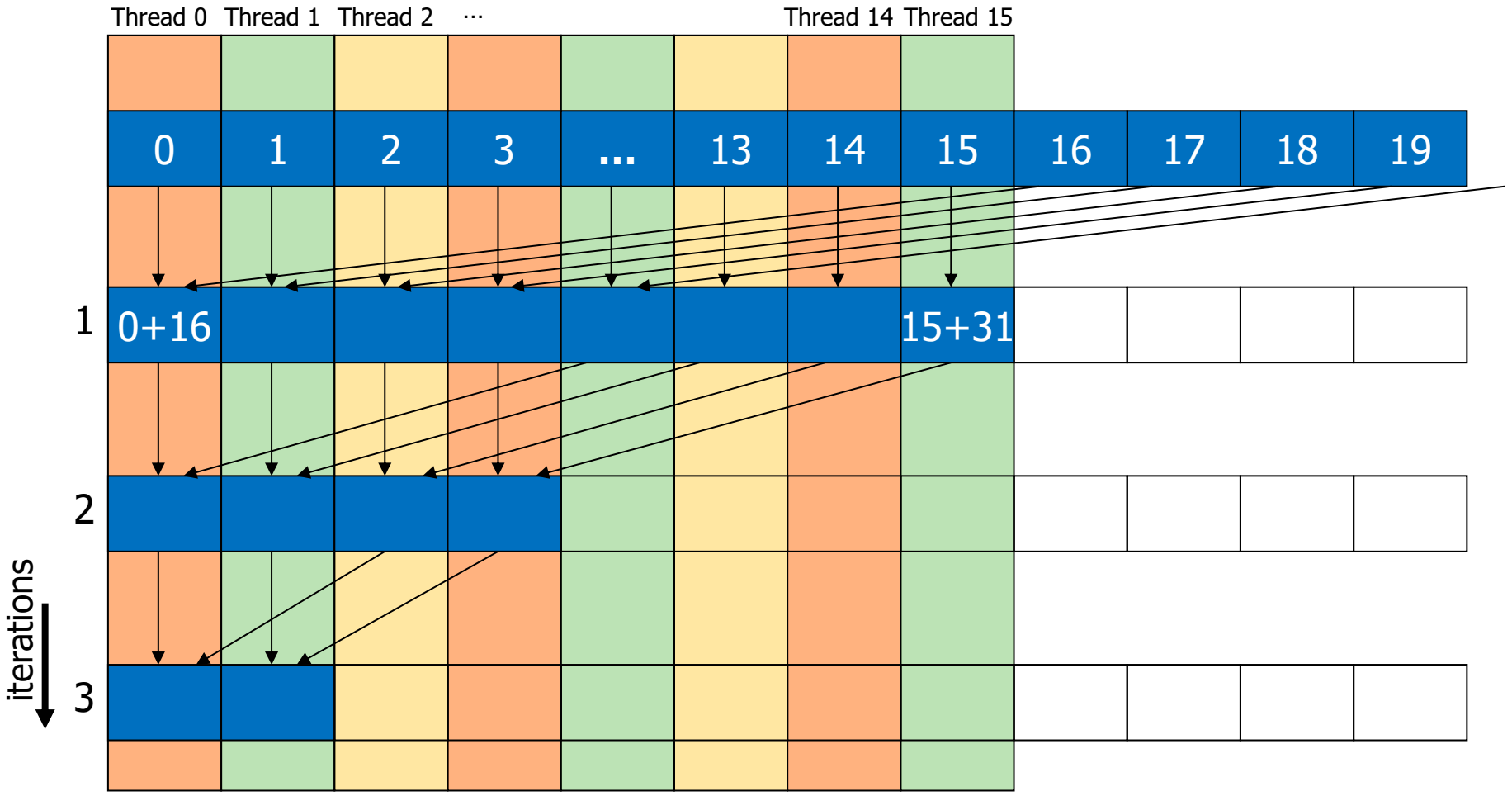
- A reduction operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples

- Properties of reduction
  - Associativity
  - Commutativity
  - Identity value

- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

Divergence-Free Mapping (I)

- All active threads belong to the same warp

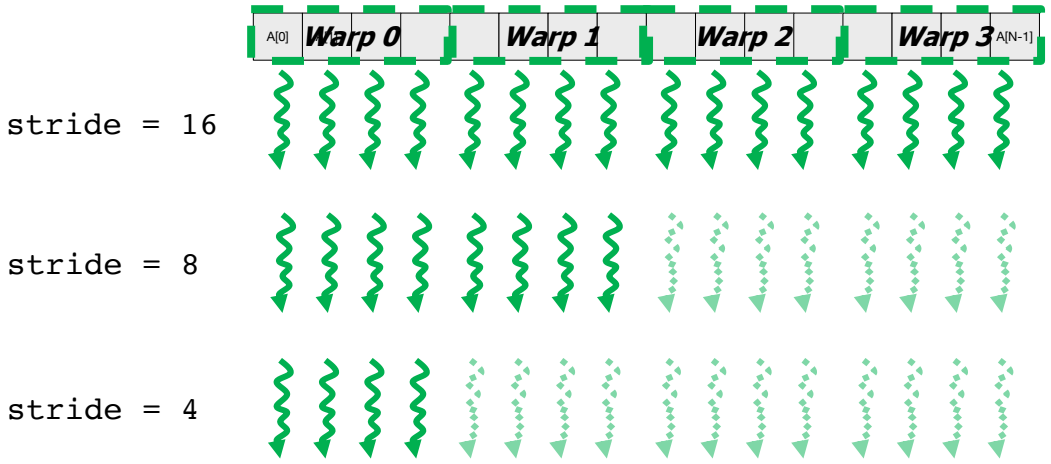


Slide credit: Hwu & Kirk

Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 0;  stride >> 1){

  __syncthreads();

  if (t < stride)
    partialSum[t] += partialSum[t + stride];

}
```



Warp utilization is maximized

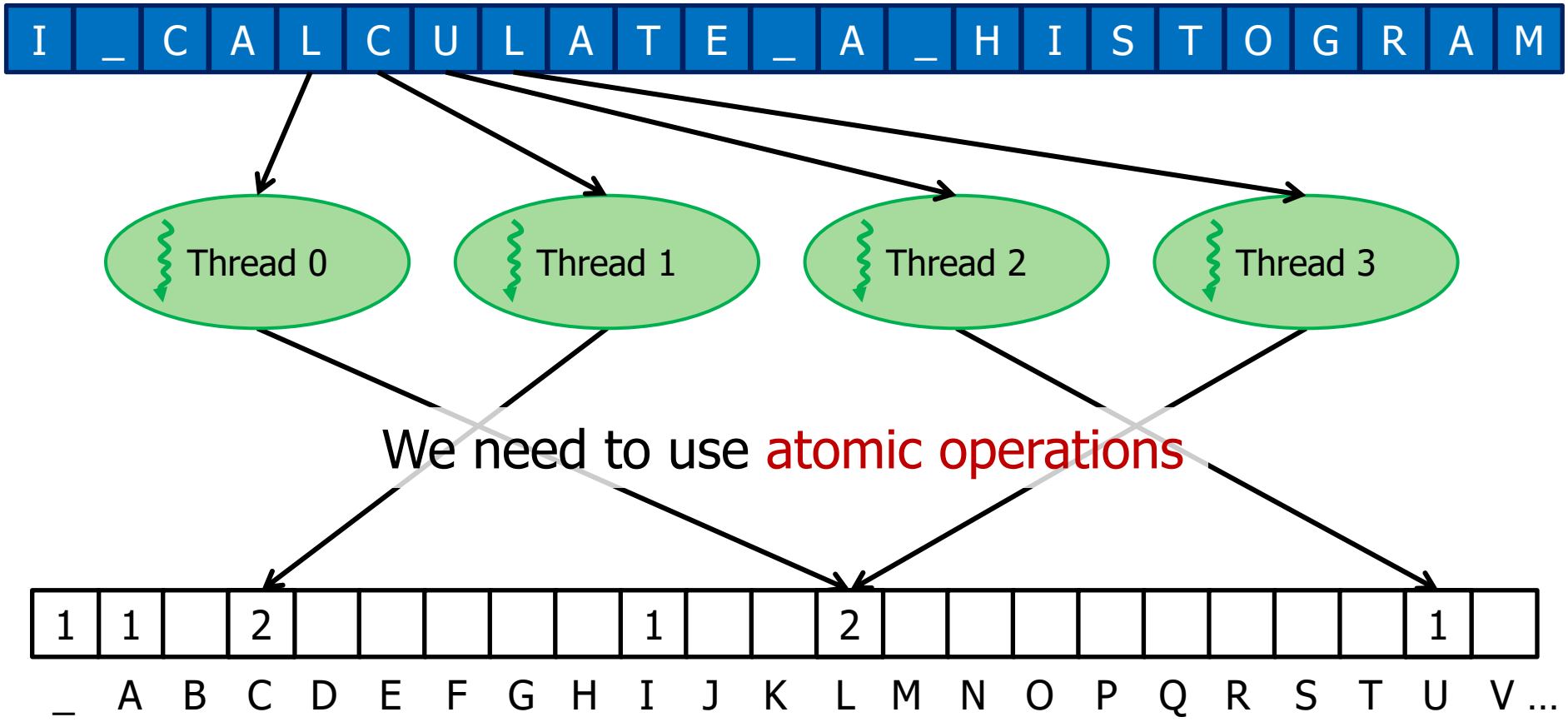
Histogram Computation

- Histogram is a frequently used computation for reducing the dimensionality and extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …
- Basic histograms - for each element in the data set, use the value to identify a "bin" to increment
  - Divide possible input value range into "bins"
  - Associate a counter to each bin
  - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

Slide credit: Hwu & Kirk

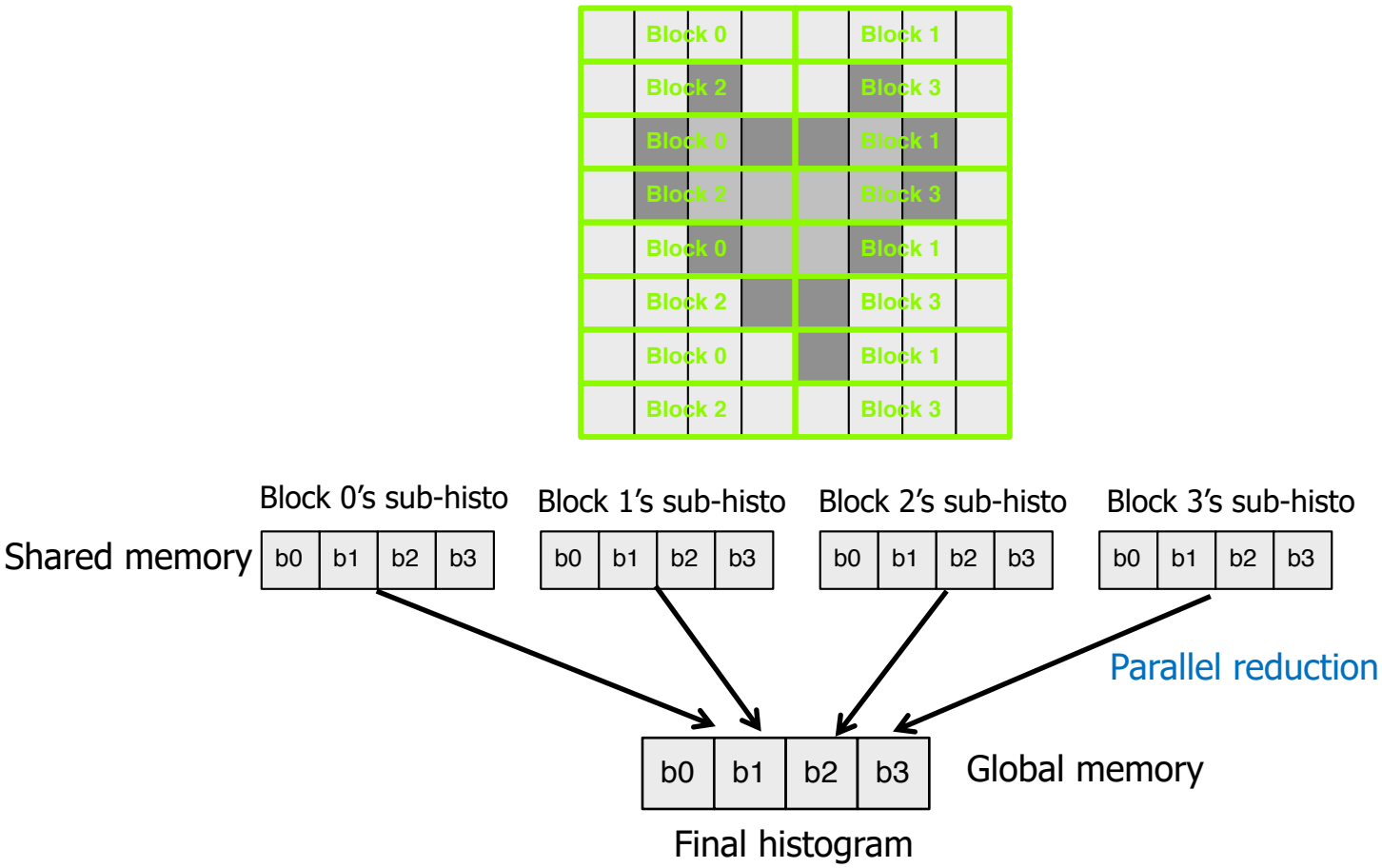
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
  - Each thread moves to element threadID + #threads



Histogram Privatization

- Privatization: Per-block sub-histograms in shared memory
  - Threads use atomic operations in shared memory



Convolution

Convolution Applications

- Convolution is a widely-used operation in signal processing, image processing, video processing, and computer vision

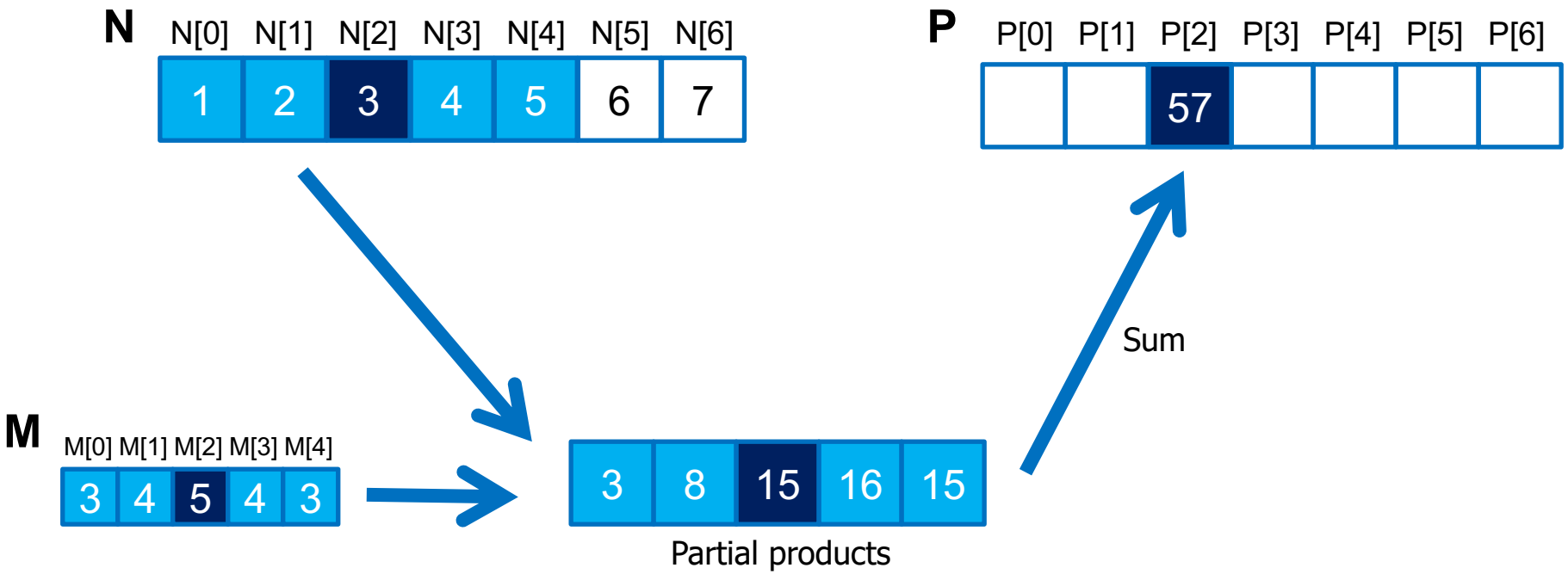
- Convolution applies a filter or mask or kernel* on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a weighted sum of a set of neighboring input elements
  - Smoothing, sharpening, or blurring an image
  - Finding edges in an image
  - Removing noise, etc.

- Applications in machine learning and artificial intelligence
  - Convolutional Neural Networks (CNN or ConvNets)

* The term "kernel" may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

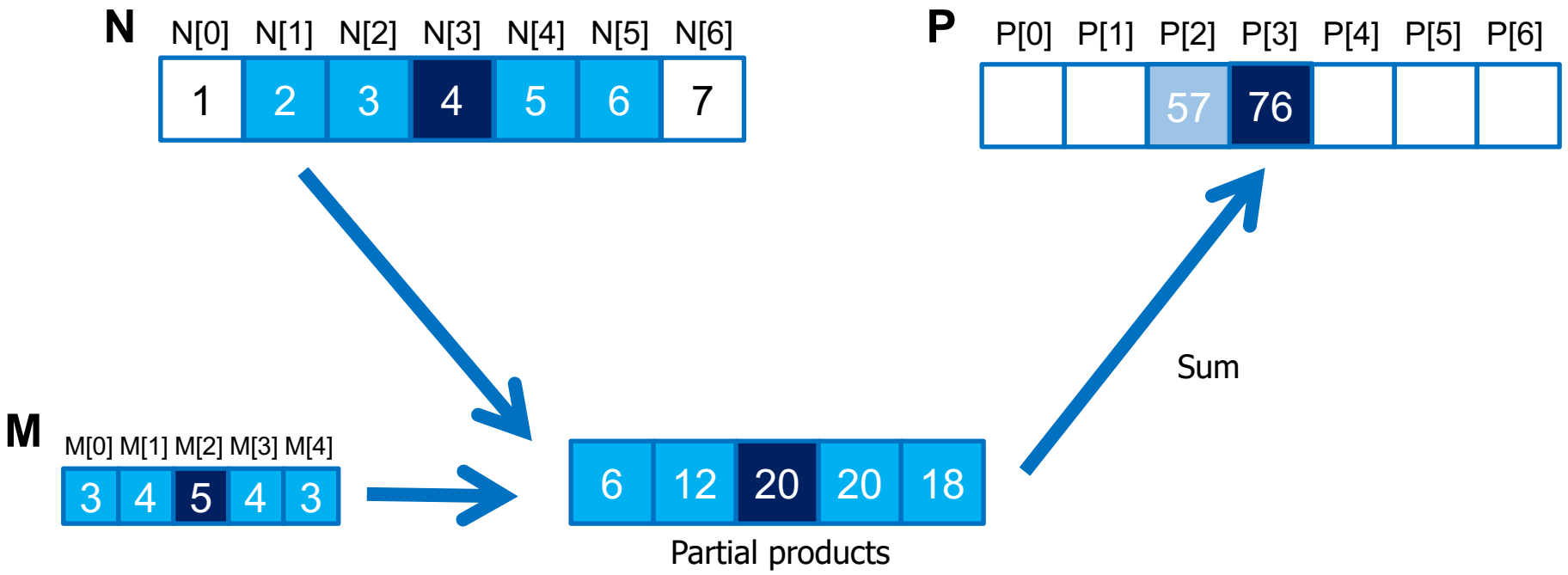
1D Convolution Example

- Commonly used for audio processing
- Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]:



1D Convolution Example: Next Element

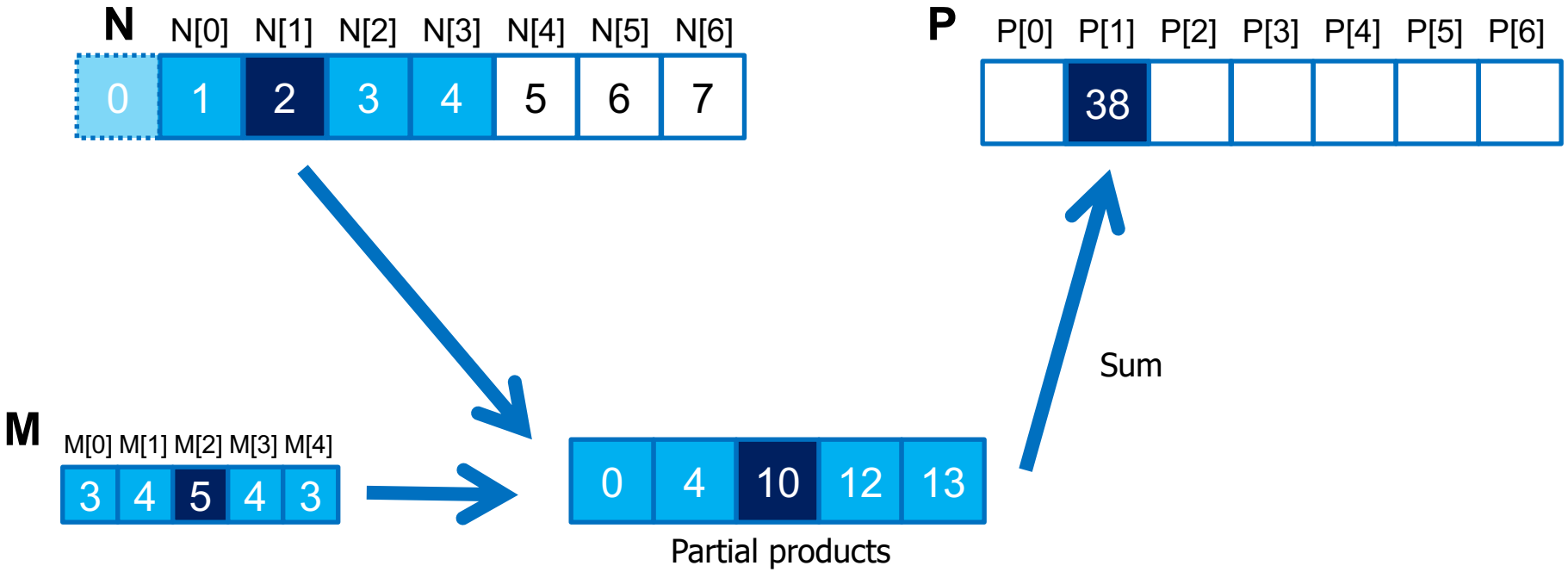
- Calculation of P[3]



Slide credit: Hwu & Kirk

1D Convolution Boundary Condition

- Calculation of output elements near the boundaries (beginning and end) of the input array need to deal with "ghost" elements
  - Different policies (0, replicates of boundary values, etc.)

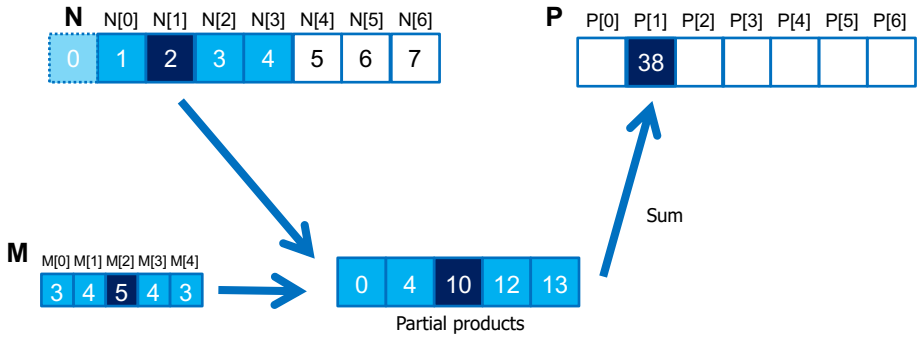


Slide credit: Hwu & Kirk

1D Convolution Kernel

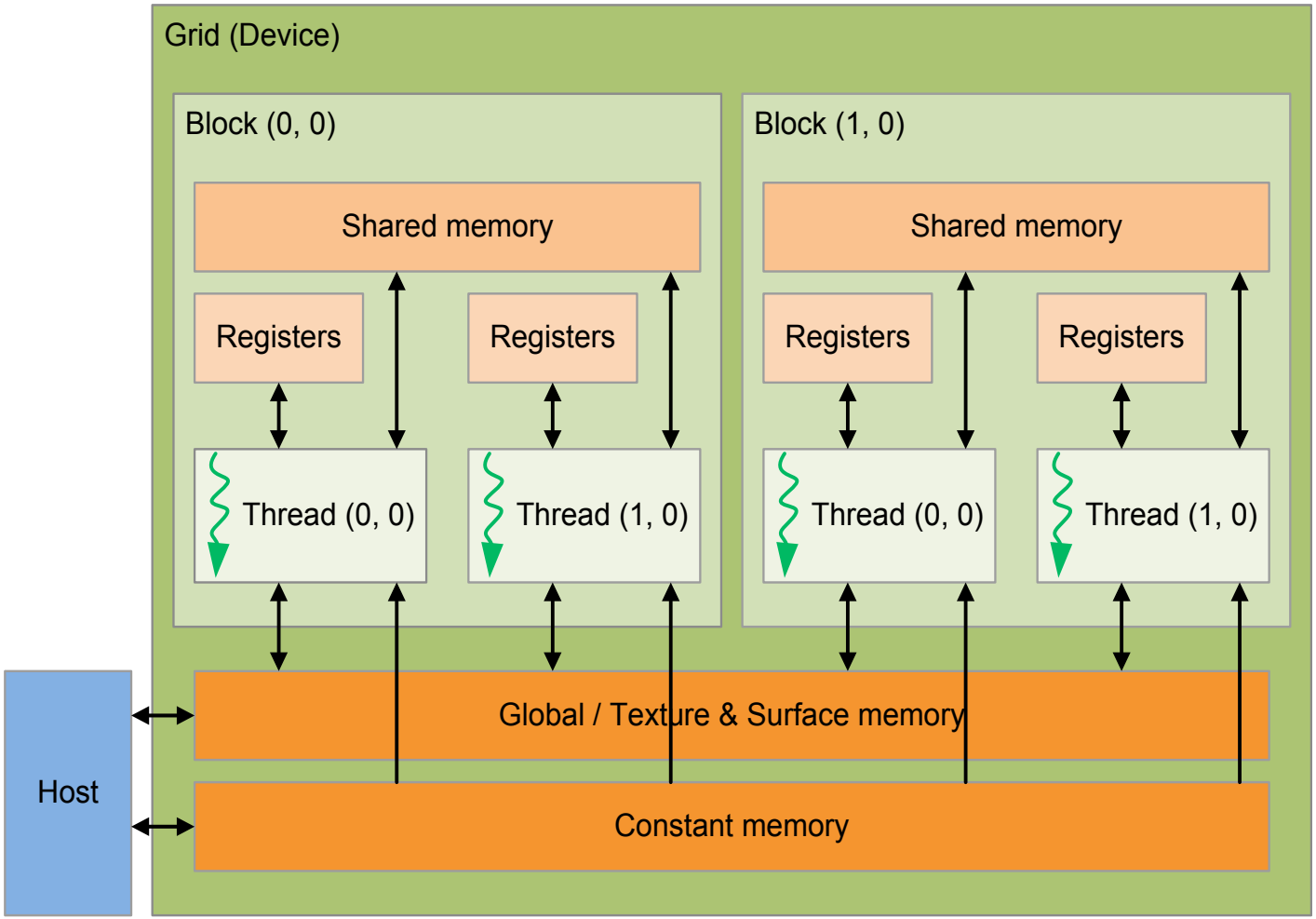
with Boundary Condition Handling*

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
                                            int Mask_Width, int Width) {

  int i = blockIdx.x * blockDim.x + threadIdx.x; // Index of output element

  float Pvalue = 0;

  int N_start_point = i - (Mask_Width/2); // Index of first neighbor

  for(int j = 0; j < Mask_Width; j++) {

    // Check the boundaries
    if(N_start_point + j >= 0 && N_start_point + j < Width) {

      Pvalue += N[N_start_point + j] * M[j]; // Multiply and accumulate
    }
  }

  P[i] = Pvalue; // Store output element
}
```

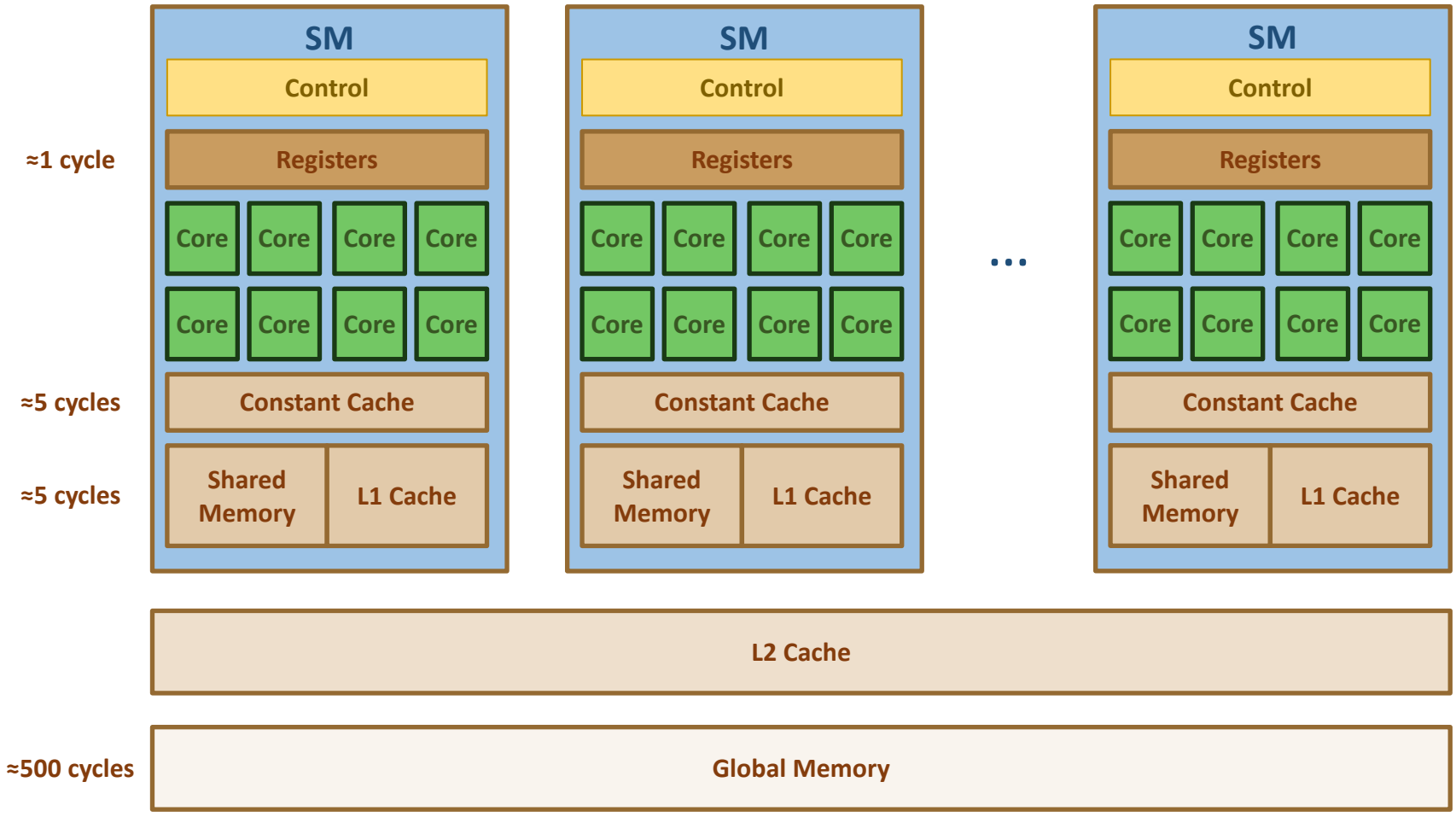


* All elements outside the image are 0 in this kernel

Recall: Memory Hierarchy in CUDA Programs



Memory in the GPU Architecture



Slide credit: Izzat El Hajj

Storing the Mask in Constant Memory

- We can store the mask in constant memory
  - The mask is small
  - It is constant
  - It is accessed by all threads
- Constant memory is cached inside each GPU core and it is particularly fast when all threads of a warp access the same value
- 1. Declare the mask as a global variable

```
#define MASK_WIDTH 5

__constant__ float M[MASK_WIDTH];
```

- 2. Initialize the mask from the host

```
cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

Destination

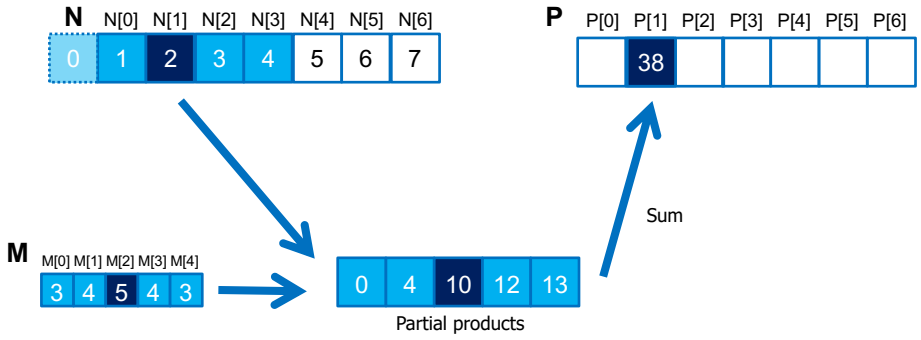
Source

Size

1D Convolution Kernel

with Boundary Condition Handling and Constant Cache for `M`

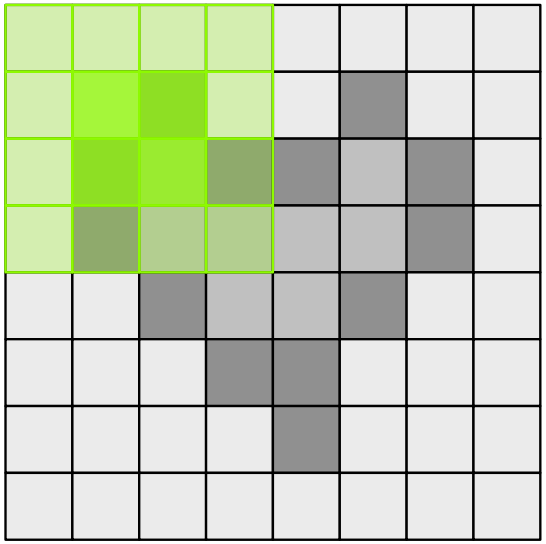
```
__global__ void convolution_1D_basic_kernel(float *N, float *P,
                                            int Mask_Width, int Width) {

  int i = blockIdx.x * blockDim.x + threadIdx.x; // Index of output element

  float Pvalue = 0;

  int N_start_point = i - (Mask_Width/2); // Index of first neighbor

  for (int j = 0; j < Mask_Width; j++) {

    // Check the boundaries
    if (N_start_point + j >= 0 && N_start_point + j < Width) {

      Pvalue += N[N_start_point + j] * M[j]; // Multiply and accumulate
    }
  }

  P[i] = Pvalue; // Store output element
}
```



Use it with caution! Constant cache is small (e.g., 64 KB)

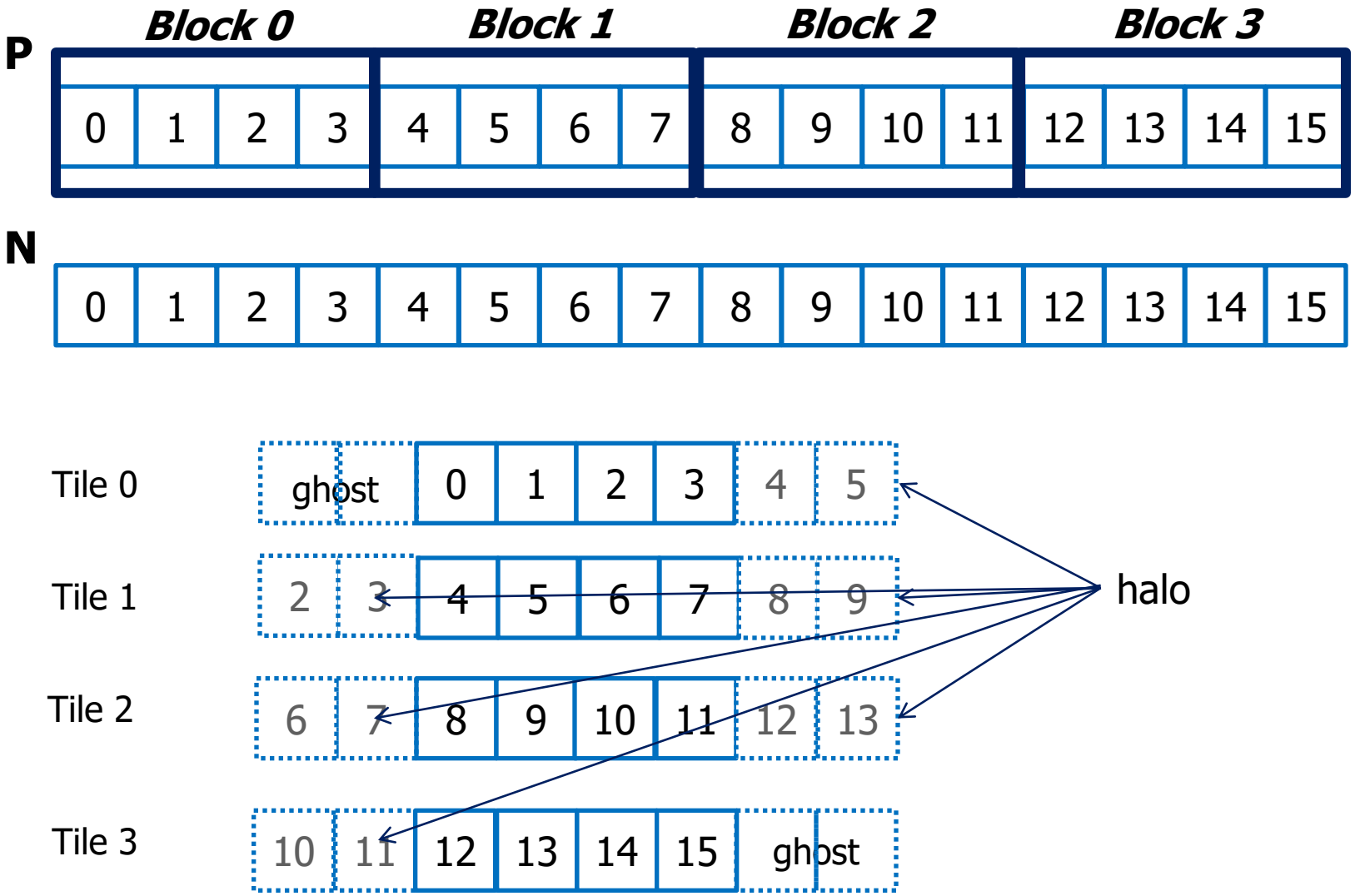
Recall: Data Reuse: Tiling

- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory



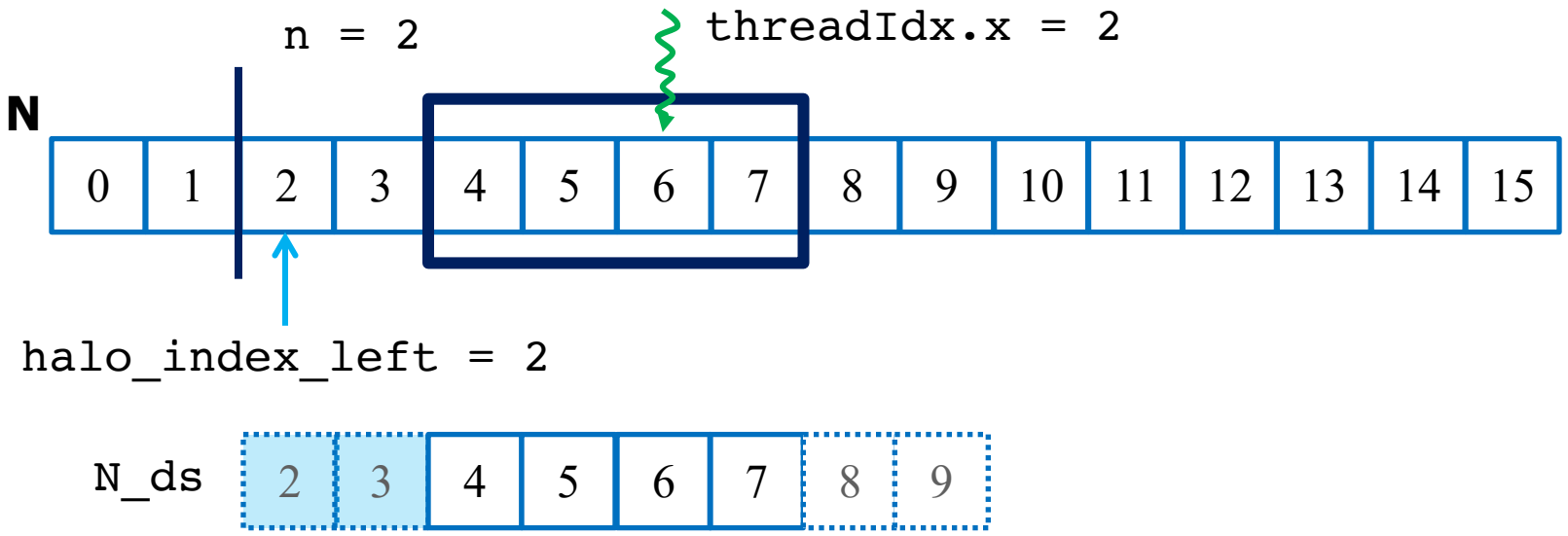
```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Tiled 1D Convolution Basic Idea



Slide credit: Hwu & Kirk

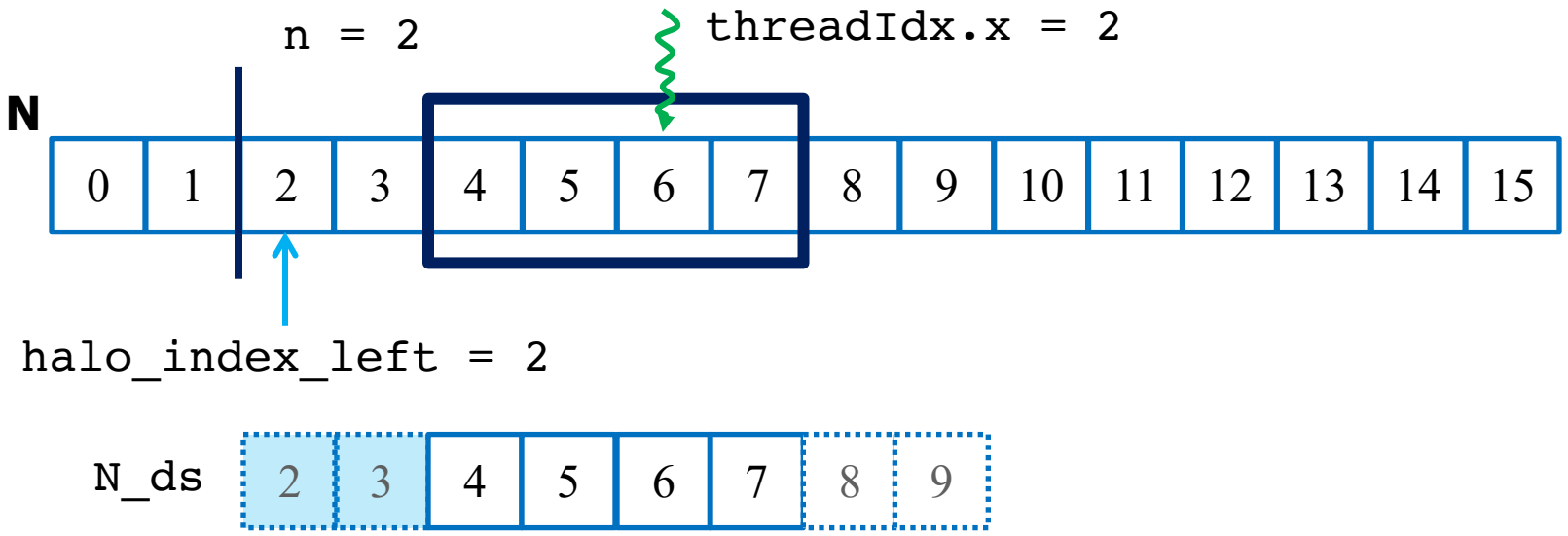
Loading the Left Halo




```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;

if (threadIdx.x >= blockDim.x - n) {

    N_ds[threadIdx.x - (blockDim.x - n)] =
                (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

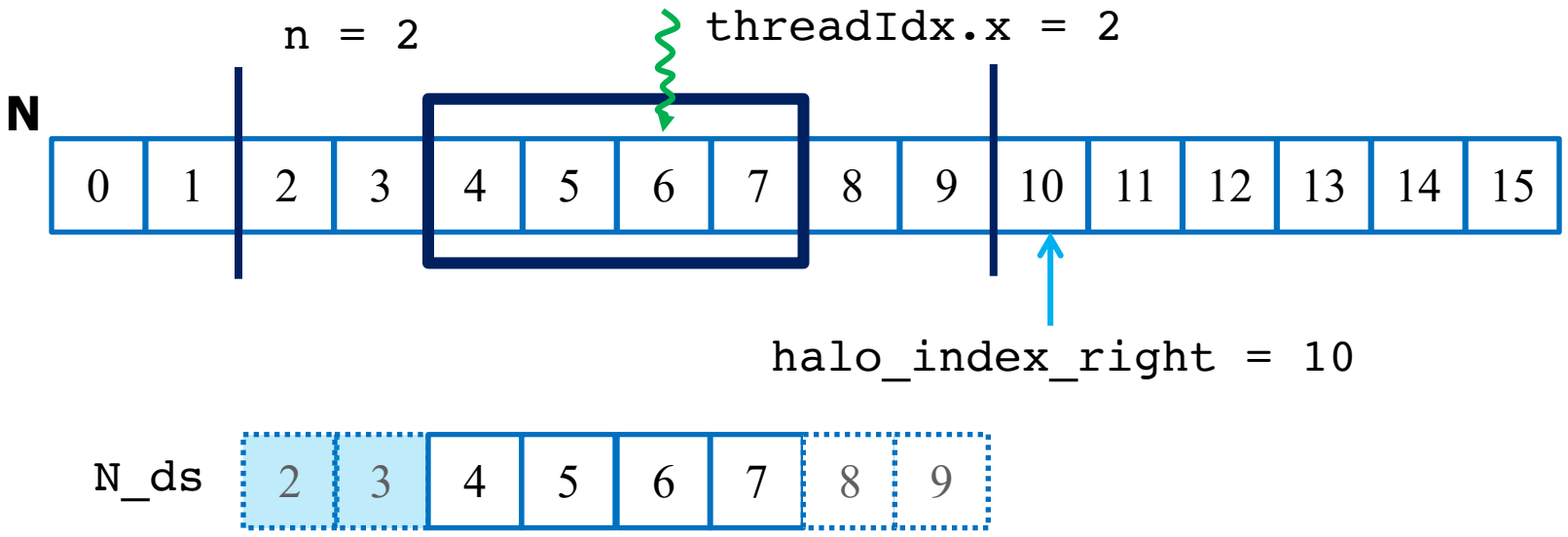
Slide credit: Hwu & Kirk

Loading the Internal Elements




```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
```

Loading the Right Halo




```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;

if (threadIdx.x < n) {

    N_ds[n + blockDim.x + threadIdx.x] =
           (halo_index_right >= Width) ? 0 : N[halo_index_right];
}
```

Slide credit: Hwu & Kirk

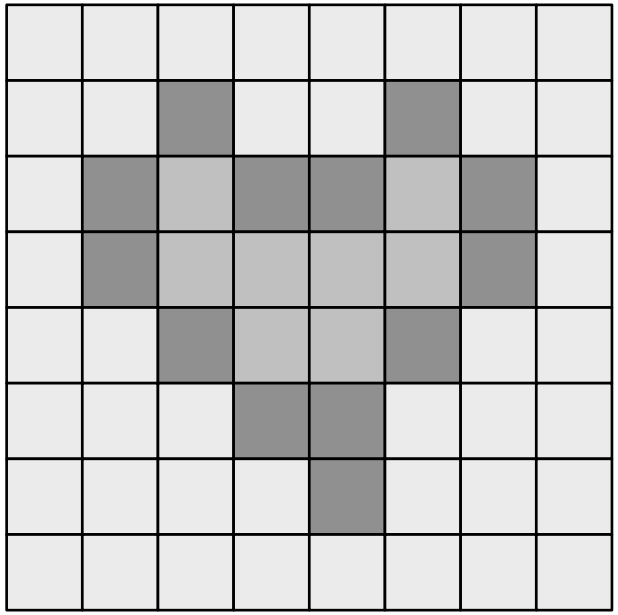
Tiled 1D Convolution Kernel

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;  // Index of output element
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];  // Shared memory tile

  int n = Mask_Width/2;  // Halo width

  // Load left halo
  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if(threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] = (halo_index_left < 0) ? 0 : N[halo_index_left];
  }

  // Load internal elements
  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  // Load right halo
  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if(threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] = (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }

  __syncthreads();  // Intra-block synchronization

  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j] * M[j];  // Convolution result
  }
  P[i] = Pvalue;
}
```

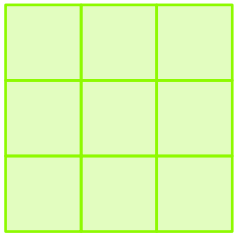
Slide credit: Hwu & Kirk

2D Convolution (I)

- The mask is 2D
  - For example, a Gaussian filter

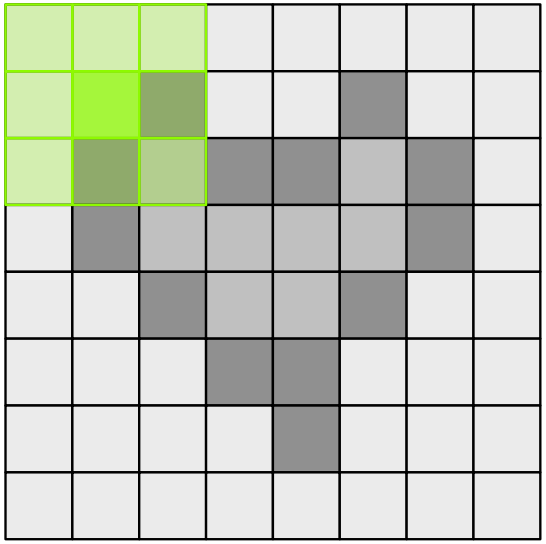


Gaussian mask



2D Convolution (II)

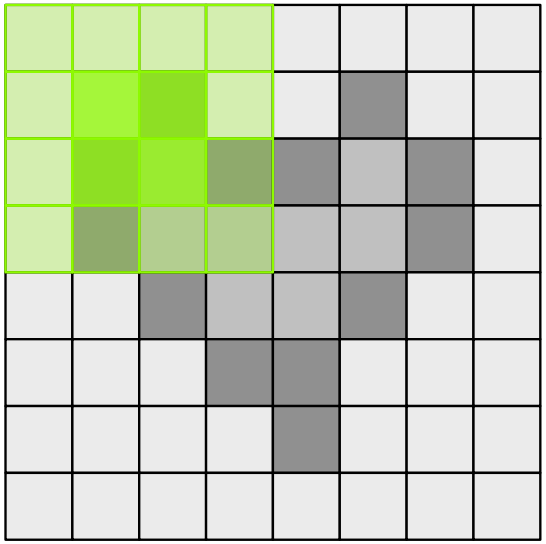
- Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

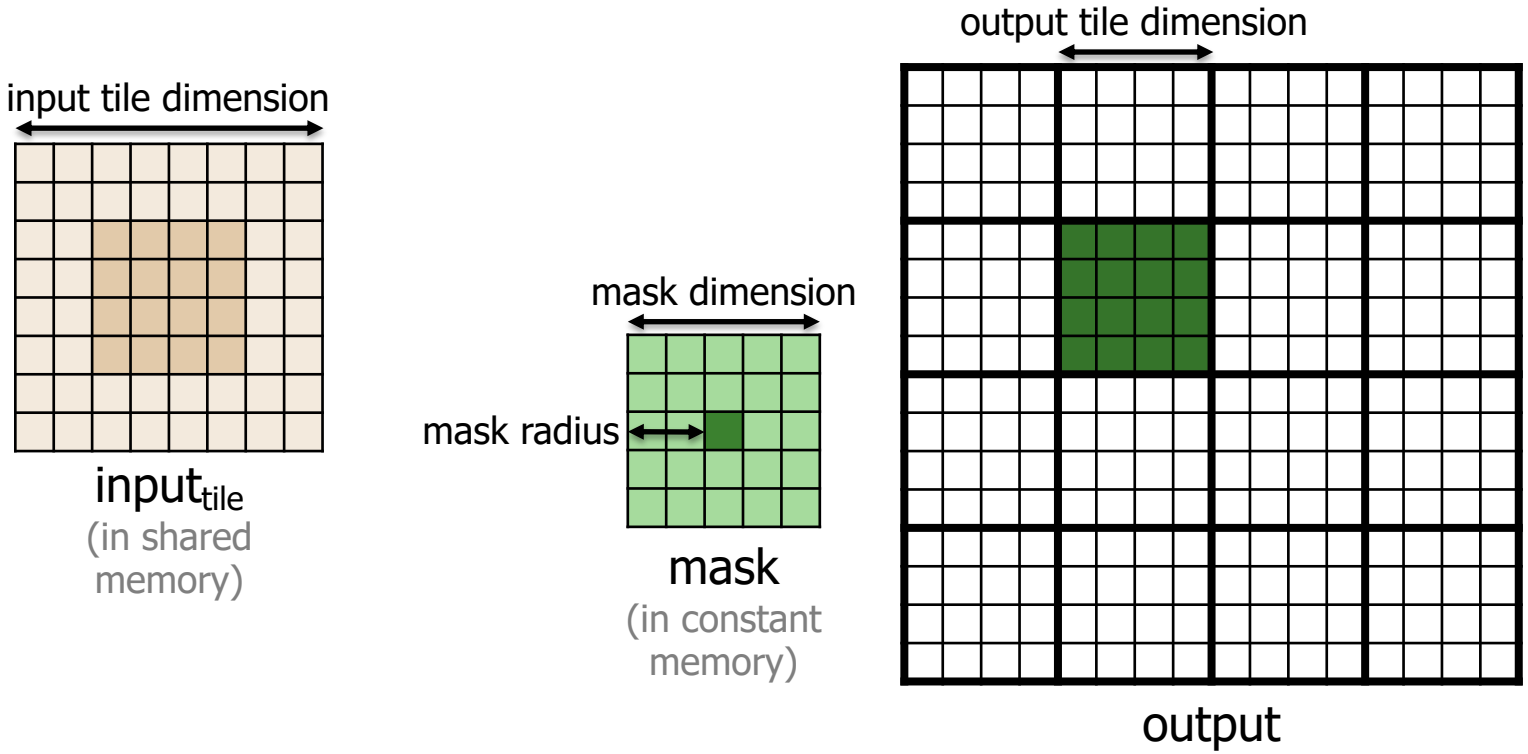
Tiling in 2D Convolution

- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Loading Tiles into Shared Memory



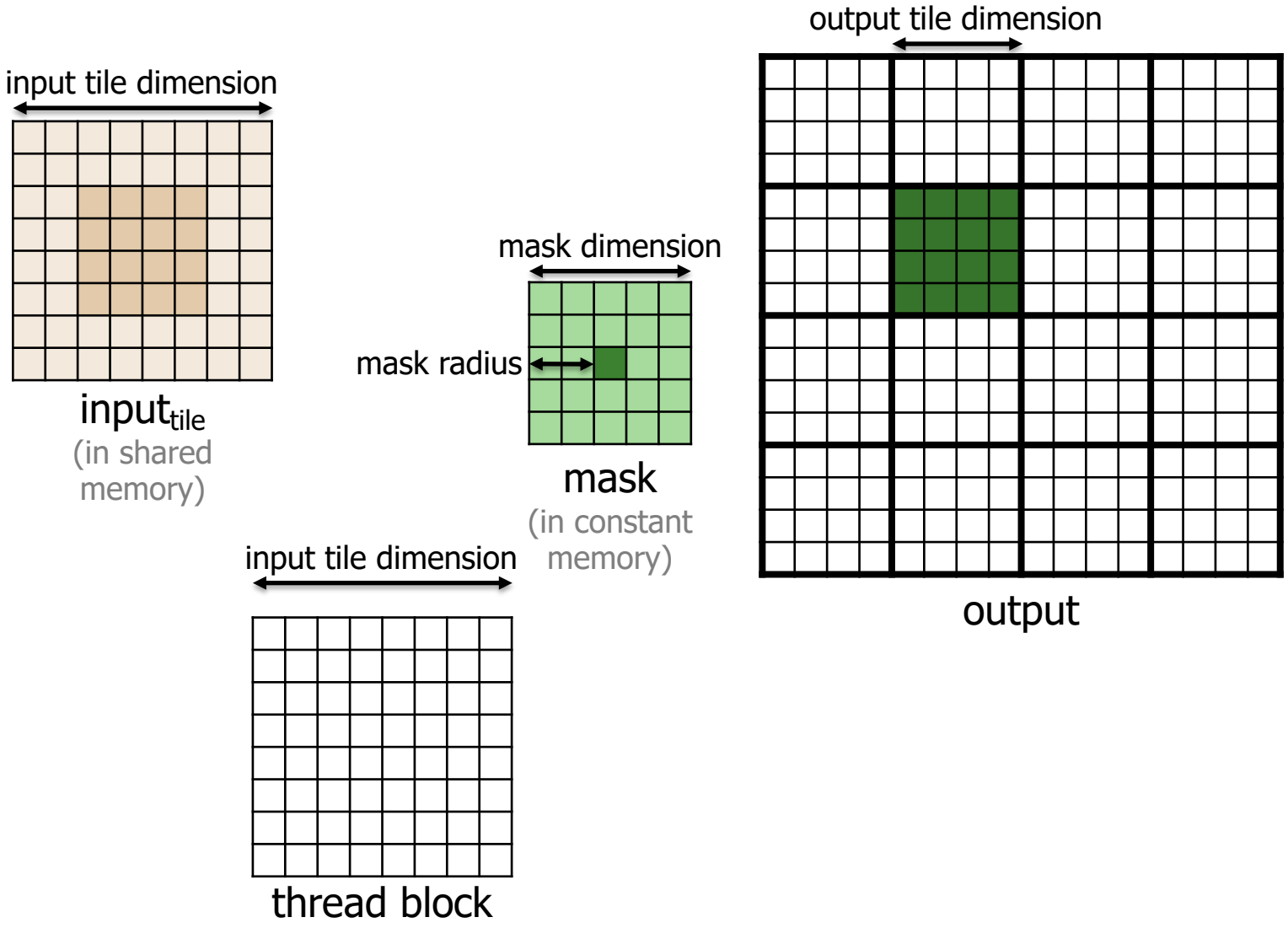
Challenge: Input and output tiles have different dimensions

(input tile dimension = output tile dimension + 2 × mask radius)

Solution: Launch enough threads per block to load the input tile to shared memory, then use a subset of them to compute and store the output tile

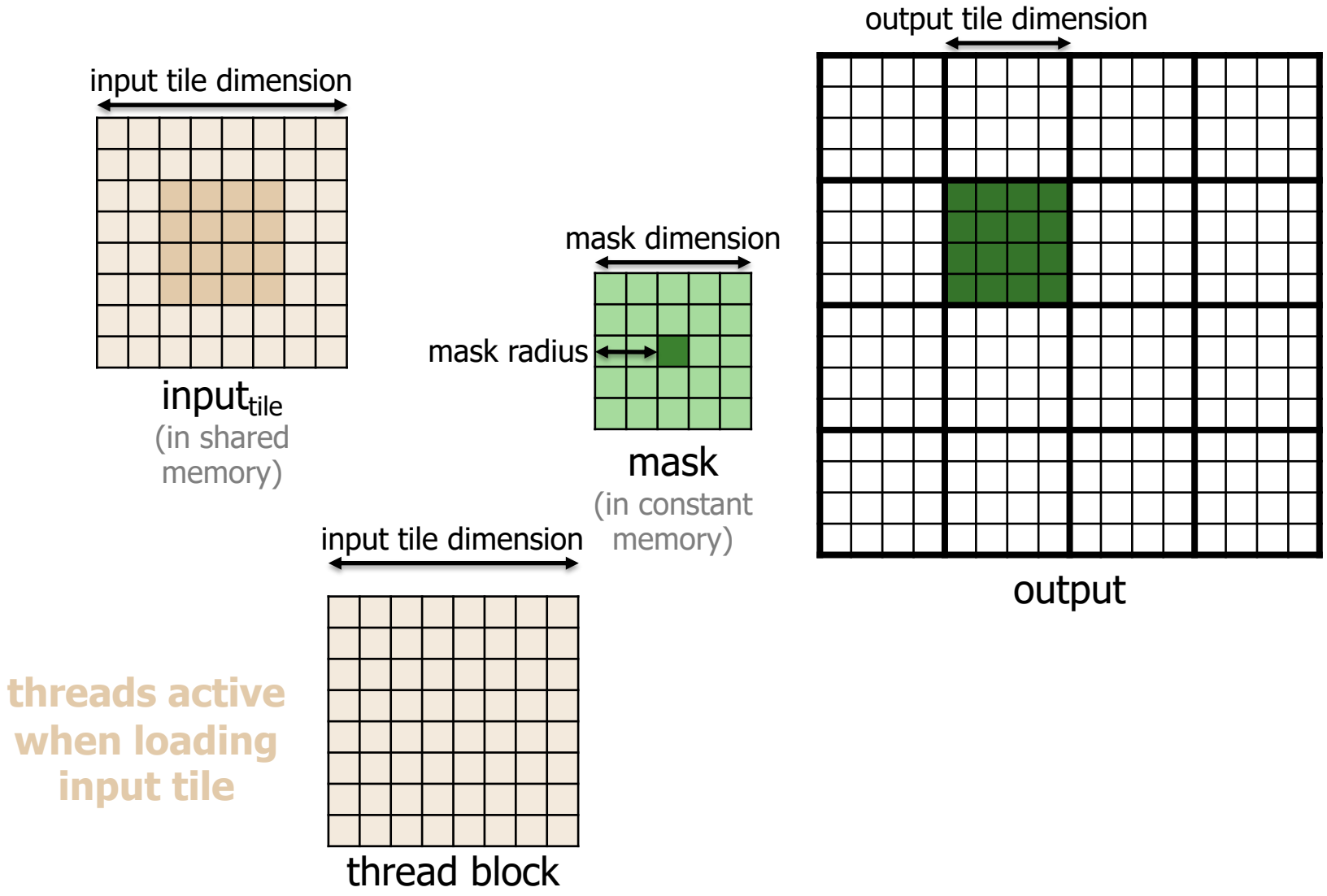
Slide credit: Izzat El Hajj

Difference in Tile Sizes (I)



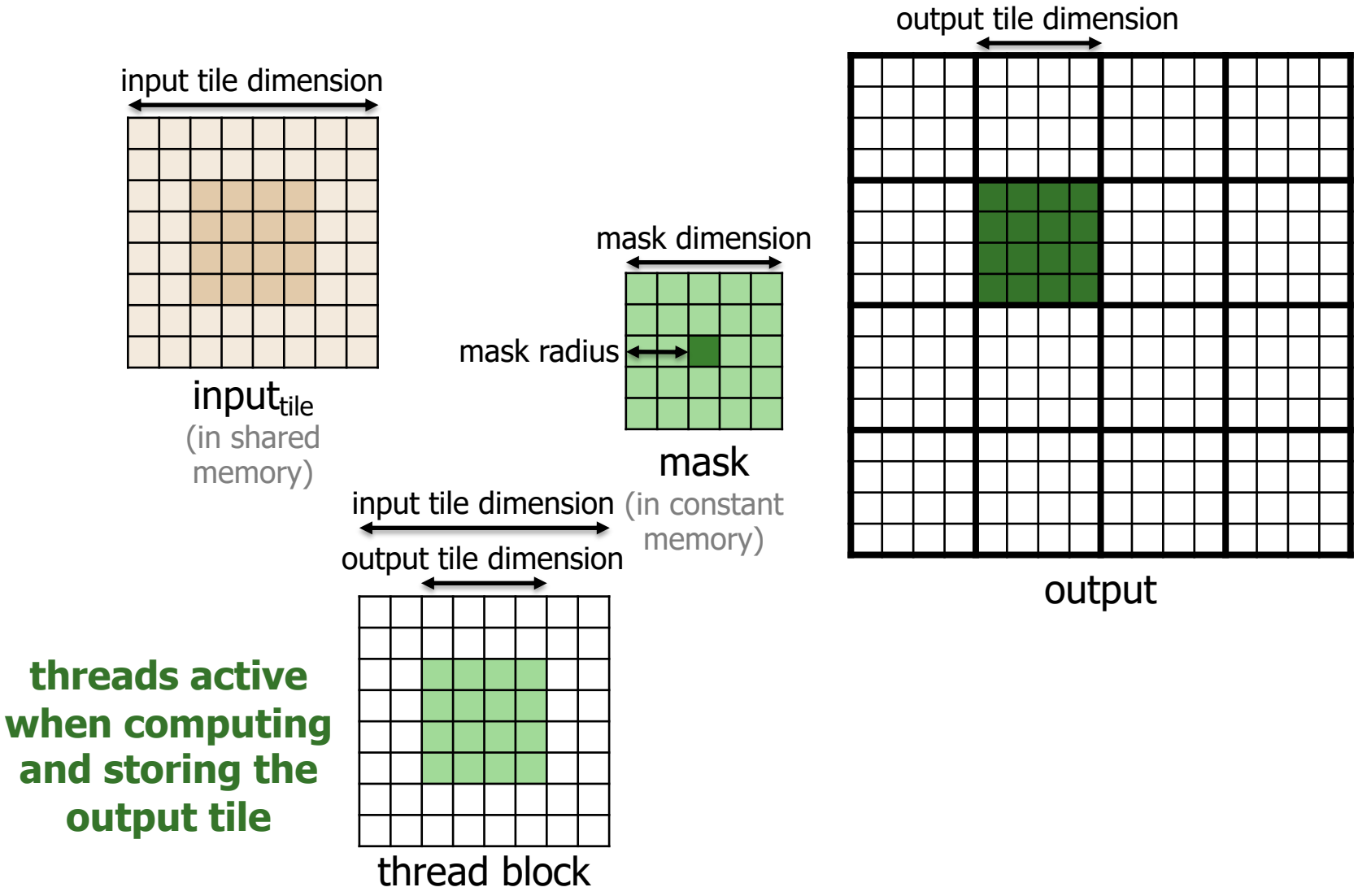
Slide credit: Izzat El Hajj

Difference in Tile Sizes (II)



Slide credit: Izzat El Hajj

Difference in Tile Sizes (III)



threads active when computing and storing the output tile

Slide credit: Izzat El Hajj

2D Convolution Examples

- Original

- Blur

```
#define filterWidth 5
#define filterHeight 5

double filter[filterHeight][filterWidth] =
{
  0, 0, 1, 0, 0,
  0, 1, 1, 1, 0,
  1, 1, 1, 1, 1,
  0, 1, 1, 1, 0,
  0, 0, 1, 0, 0,
};
```

- Motion blur

```
#define filterWidth 9
#define filterHeight 9

double filter[filterHeight][filterWidth] =
{
  1, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 1, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 1, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 1, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 1, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 1, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 1,
};
```

- Sharpen

```
#define filterWidth 3
#define filterHeight 3

double filter[filterHeight][filterWidth] =
{
  -1, -1, -1,
  -1,  9, -1,
  -1, -1, -1
};
```

https://lodev.org/cgtutor/filtering.html

Canny Edge Detection

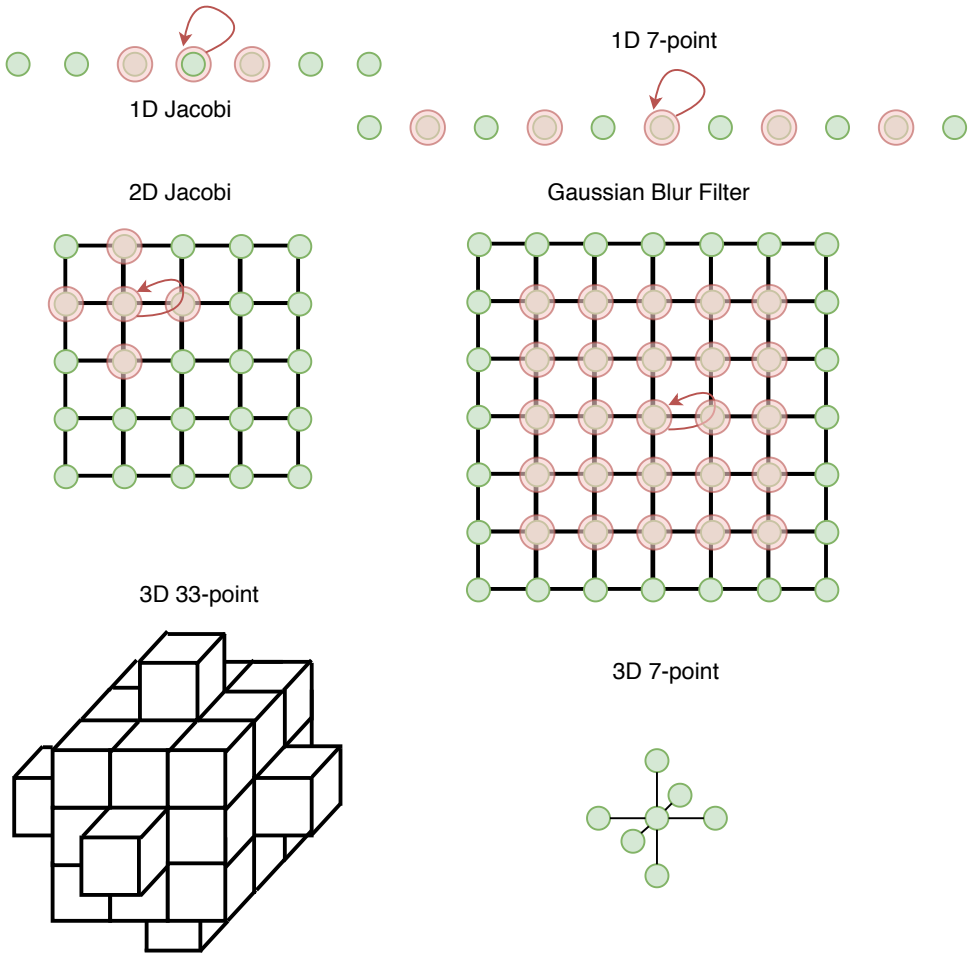
- Gaussian filtering
  - Smooth the image to remove noise
- Sobel filtering
  - Find the intensity gradients of the image
- Non-maximum suppression
  - Suppress spurious response to edge detection
- Hysteresis threshold
  - Strong edges + weak edges connected to a strong edge

Canny, "A Computational Approach To Edge Detection," IEEE TPAMI, 1986

Convolutions are Stencils

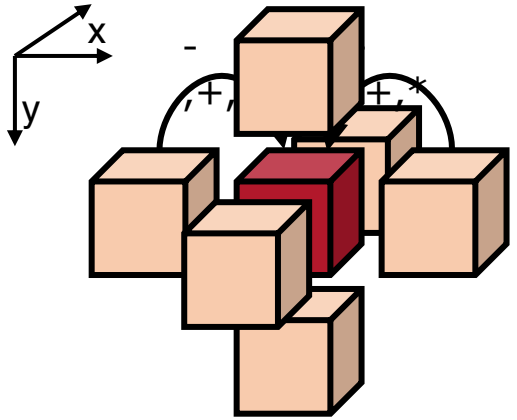
Stencil Computation

- Stencils are a class of algorithms where each output element is calculated from a set of neighboring input elements in a structured grid
  - Widely-used in high performance computing to solve partial differential equations (PDEs)

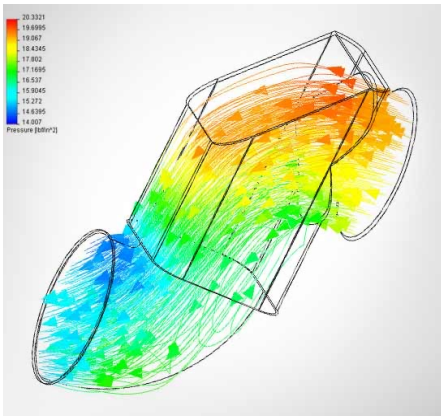


Stencil Computation and Applications

- Stencil computations update values in a grid using a fixed pattern of grid points
- Stencils are used in ~30% of high-performance computing applications



7-point Jacobi in 3D plane



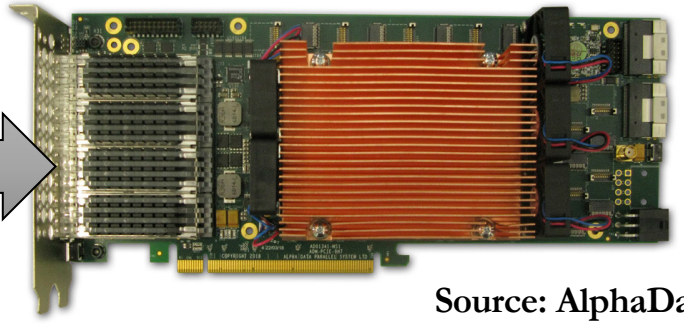
http://www.flometrics.com/fluid-dynamics/computational-fluid-dynamics

Naoe et al., "Secure Key Generation for Static Visual Watermarking by Machine Learning in Intelligent Systems and Services," IJSSOE, 2010

Heterogeneous System: CPU+FPGA

POWER9 AC922

CAPI2



HBM-based AD9H7 board

We evaluate two POWER9+FPGA systems:

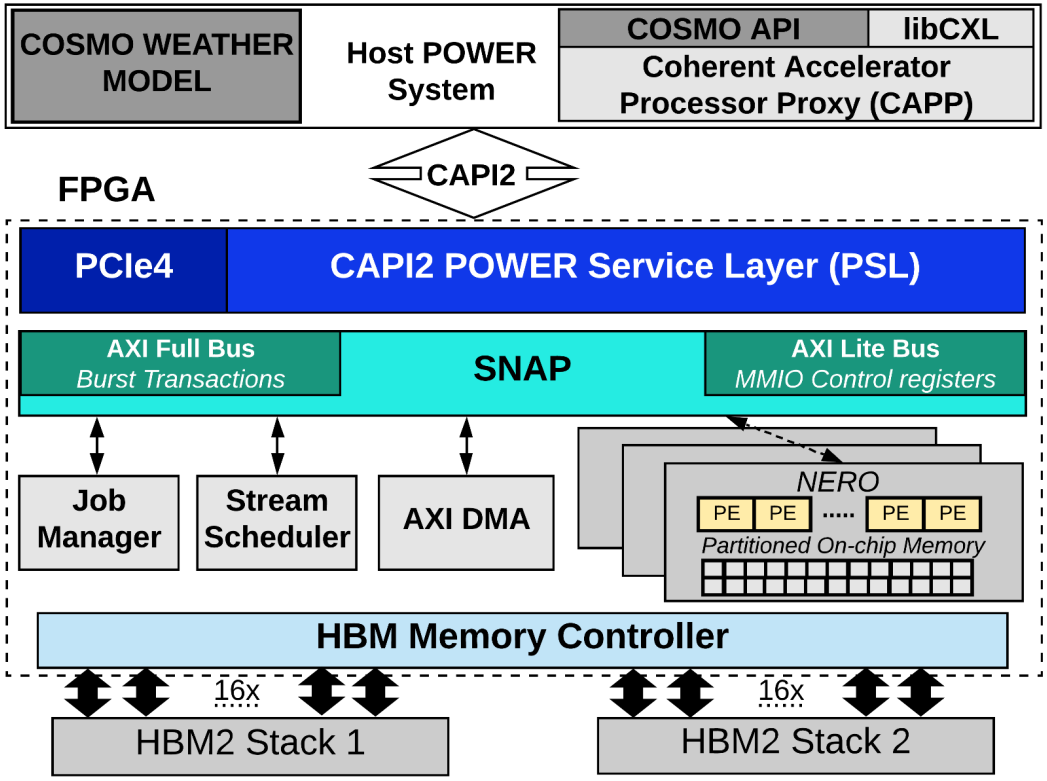
1. HBM-based board AD9H7
Xilinx Virtex Ultrascale+™ XCVU37P-2

2. DDR4-based board AD9V3
Xilinx Virtex Ultrascale+™ XCVU3P-2

Singh et al., NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling, FPL 2020

NERO Application Framework

- NERO communicates to Host over **CAPI2** (Coherent Accelerator Processor Interface)
- **COSMO API** handles offloading jobs to NERO
- **SNAP** (Storage, Network, and Analytics Programming) allows for seamless integration of the COSMO API



https://github.com/open-power/snap

Singh et al., NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling, FPL 2020

Accelerating Climate Modeling

- Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal,
  "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling"
  *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (**FPL**)*, Gothenburg, Sweden, September 2020.
  [Slides (pptx) (pdf)]
  [Lightning Talk Slides (pptx) (pdf)]
  [Talk Video (23 minutes)]
  Nominated for the Stamatis Vassiliadis Memorial Award.

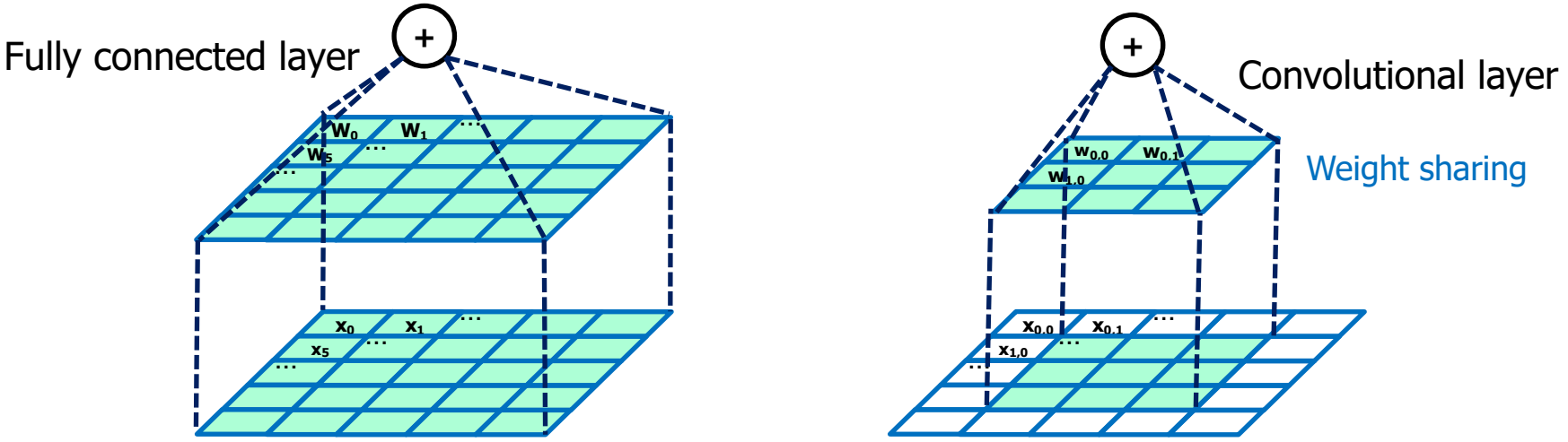
NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling

Gagandeep Singh[a,b,c] Dionysios Diamantopoulos[c] Christoph Hagleitner[c] Juan Gómez-Luna[b]
Sander Stuijk[a] Onur Mutlu[b] Henk Corporaal[a]
[a]Eindhoven University of Technology [b]ETH Zürich [c]IBM Research Europe, Zurich

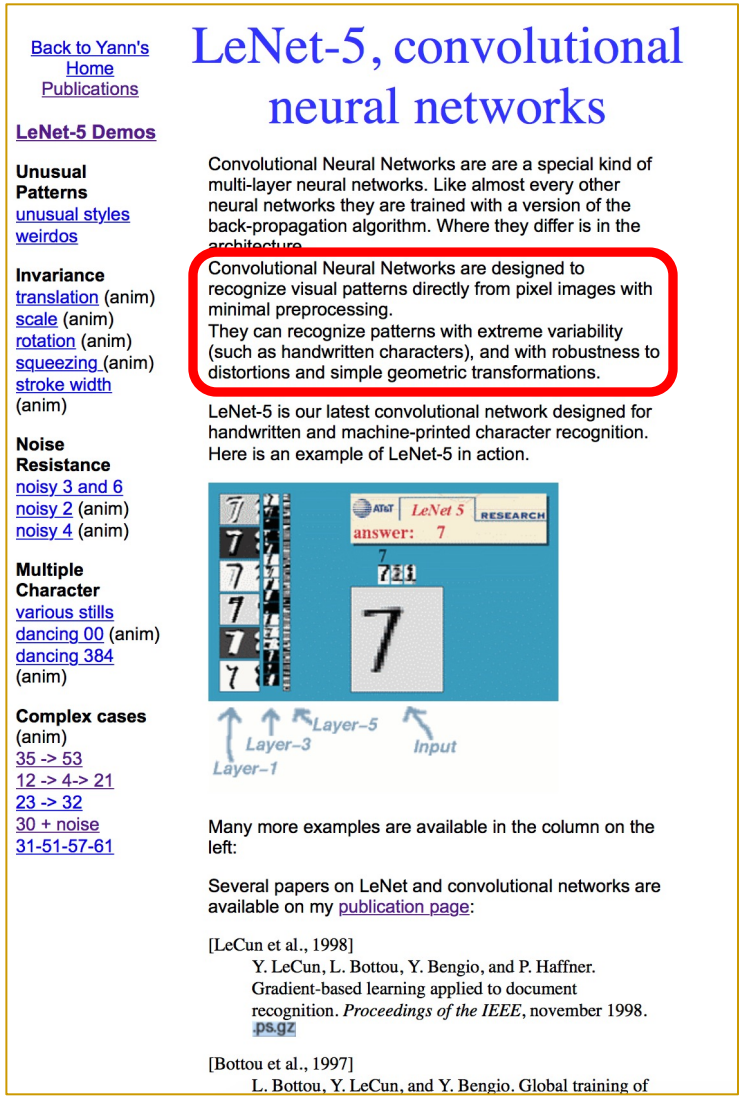
Convolutions in Machine Learning

Convolutions in Machine Learning

- Convolutions are traditionally used for feature detection in image processing
  - They can be used as neural network layers
- Convolutions have an advantage over fully connected layers (e.g., in multilayer perceptron, MLP)
  - Local weights: They compute on only a window around the element of interest
  - Data sharing via on-chip memories is feasible



Convolutional Neural Networks: Demo



Back to Yann's Home Publications

LeNet-5 Demos

Unusual Patterns
unusual styles
weirdos

Invariance
translation (anim)
scale (anim)
rotation (anim)
squeezing (anim)
stroke width (anim)

Noise Resistance
noisy 3 and 6
noisy 2 (anim)
noisy 4 (anim)

Multiple Character
various stills
dancing 00 (anim)
dancing 384 (anim)

Complex cases (anim)
35 -> 53
12 -> 4-> 21
23 -> 32
30 + noise
31-51-57-61

LeNet-5, convolutional neural networks

Convolutional Neural Networks are are a special kind of multi-layer neural networks. Like almost every other neural networks they are trained with a version of the back-propagation algorithm. Where they differ is in the architecture.

Convolutional Neural Networks are designed to recognize visual patterns directly from pixel images with minimal preprocessing.
They can recognize patterns with extreme variability (such as handwritten characters), and with robustness to distortions and simple geometric transformations.

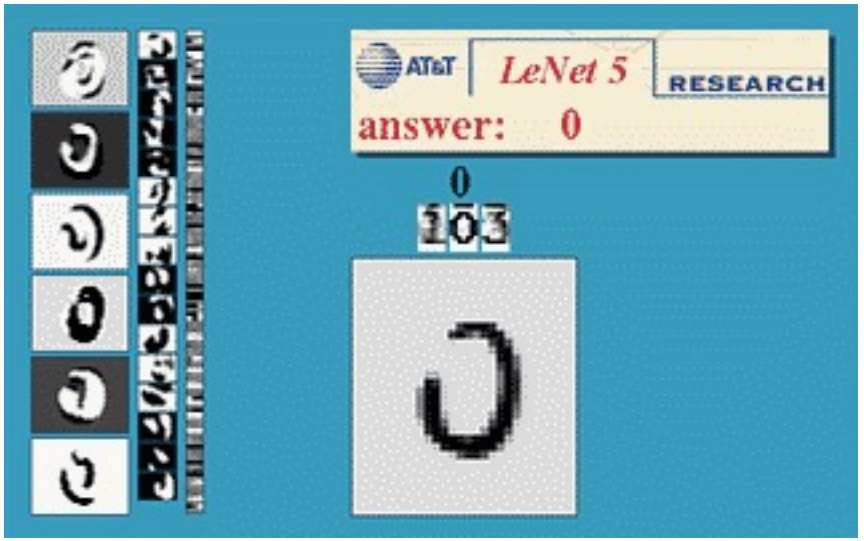
LeNet-5 is our latest convolutional network designed for handwritten and machine-printed character recognition. Here is an example of LeNet-5 in action.

Many more examples are available in the column on the left:

Several papers on LeNet and convolutional networks are available on my publication page:

[LeCun et al., 1998]
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998. ps.gz

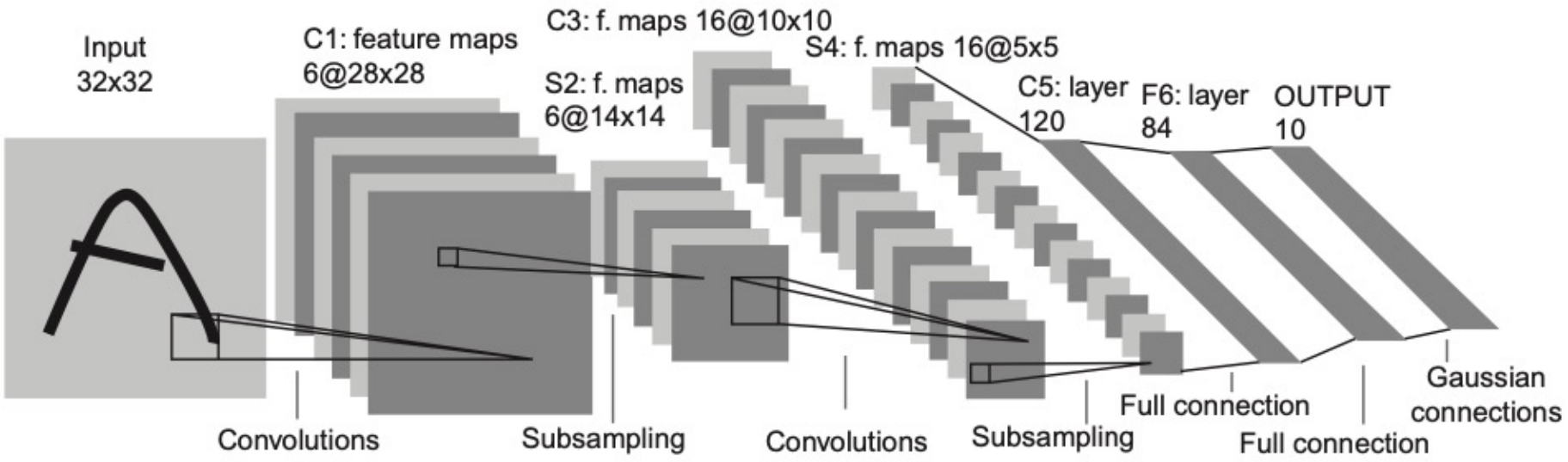
[Bottou et al., 1997]
L. Bottou, Y. LeCun, and Y. Bengio. Global training of



http://yann.lecun.com/exdb/lenet/index.html

LeNet-5, a Convolutional Neural Network for Hand-Written Digit Recognition

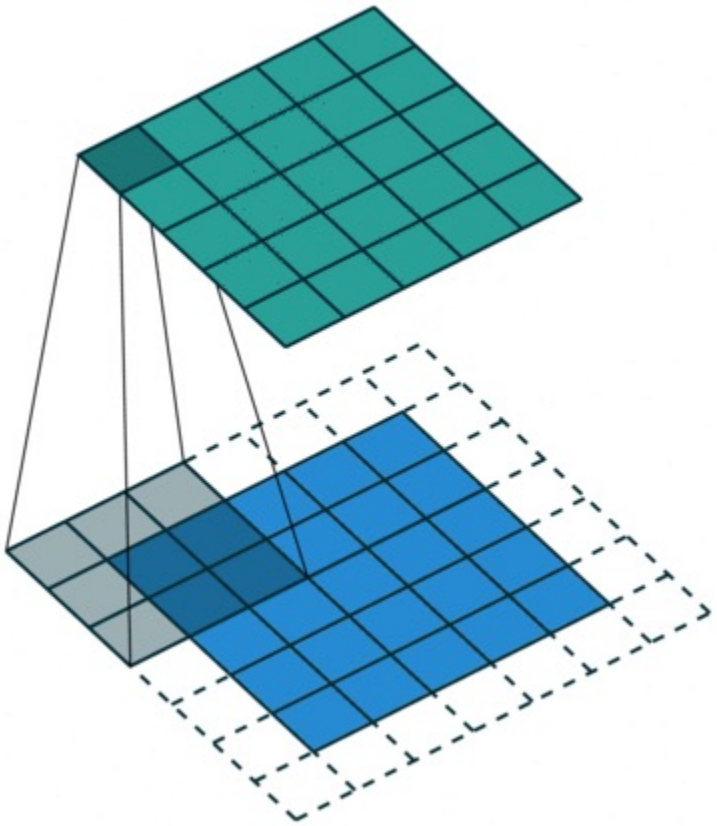
This is a 1024*8 bit input, which will have a truth table of 2^{8196} entries



Slide credit: Hwu & Kirk

An Example of 2D Convolution

Output feature map



Input feature map

Structure information

Input: 5*5 (blue)

Kernel (filter): 3*3 (grey)

Output: 5*5 (green)

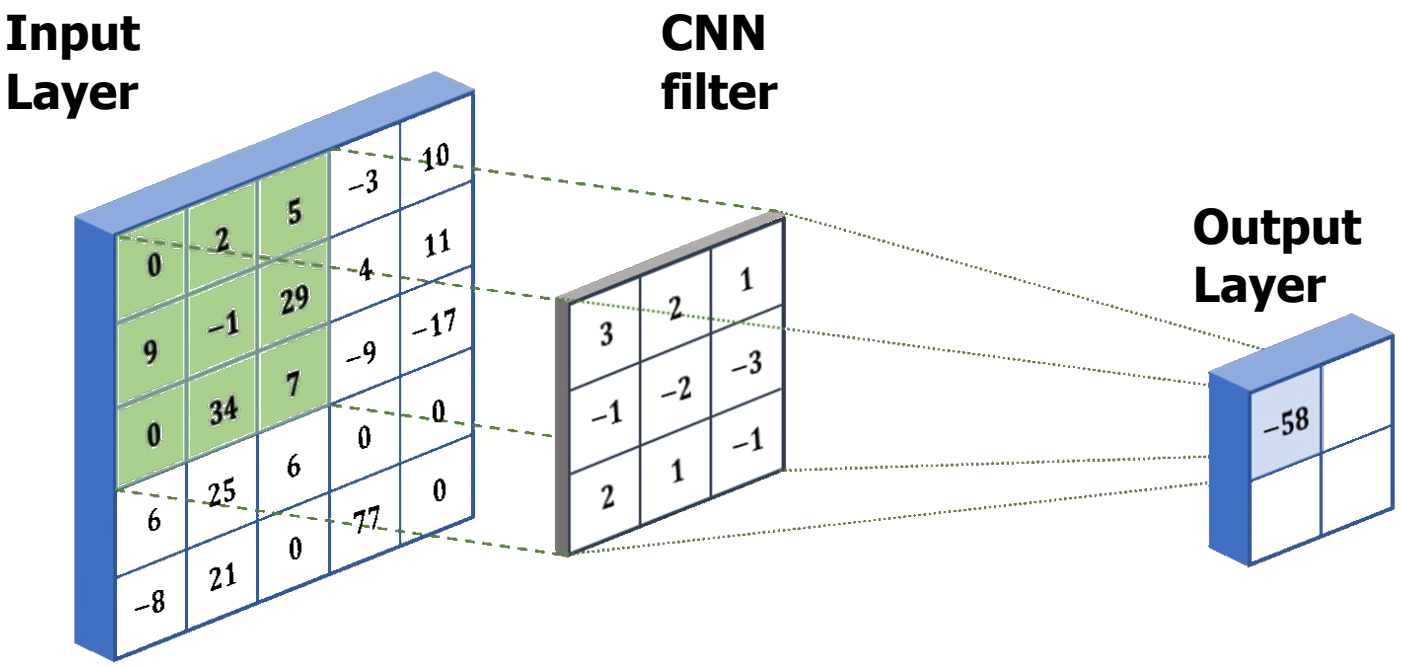
Computation information

Stride: 1

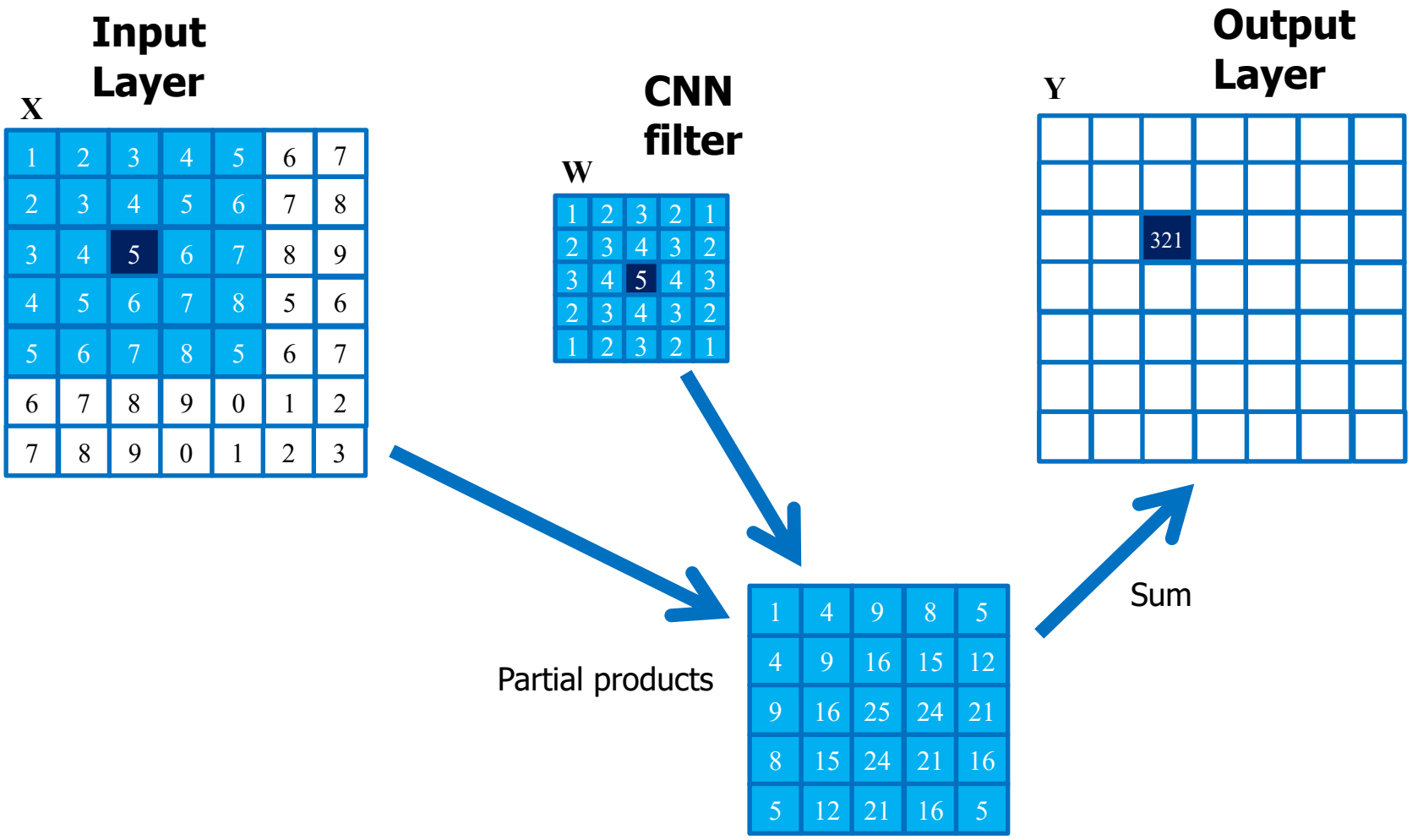
Padding: 1 (white)

Output Dim = (Input + 2*Padding - Kernel) / Stride + 1

An Example of 2D Convolution



Another Example of 2D Convolution



Slide credit: Hwu & Kirk

A Basic Conv. Layer Forward Kernel

(Code is Incomplete!)

```
__global__ void ConvLayerForward_Basic_Kernel(int C, int W_grid,
                                              int K, float* X, float* W, float* Y){

    int m =  blockIdx.x;
    int h =  blockIdx.y / W_grid + threadIdx.y;
    int w =  blockIdx.y % W_grid + threadIdx.x;

    float acc = 0.;

    for(int c = 0;  c < C; c++) {     // Sum over all input channels

       for(int p = 0; p < K; p++)     // Loop over KxK  filter

          for(int q = 0; q < K; q++)

             acc += X[c, h + p, w + q] * W[m, c, p, q];
    }

    Y[m, h, w] = acc;
}
```

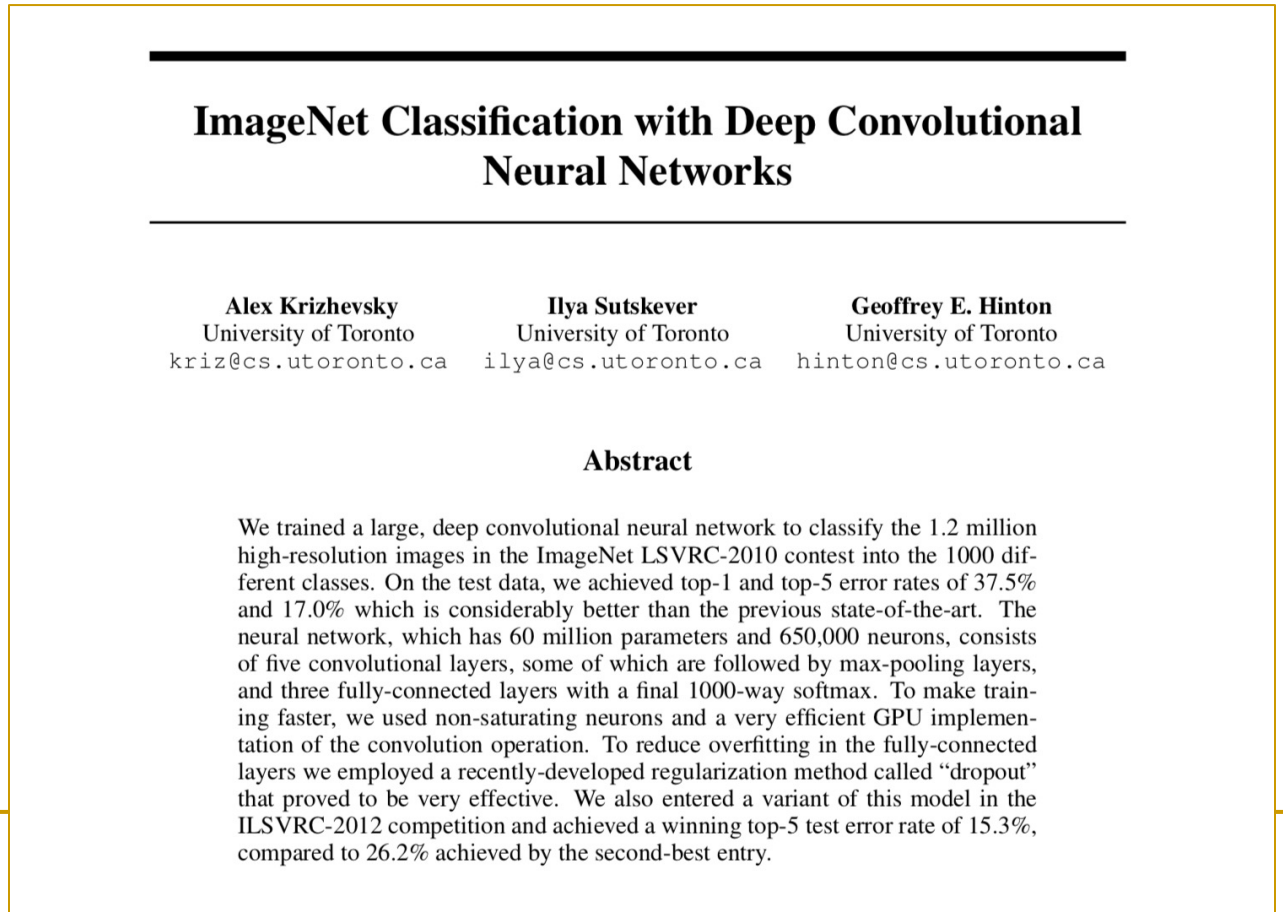
Power of Convolutions and Applied Courses

- In 2010, Prof. Andreas Moshovos adopted Professor Hwu's ECE498AL Programming Massively Parallel Processors Class
- Several of Prof. Geoffrey Hinton's graduate students took the course
- These students developed the GPU implementation of the Deep CNN that was trained with 1.2M images to win the ImageNet competition

Slide credit: Hwu & Kirk

Example: AlexNet (2012)

- AlexNet wins the ImageNet classification competition with ~10% points higher accuracy than state-of-the-art
  - Krizhevsky et al., "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

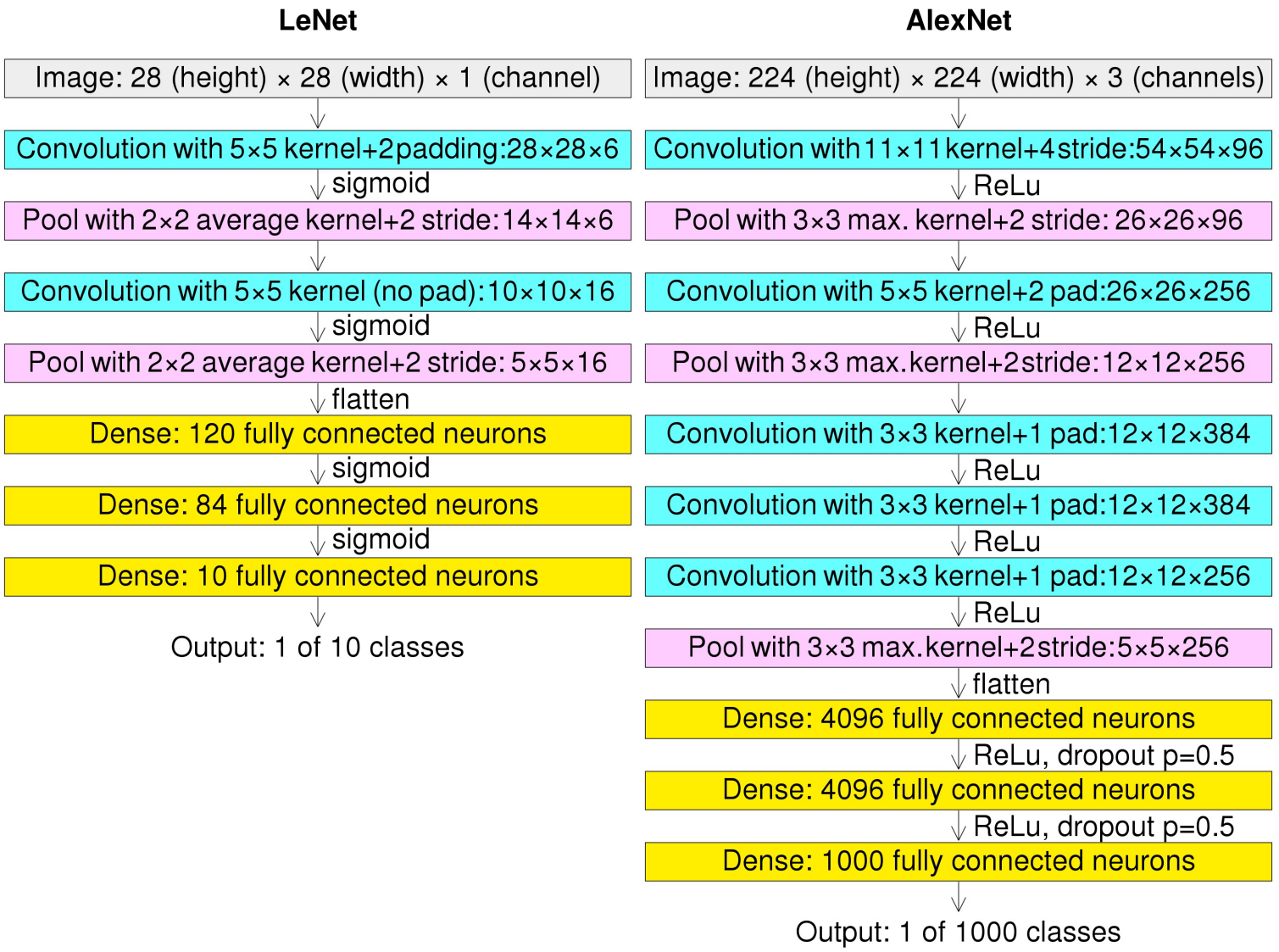
Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

Neural Network Layer Examples



By Cmglee - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=104937230

Example: GoogLeNet (2014)

- Google improves accuracy by adding more network layers
  - From 8 in AlexNet to 22 in GoogLeNet
  - Szegedy et al., "Going Deeper with Convolutions", CVPR 2015.

Going Deeper with Convolutions

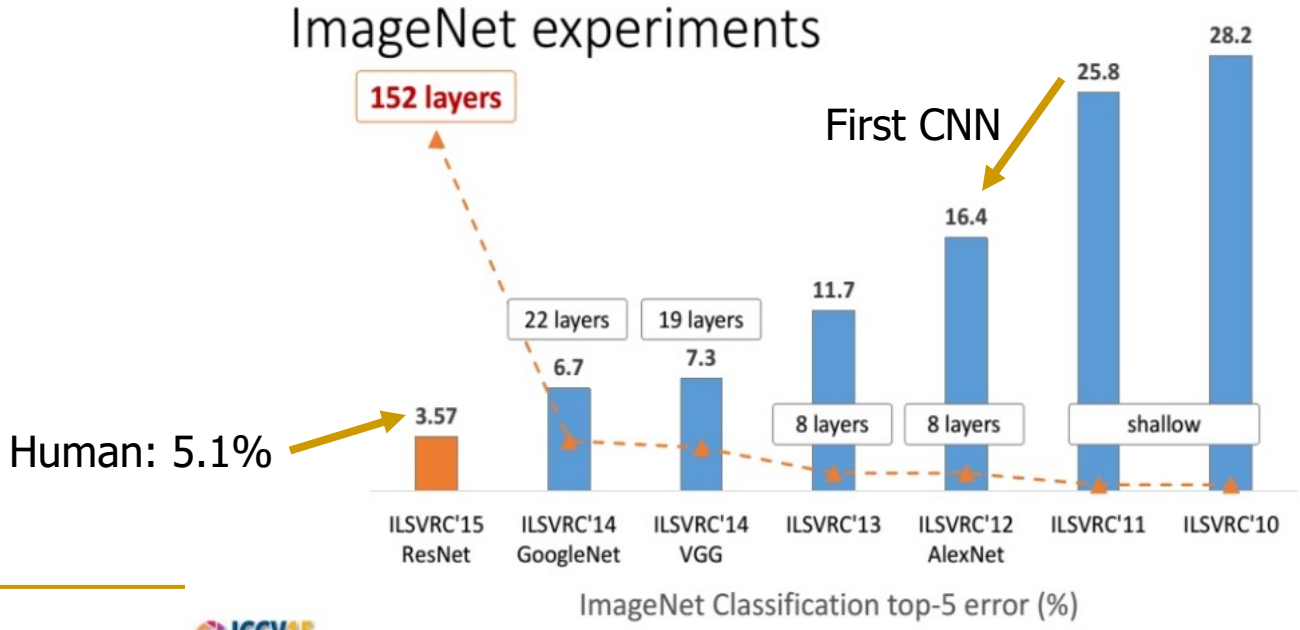
Christian Szegedy[1], Wei Liu[2], Yangqing Jia[1], Pierre Sermanet[1], Scott Reed[3],
Dragomir Anguelov[1], Dumitru Erhan[1], Vincent Vanhoucke[1], Andrew Rabinovich[4]
[1]Google Inc. [2]University of North Carolina, Chapel Hill
[3]University of Michigan, Ann Arbor [4]Magic Leap Inc.
[1]{szegedy,jiayq,sermanet,dragomir,dumitru,vanhoucke}@google.com
[2]wliu@cs.unc.edu, [3]reedscott@umich.edu, [4]arabinovich@magicleap.com

Example: ResNet (2015)

- He et al., "Deep Residual Learning for Image Recognition", CVPR 2016.

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com



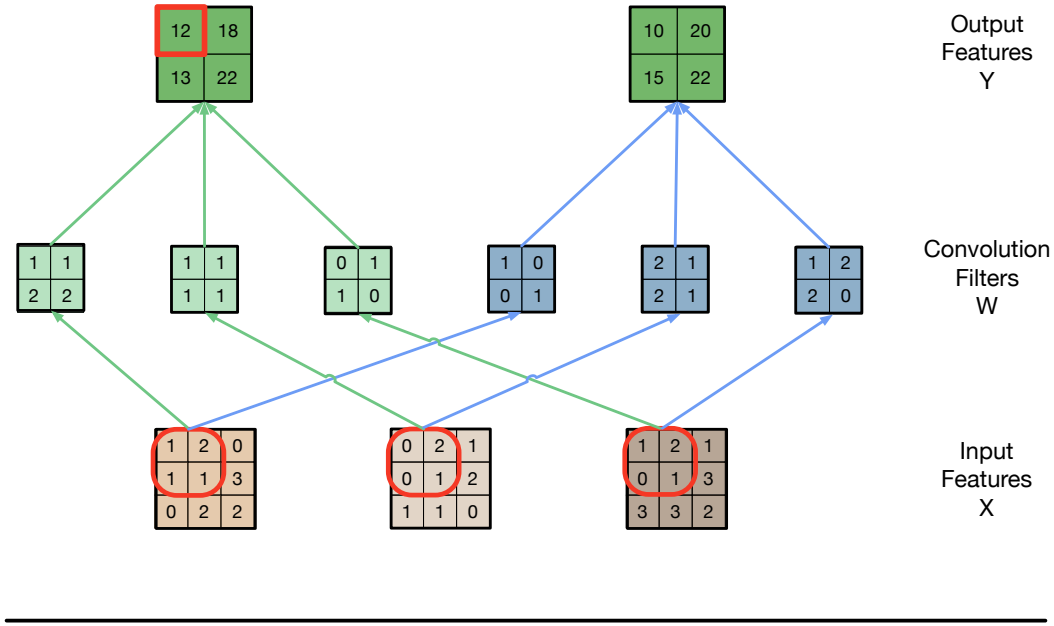
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

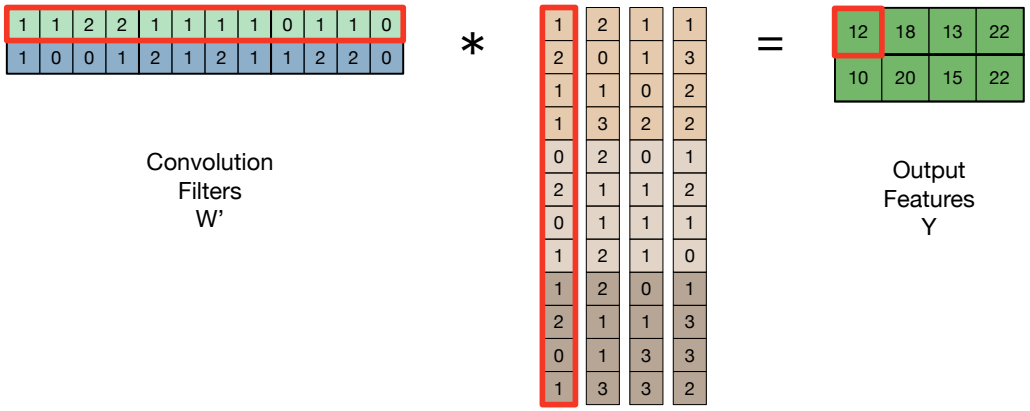
Reducing Convolution Layers to Matrix Multiplications

- Convolution layers are the compute intensive parts of a CNN
- GPUs have extremely high-performance implementations of matrix multiplications
- Tiling techniques for matrix multiplication naturally reuse input features across output feature maps
- Converting convolutions in a convolution layer to a matrix multiplication helps to keep the level of parallelism stable across CNN layers

Slide credit: Hwu & Kirk

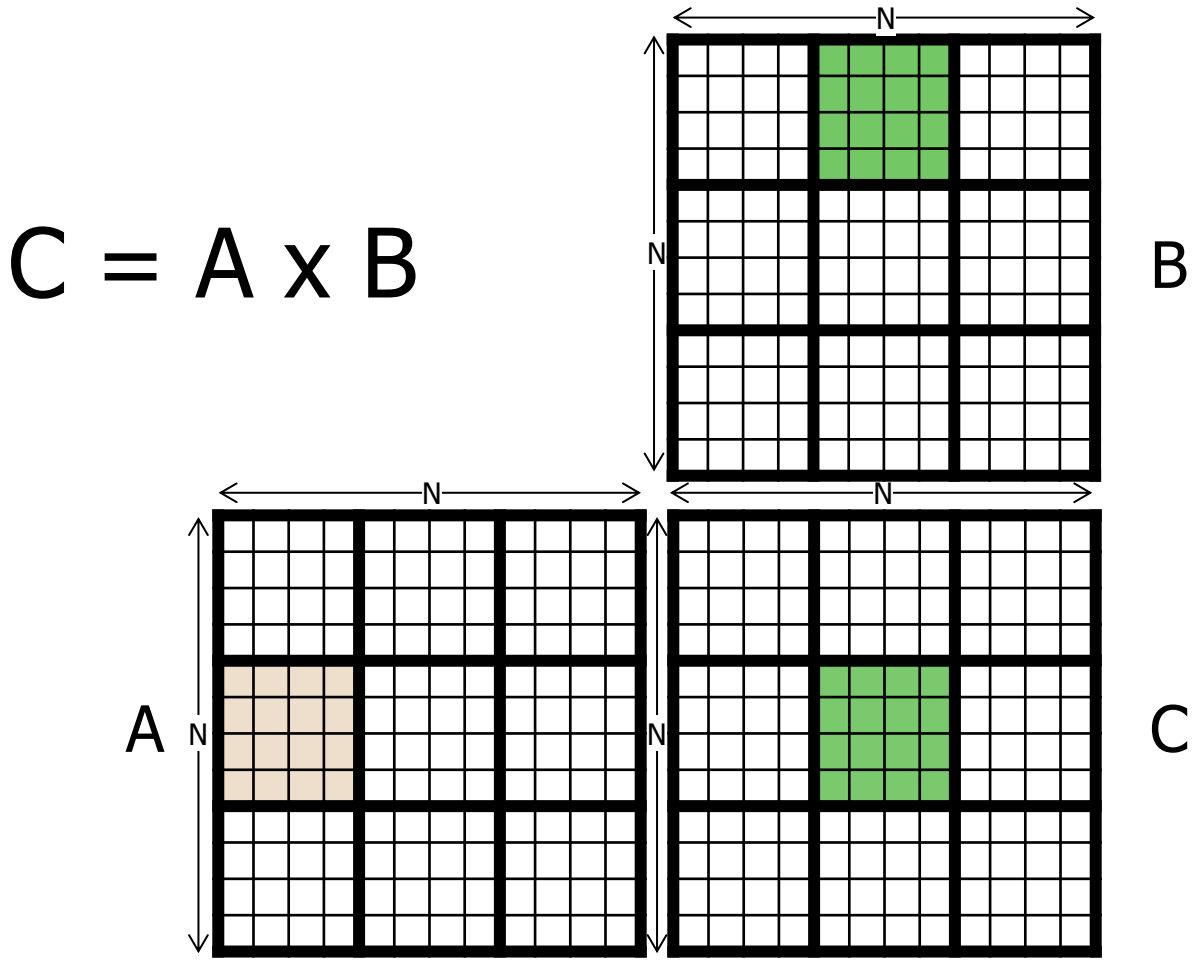
Implementing a Convolutional Layer with Matrix Multiplication





Slide credit: Reproduced from Hwu & Kirk

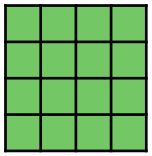
Tiled Matrix-Matrix Multiplication (I)



Step 1: Load the first tile of each input matrix to shared memory (each thread loads one element)

Slide credit: Izzat El Hajj

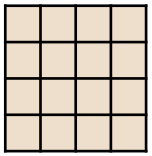
Tiled Matrix-Matrix Multiplication (II)

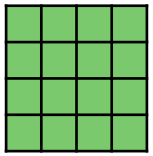


B_{tile}

$$C_{tile} = A_{tile} \times B_{tile}$$

A_{tile}



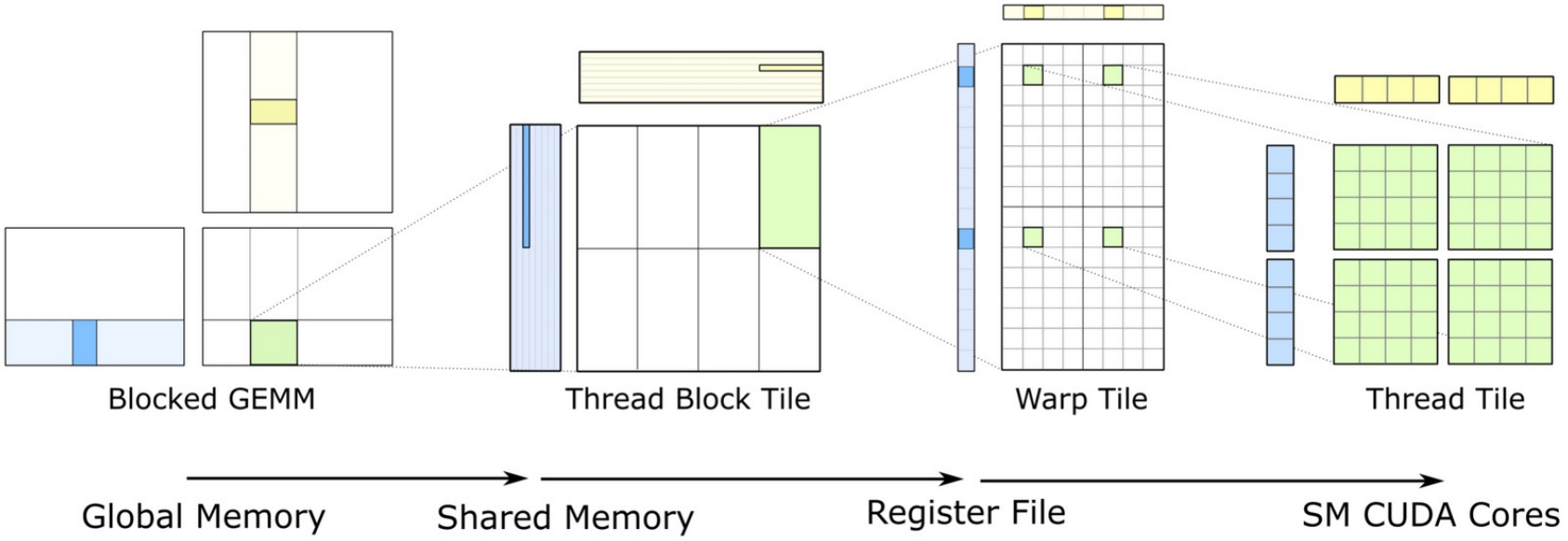


C_{tile}

Step 2: Each thread computes its partial sum from the tiles in shared memory (threads wait for each other to finish)

Slide credit: Izzat El Hajj

Deep Learning Matrix Multiplication Hierarchical Decomposition



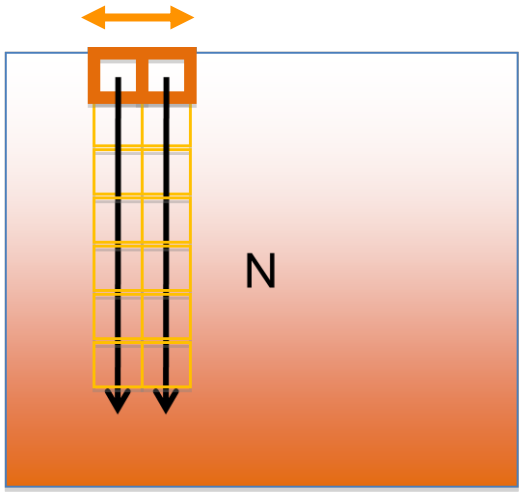
https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive/3

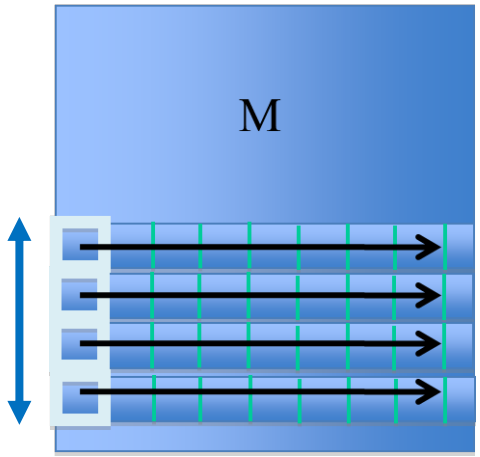
Joint Register and Shared Memory Tiling

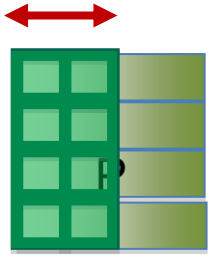
- Store input M tile and output P tile elements in registers
- Store input N tile elements in shared memory
- Decouple of M and N input tile widths
  - TILE_WIDTH_M , TILE_WIDTH_N
- Key quantities
  - Number of threads = TILE_WIDTH_M
  - Output tile size = TILE_WIDTH_M * TILE_WIDTH_N
  - Reuses for each N element = TILE_WIDTH_M
  - Reuses for each M element = TILE_WIDTH_N
  - Each thread calculates TILE_WIDTH_N P elements

Example:

TILE_WIDTH_M = 4
TILE_WIDTH_N = 2





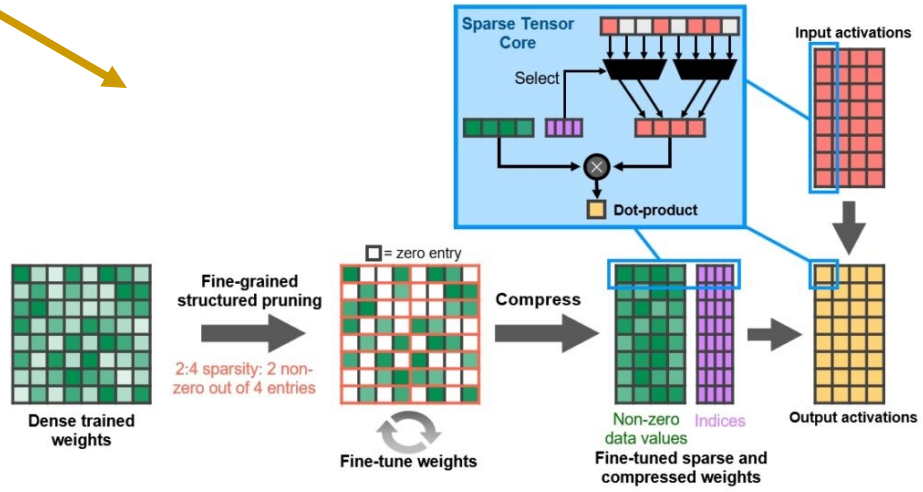


NVIDIA A100 Core

19.5 TFLOPS Single Precision

9.7 TFLOPS Double Precision

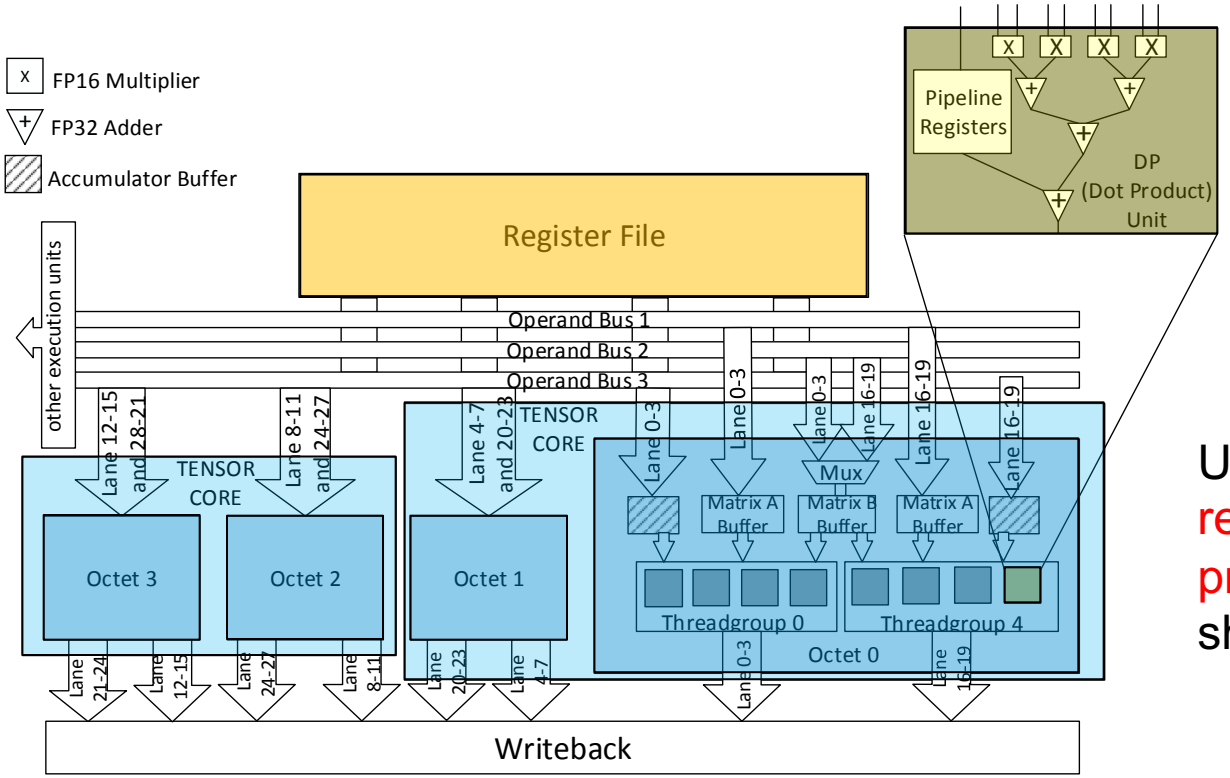
312 TFLOPS for Deep Learning (Tensor cores)



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Tensor Core Microarchitecture (Volta)

- Each warp utilizes two tensor cores
- Each tensor core contains two "octets"
  - 16 SIMD units per tensor core (8 per octet)
  - 4x4 matrix-multiply and accumulate each cycle per tensor core



SIMD unit

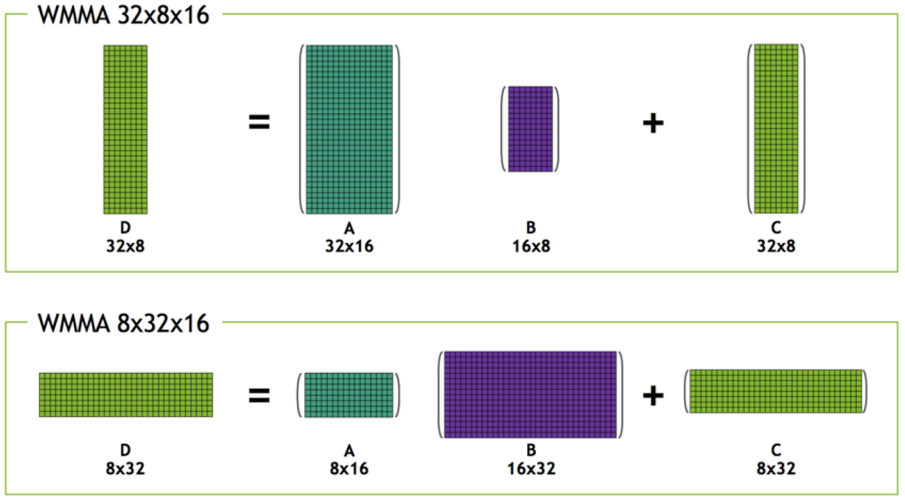
Proposed* tensor core microarchitecture

Unlike conventional SIMD, register contents are *not* private to each thread, but shared inside the warp

* M. A. Raihan, N. Goli and T. M. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," ISPASS 2019.

Tensor Core Operations

- Load/Store from TensorCore
- Fixed size matrix multiplication: usually 4x4, 16x16, or 64x64
- NVIDIA supports non-square matrix dimensions



Slide credit: Hwu & Kirk (PUMPS 2021)

TensorCore WMMA (WARP Matrix Multiply Accumulate) API

```
wmma::fragment<_Use, _M, _N, _K, _Type, _Layout>;

wmma::fill_fragment(fragment<…> &, const _Type &);

wmma::load_matrix_sync(fragment<…> &, const _Type *, unsigned,
_Layout);

wmma::store_matrix_sync(_Type *, const fragment<…> &, unsigned,
_Layout);

wmma::mma_sync(fragment<…> &, const fragment<…> &, const
fragment<…> &,
               const fragment<…> &, bool);
```

Fairly simple low level API that operates on memory fragments (internally registers)

Slide credit: Hwu & Kirk (PUMPS 2021)

Matrix Fragments

```
wmma::fragment<
        typename _Use,  /* matrix_a | matrix_b | accumulator */
        int _M,                 /* Fragment dimensions */
        int _N,
        int _K,
        typename _Type,  /* Data type  (half | float | …) */
        type _Layout = void /* row_major | col_major */
>;
```

A memory fragment (internally a set of registers)

Matrix Fragment Initialization with Constant Values

```
wmma::fill_fragment(
        fragment<…> & fragment,        /* Fragment to fill */
        const _Type & C                /* Constant value to fill
with */
);
```

Fills the fragment with some constant value **c**

Loading Matrix Fragment from Memory

```
wmma::load_matrix_sync(
        fragment<…> & fragment,      /* Fragment to load */
        const _Type * mptr,   /* Staring address to load from */
        unsigned lda,         /* Leading matrix dimension */
        _Layout layout;               /* Matrix layout */
);

wmma::load_matrix_sync(a_frag, &a[a_row + a_col * lda], lda);
```

Loads data from global or shared memory with specified stride into the fragment.

If the fragment is **matrix_a** or **matrix_b**, the layout is inferred from the fragment type.

In the example, we load a tile of a matrix (at **`&a[aRow+aCol*lda]`**) into **`a_frag`** with **`lda`** as the stride.

Slide credit: Hwu & Kirk (PUMPS 2021)

Storing Matrix Fragment to Memory

```
wmma::store_matrix_sync(
        _Type * mptr,              /* Staring address to store to */
        const fragment<…> & fragment, /* Fragment to store */
        unsigned lda,              /* Leading matrix dimension */
        _Layout layout;                  /* Matrix layout */
);
```

Stores the fragment into global or shared memory with specified stride and layout

GEMM of Matrix Fragments

```
wmma::mma_sync(
        fragment<…> & out,    /* Output fragment */
        const fragment<…> & a,        /* Fragment A */
        const fragment<…> & b,        /* Fragment B */
        const fragment<…> & c,        /* Fragment C */
        bool satf = false       /* Saturate to +-MAX_NORM */
);
```

Stores in **out** the result of computing **a x b + c**

If **satf** is true, the following saturation happens:

- Infinite saturates to MAX_NORM
- Minus infinite saturates to –MAX_NORM
- NaN is transformed to 0

Slide credit: Hwu & Kirk (PUMPS 2021)

GEMM Kernel Using MMA in Half Precision (1 of 3)

```
#include <mma.h>
using namespace nvcuda::wmma;
__global__ void dot_wmma(half * a, half * b, half * c,
                         int M, int N, int K, float alpha, float beta) ) {
   int lda = M; int ldb = K; int ldc = M;
   int warp_m = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
   int warp_n = (blockIdx.y * blockDim.y + threadIdx.y);

   fragment<matrix_a, 16, 16, 16, half, col_major> a_frag;
   fragment<matrix_b, 16, 16, 16, half, row_major> b_frag;
   fragment<accumulator, 16, 16, 16, half> acc_frag;
   fragment<accumulator, 16, 16, 16, half> c_frag;

   fill_fragment(acc_frag, 0.0f);
```

GEMM Kernel Using MMA in Half Precision (2 of 3)

```
for (int i = 0; i < K; i += 16) {
   int a_row = warp_m * 16; int a_col = i;
   int b_row = i; int b_col = warp_n * 16;

   if (a_row < M && a_col < K && b_row < K && b_col < N) {
      /* Load fragments in the tile */
      load_matrix_sync(a_frag, a + a_row + a_col * lda, lda);
      load_matrix_sync(b_frag, b + b_row + b_col * ldb, ldb);

      /* Compute partial result and accumulate */
      mma_sync(acc_frag, a_frag, b_frag, acc_frag);
   }
}
```

Slide credit: Hwu & Kirk (PUMPS 2021)

GEMM Kernel Using MMA in Half Precision (3 of 3)

```
   int c_row = warp_m * 16; int c_col = warp_n * 16;

   if (c_row < M && c_col < N) {
      /* Load output fragment from memory */
      load_matrix_sync(c_frag, c + c_row + c_ol * ldc, ldc, mem_col_major);

      /* Update fragment with computed result */
      for(int i = 0; i < c_frag.num_elements; i++) {
         c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
      }
      /* Store output fragment */
      store_matrix_sync(c + c_row + c_col * ldc, c_frag, ldc, mem_col_major);
   }
}
```

Slide credit: Hwu & Kirk (PUMPS 2021)

Systolic Arrays

Google TPU Generation I (~2016)

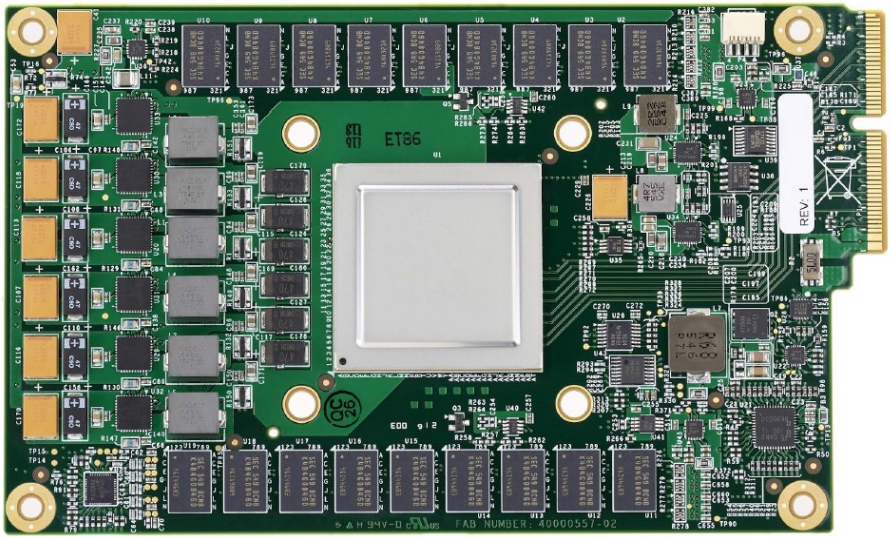


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

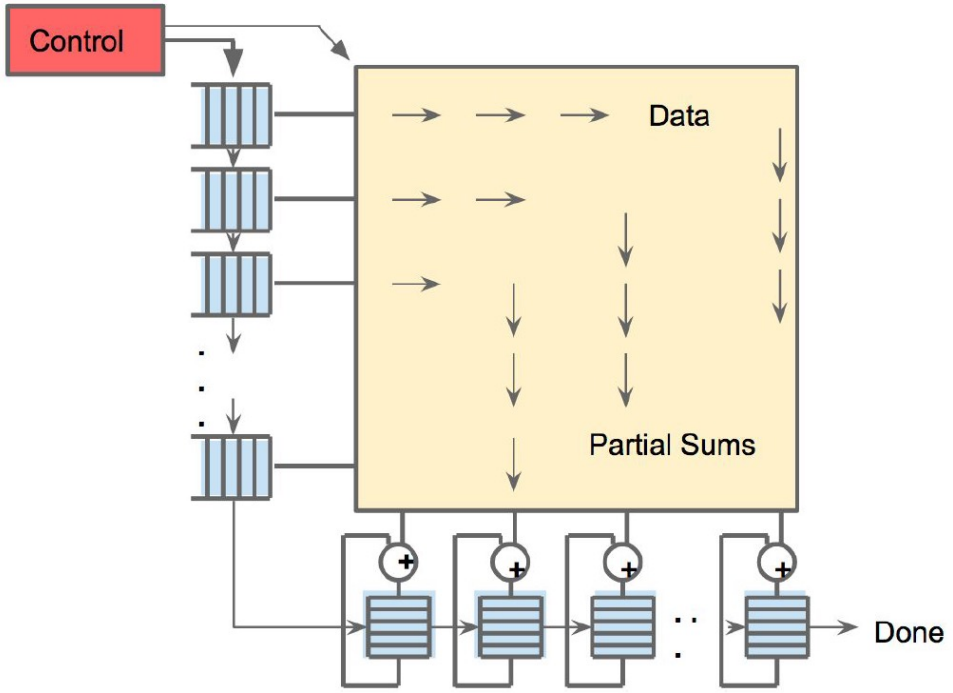
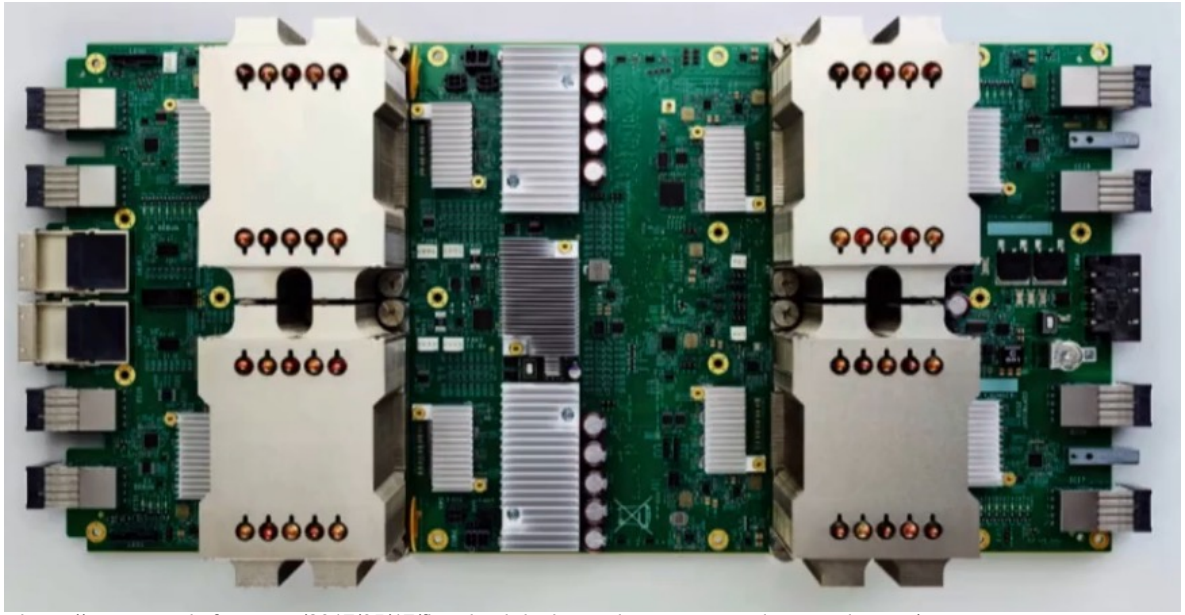


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA 2017.

Google TPU Generation II (2017)



https://www.nextplatform.com/2017/05/17/first-depth-look-googles-new-second-generation-tpu/

4 TPU chips
vs 1 chip in TPU1

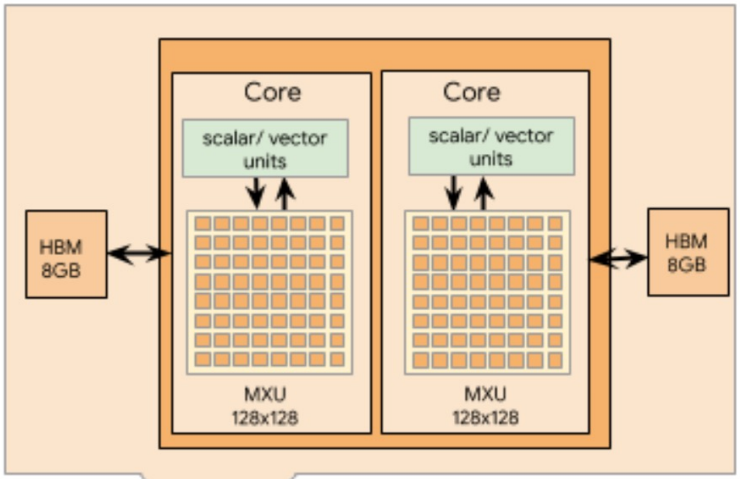
High Bandwidth Memory
vs DDR3

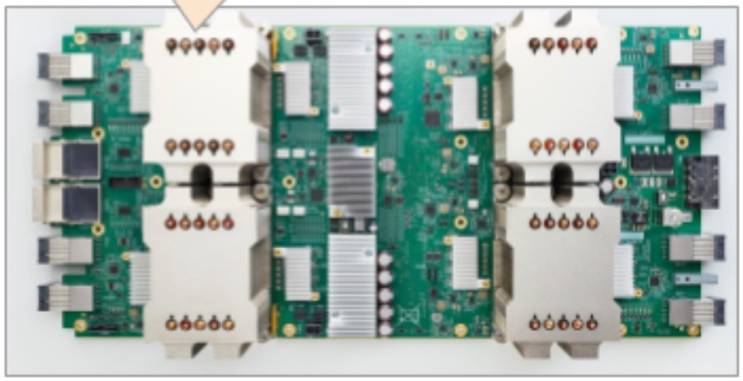
Floating point operations
vs FP16

45 TFLOPS per chip
vs 23 TOPS

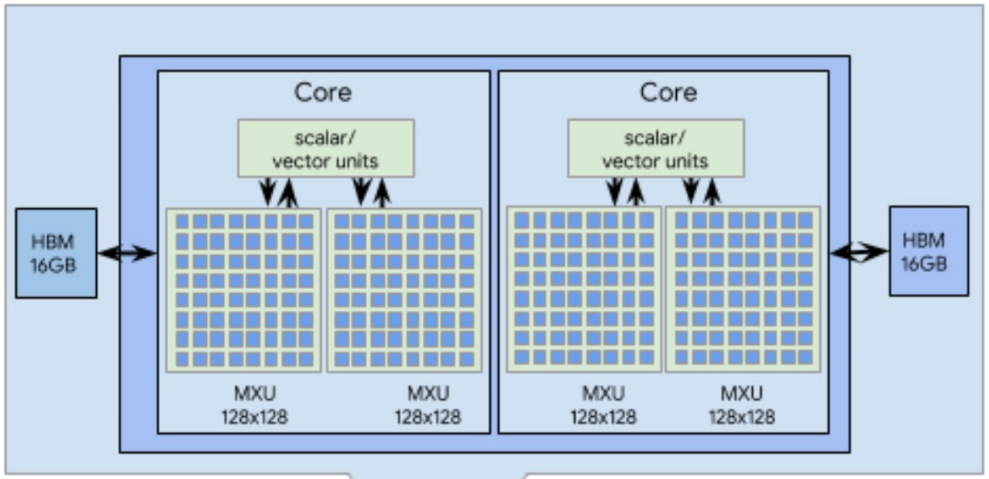
Designed for training
and inference
vs only inference

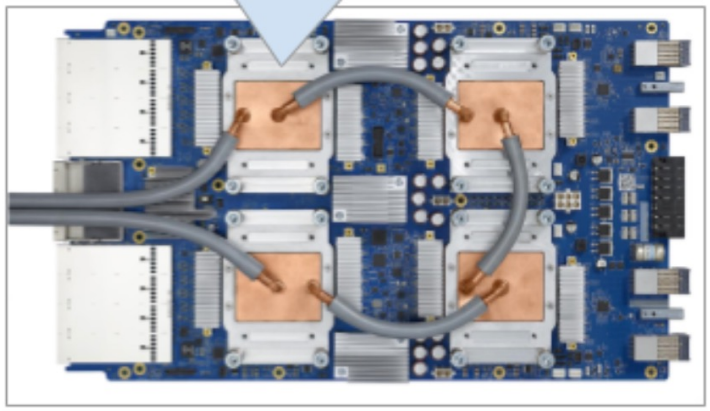
Google TPU Generation III (2019)





TPU v2 - 4 chips, 2 cores per chip





TPU v3 - 4 chips, 2 cores per chip

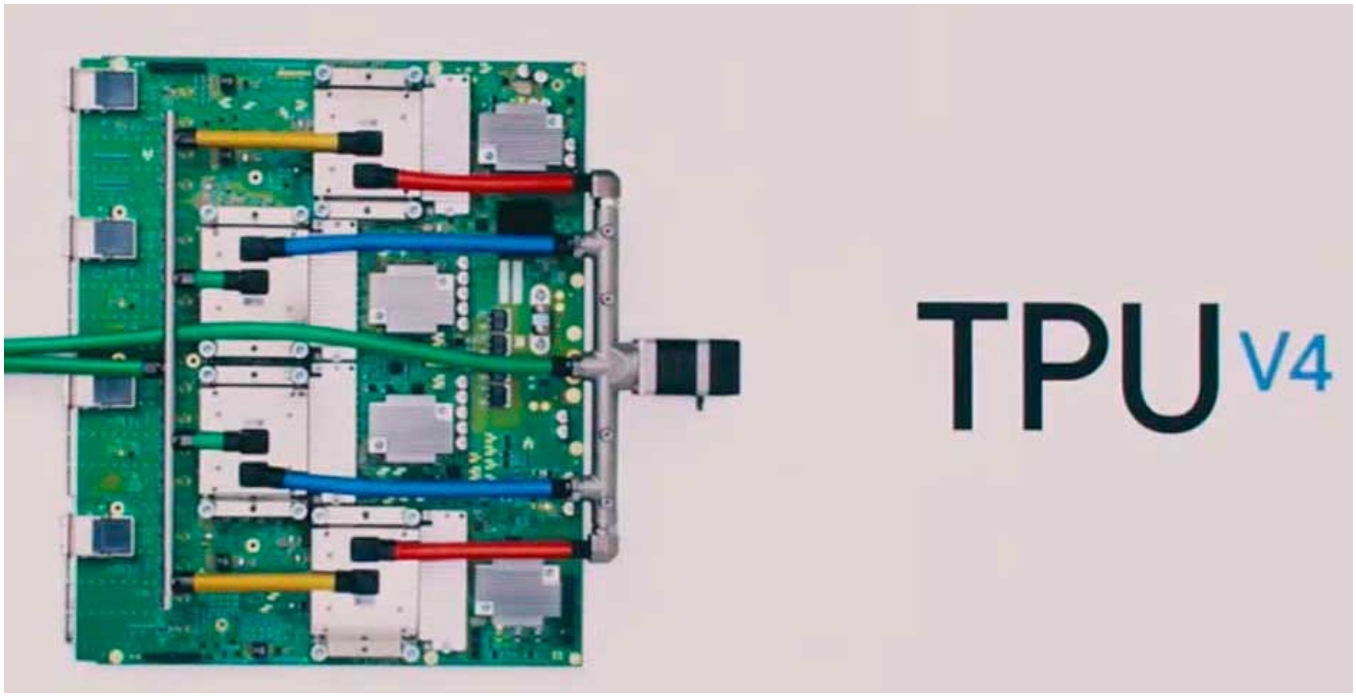
32GB HBM per chip
vs 16GB HBM in TPU2

4 Matrix Units per chip
vs 2 Matrix Units in TPU2

90 TFLOPS per chip
vs 45 TFLOPS in TPU2

https://cloud.google.com/tpu/docs/system-architecture

Google TPU Generation IV (2019)



New ML applications (vs. TPU3):

- Computer vision
- Natural Language Processing (NLP)
- Recommender system
- Reinforcement learning that plays Go

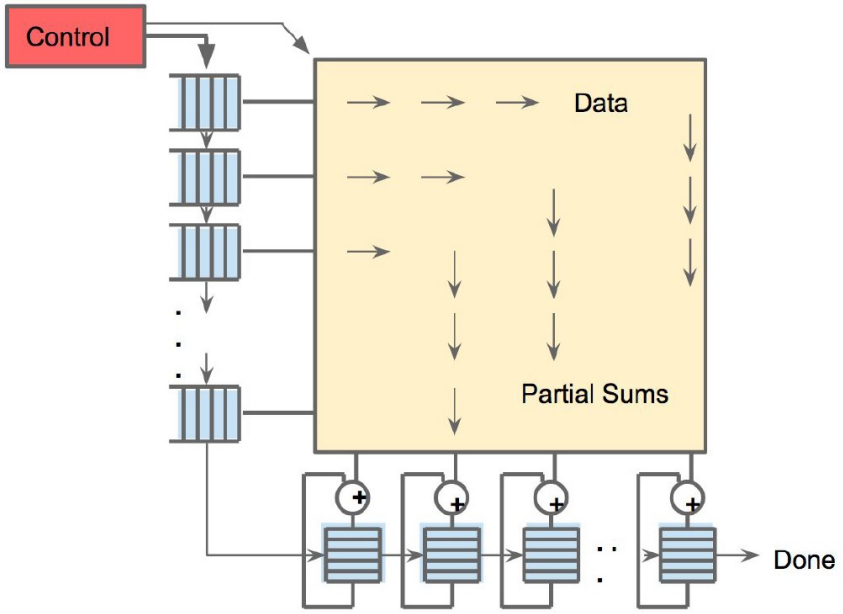
250 TFLOPS per chip in 2021
vs 90 TFLOPS in TPU3

1 ExaFLOPS per board

https://spectrum.ieee.org/tech-talk/computing/hardware/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests

An Example Modern Systolic Array: TPU (II)

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.



Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA 2017.

Example 2D Systolic Array Computation

- Multiply two 3x3 matrices (inputs)
  - Keep the final result in PE accumulators

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

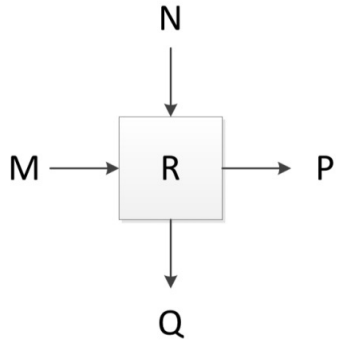
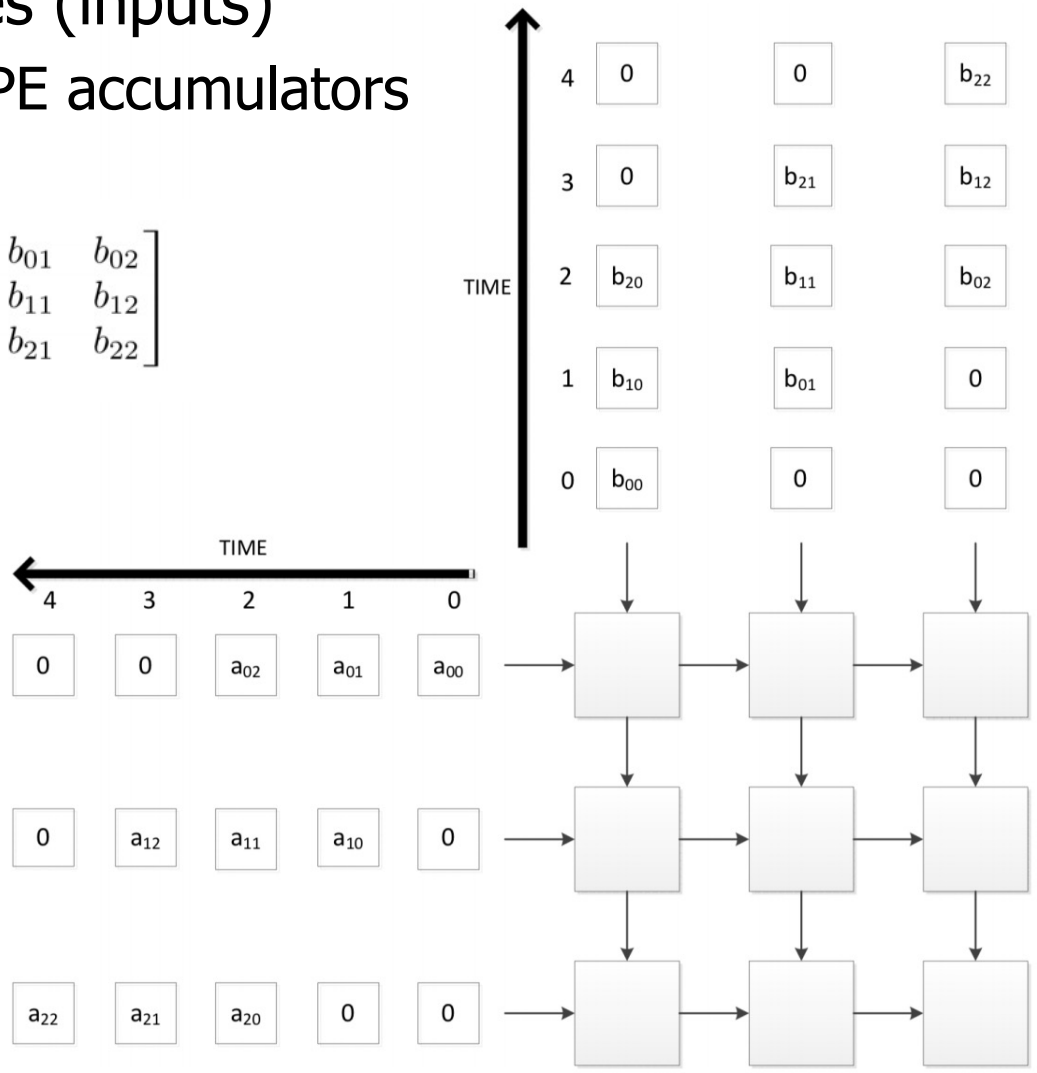


Figure 1: A systolic array processing element

P = M

Q = N

R = R + M*N



An Example Modern Systolic Array: TPU (III)

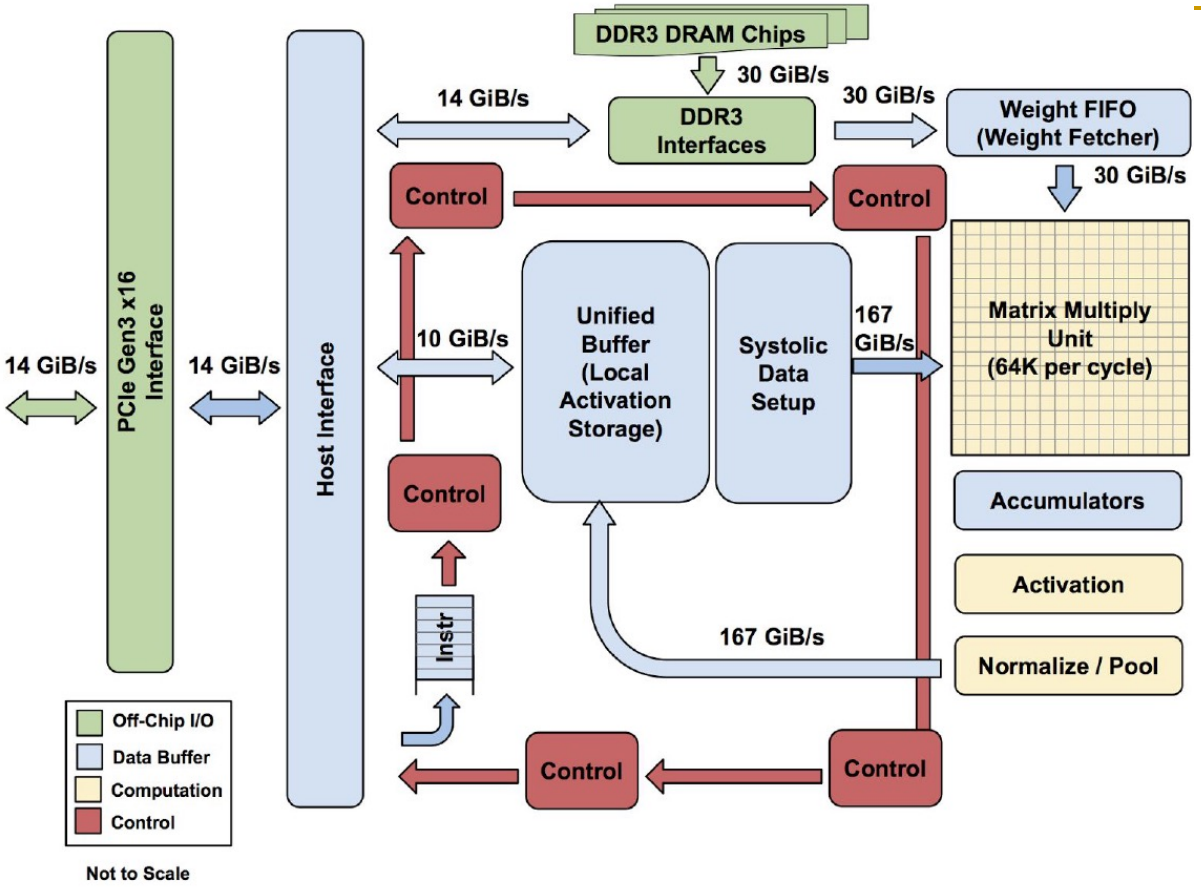
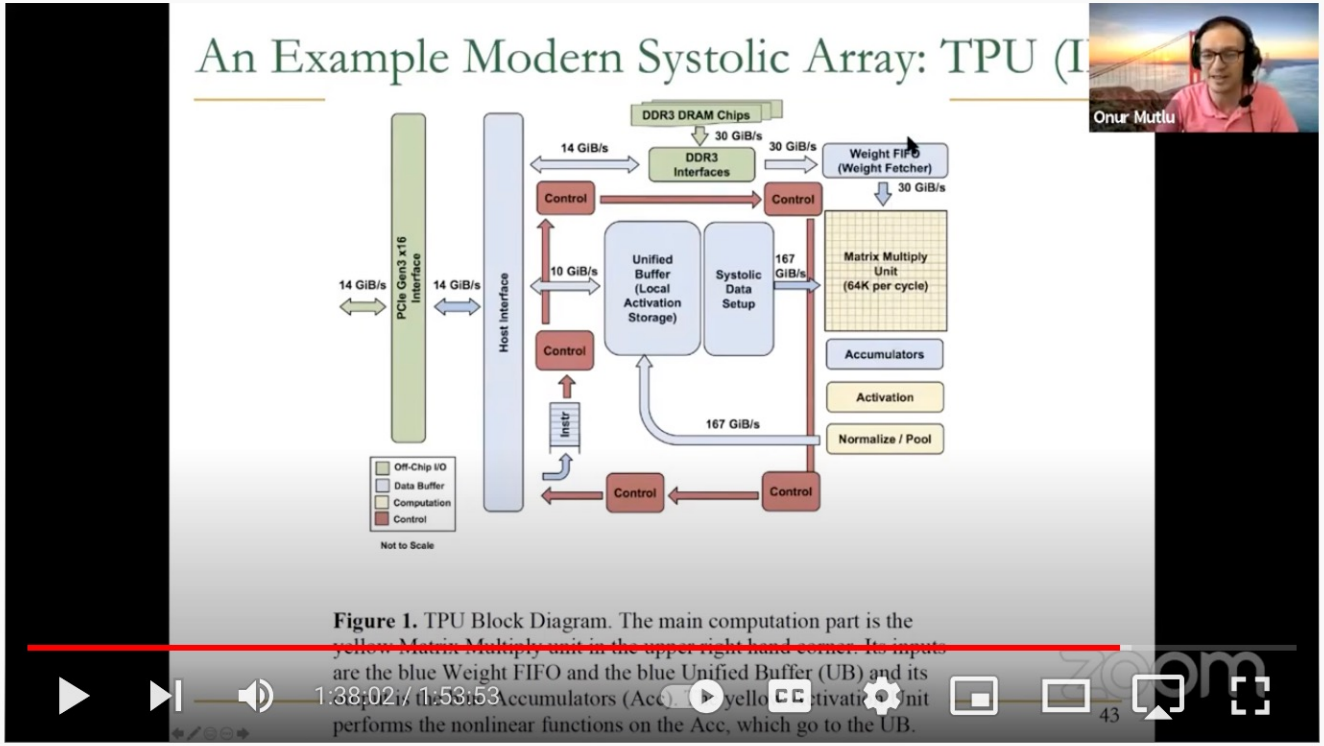


Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

Lecture on Systolic Arrays



Digital Design & Computer Arch. - Lecture 19: VLIW, Systolic Arrays, DAE (ETH Zürich, Spring 2021)

2,724 views • Streamed live on May 7, 2021 63 DISLIKE SHARE SAVE ...

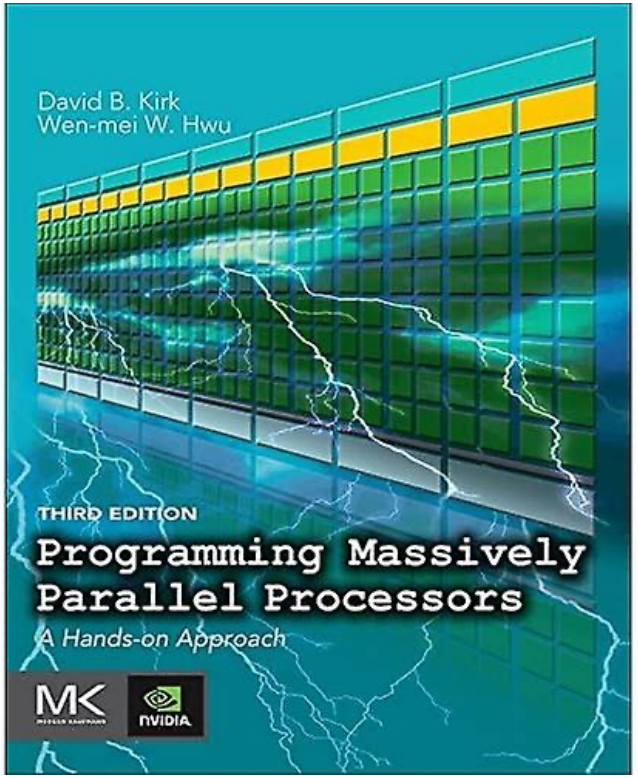
Onur Mutlu Lectures
20.1K subscribers

SUBSCRIBED

https://youtu.be/UtLy4Yagdys?t=2948

Recommended Readings

- Hwu and Kirk, "Programming Massively Parallel Processors," Third Edition, 2017
  - Chapter 7 - Parallel patterns — convolution: An introduction to stencil computation
  - Chapter 16 - Application case study — machine learning



P&S Heterogeneous Systems

Parallel Patterns: Convolution

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2021
25 November 2021