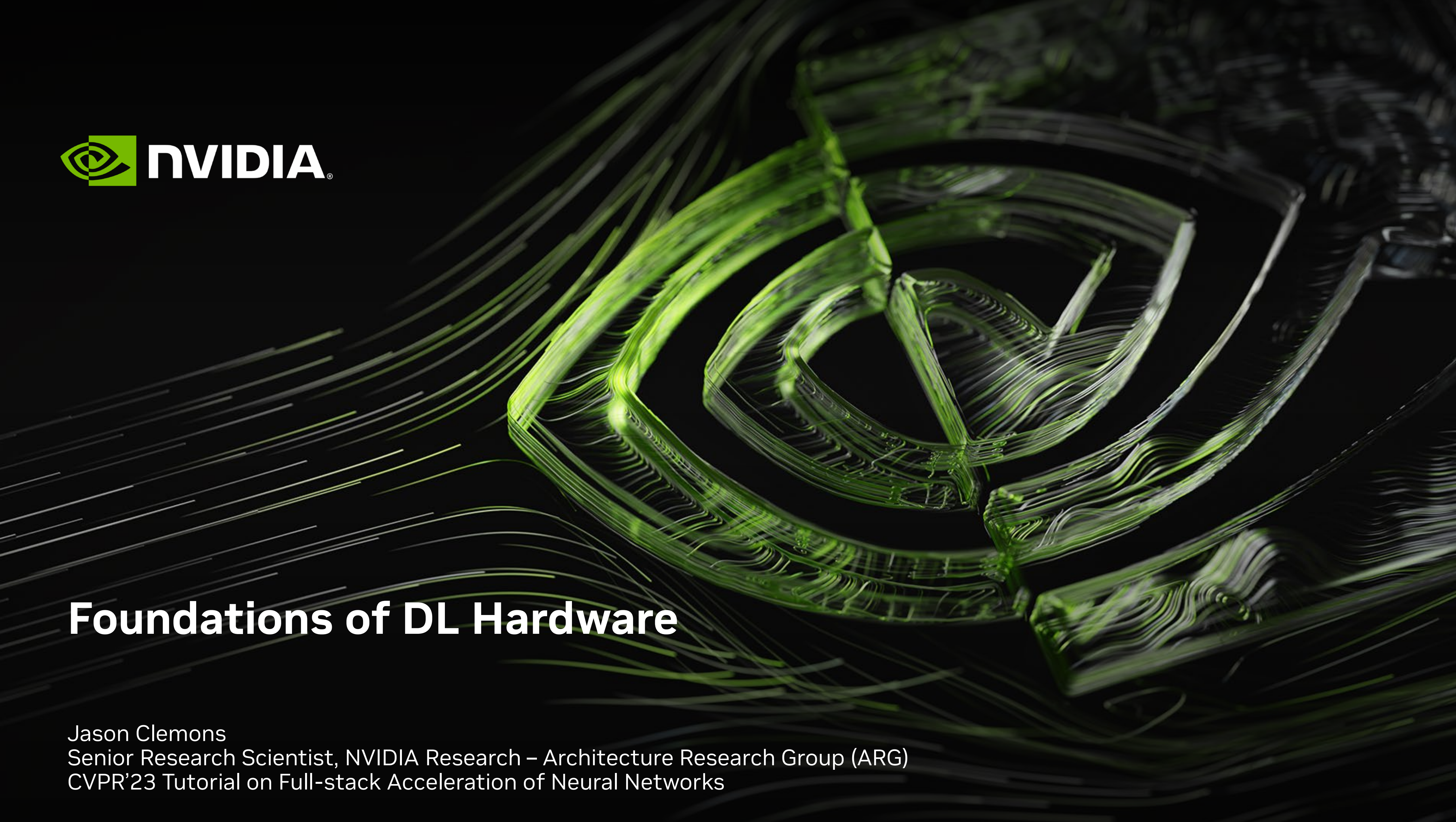
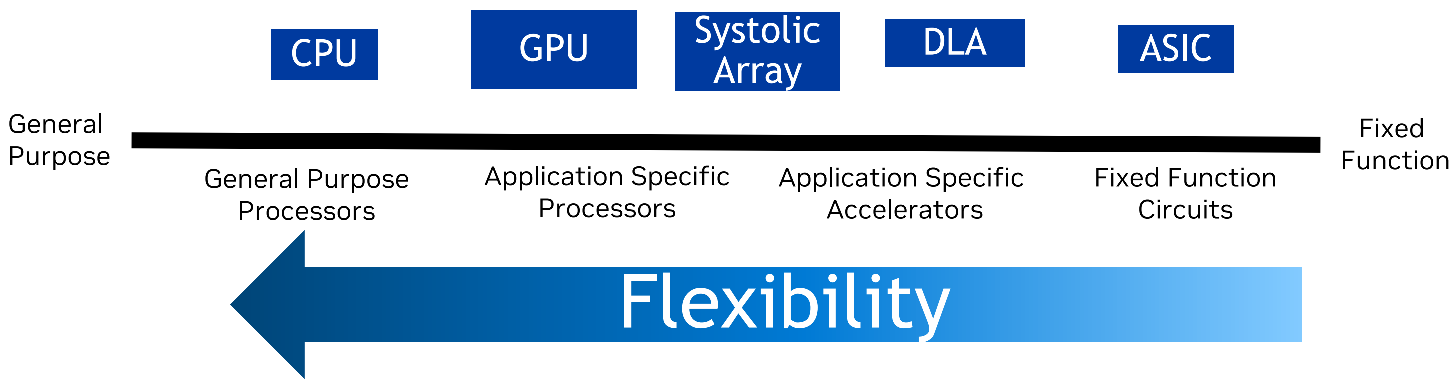

NVIDIA
Foundations of DL Hardware
Jason Clemons
Senior Research Scientist, NVIDIA Research – Architecture Research Group (ARG)
CVPR'23 Tutorial on Full-stack Acceleration of Neural Networks

Hardware Systems For DL



NVIDIA.

Hardware Systems For DL

CPU | GPU | Systolic Array | DLA | ASIC

General Purpose ——— Fixed Function

General Purpose Processors | Application Specific Processors | Application Specific Accelerators | Fixed Function Circuits

← Flexibility

CPU | GPU | Systolic Array | DLA | ASIC

General Purpose ——— Fixed Function

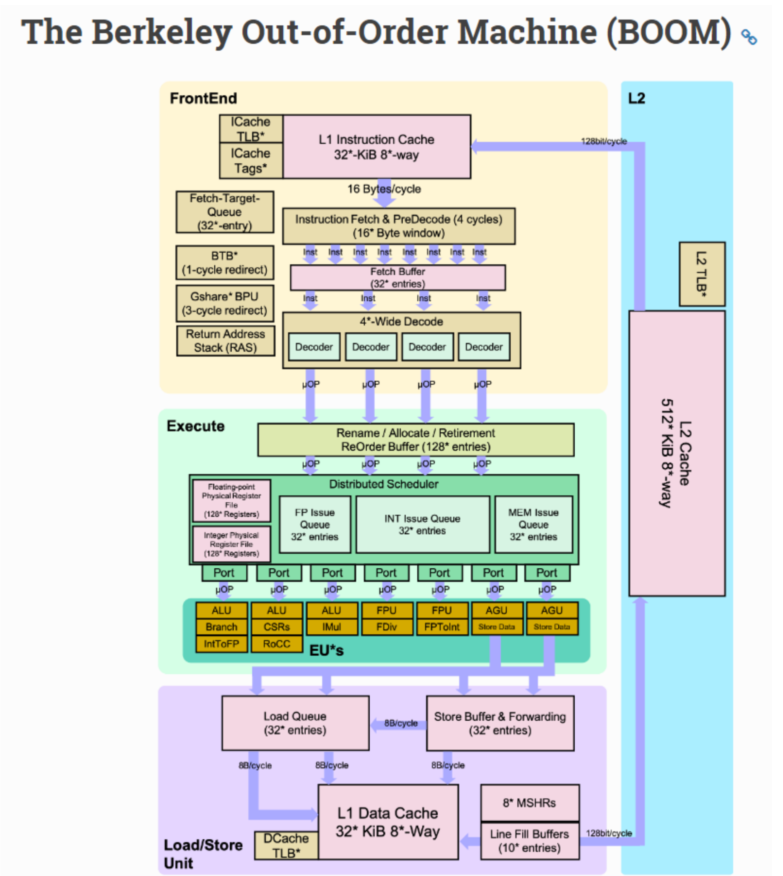
General Purpose Processors | Application Specific Processors | Application Specific Accelerators | Fixed Function Circuits

Efficiency →

CPU

Jack of all trades. Master of none

- Flexible instruction stream
  - Handle complex control flow programs
- Memory system for low latency
  - Fast local memory close to the compute (registers)
  - Caches provide locality benefits
- Lower core count/thread count
- Very good for complex control flow programs
- SIMD (Single Instruction Multiple Data)
  - Same operation on multiple pieces of data in parallel
  - Vector operations through vector extension units
- Viable for vector codes with limited parallelism
  - Or lower memory requirements
- Typically coordinates the system
  - Some level of good performance benefits attached accelerators



RISC-V Based CPU

Image from The Berkeley Out-of-Order Machine (BOOM) — RISCV-BOOM documentation (boom-core.org)

GPU

Highly Parallel

- Fully programmable
- Designed for parallel compute
  - Supported with high bandwidth memory system
- Efficient when operating in converged (lock step)
  - Supports divergent behavior unlike SIMD (vector) engines

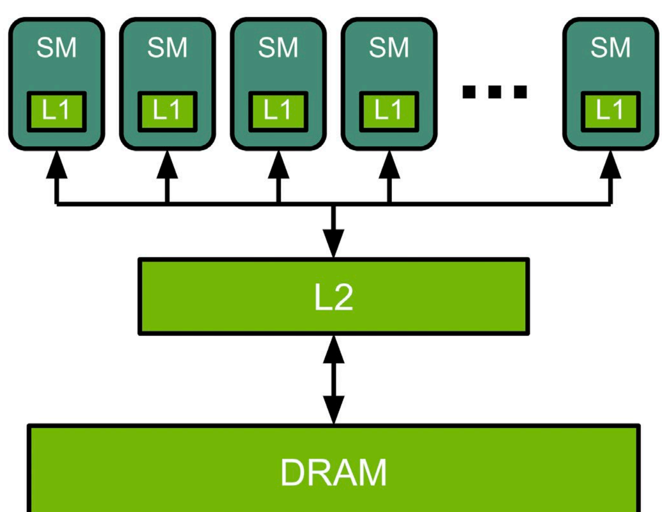


Figure 1. Simplified view of the GPU architecture

Figure 3. Utilization of an 8-SM GPU when 12 thread blocks with an occupancy of 1 block/SM at a time are launched for execution. Here, the blocks execute in 2 waves, the first wave utilizes 100% of the GPU, while the 2nd wave utilizes only 50%.

Figures from: https://docs.nvidia.com/deeplearning/performance/dl-performance-getting-started/index.html

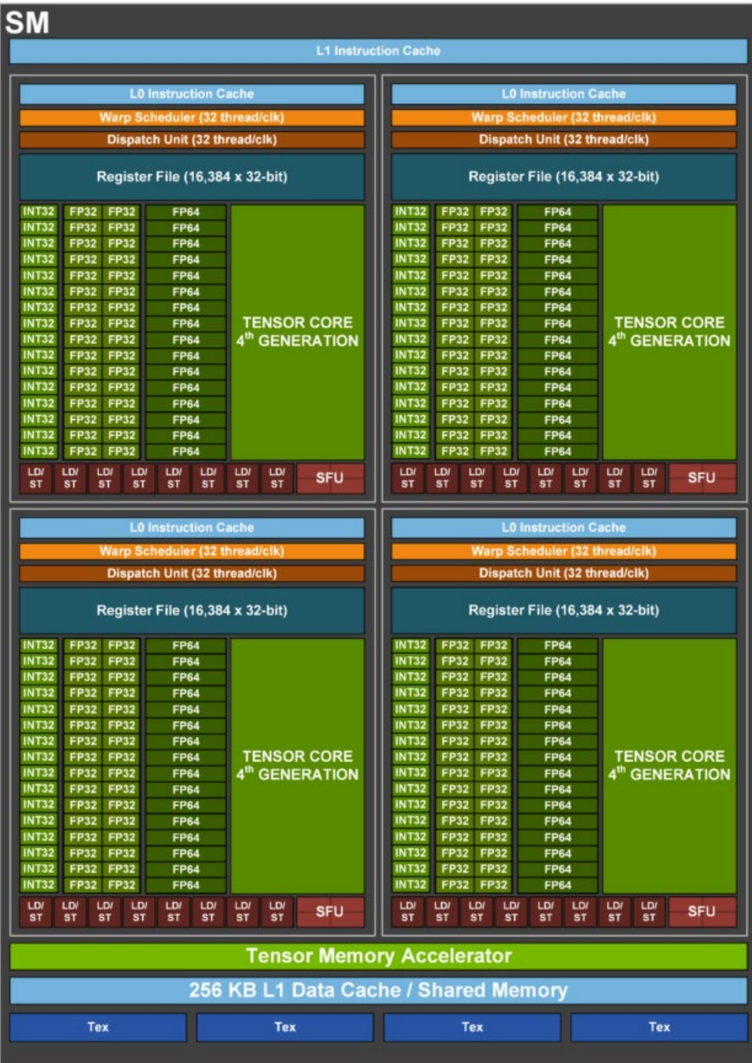
NVIDIA

Full system

- Full system has large amount of capability
- Memory system built for high throughput
  - Memory request coalescing
- Custom units for specific common operations
  - Includes tensor cores for tensor math operations
  - Tensor Memory Accelerator for efficient tensor memory accesses
- Supports multiple data formats: int8, FP32, FP16, BF16, FP8

Full H100: 144 SMs

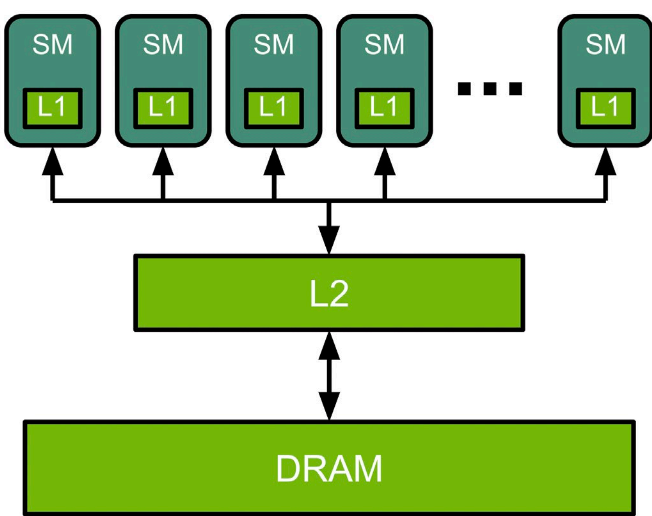
Hopper SM

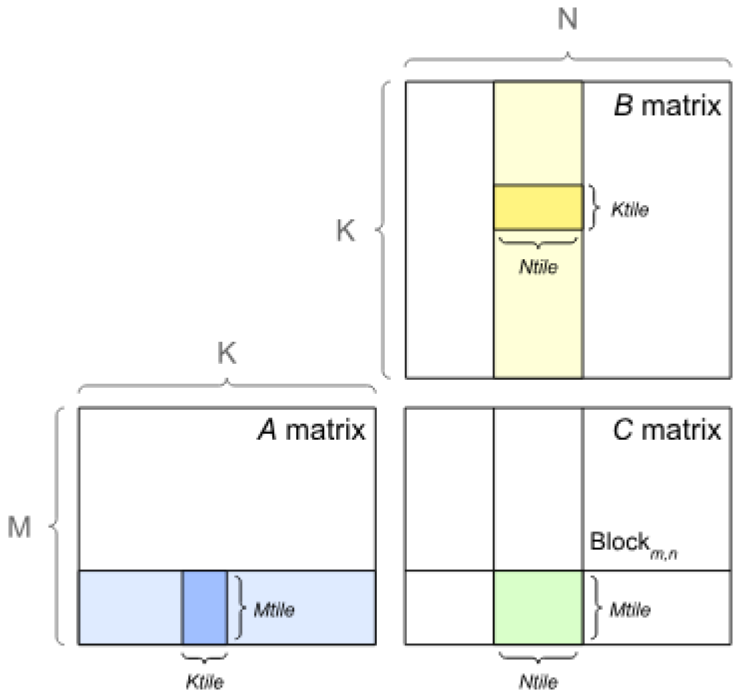


Mapping Work To The System

Tiling for GPU

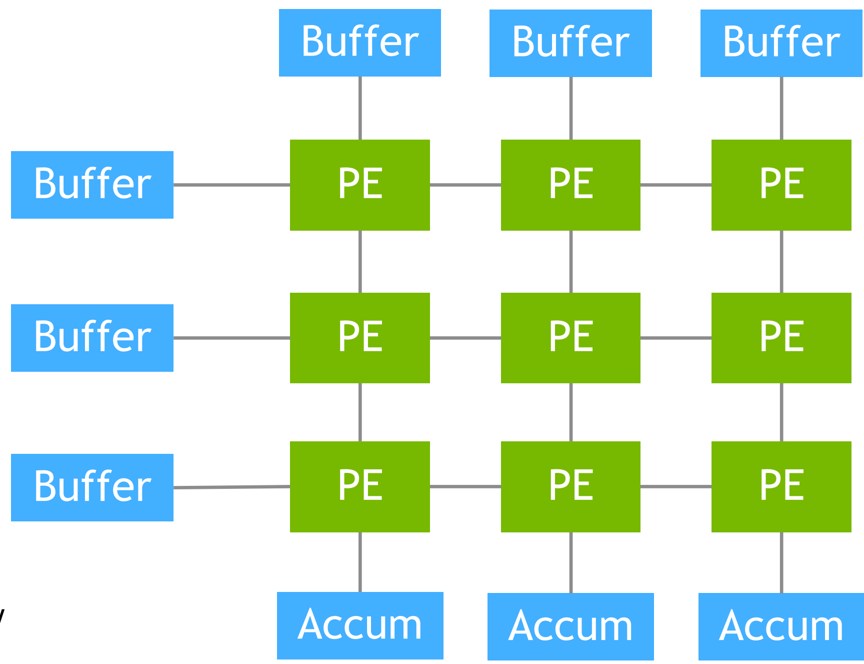
- Memory changes as you get closer to compute
  - Higher bandwidth, lower size
- Split problem up to fit in memory system
  - Problem is typically tiled to split the problem up





Systolic Array

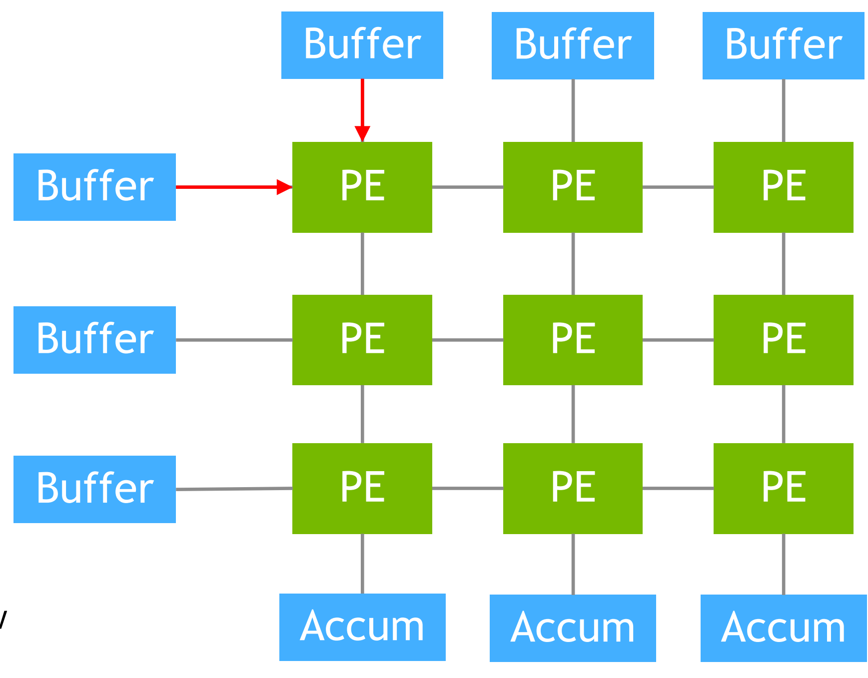
- Array of ALUs or processing elements (PEs)
- Each PE typically simple
  - Small local storage
  - Simple set of operations
- Data flows through the array
  - Each PE computes a part of the results and passes data along
- Paired with a host
  - Host interacts with a controller to decide operations
- Can be efficient if the problem maps well to the system
- Has limited flexibility due to application specific nature
- Problem needs to map well to the hardware array shape and flow
- Examples:
  - TPU[1], Eyeriss[2]



1. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
2. Chen el al. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. ISCA 2016

Systolic Array

- Array of ALUs or processing elements (PEs)
- Each PE typically simple
  - Small local storage
  - Simple set of operations
- Data flows through the array
  - Each PE computes a part of the results and passes data along
- Paired with a host
  - Host interacts with a controller to decide operations
- Can be efficient if the problem maps well to the system
- Has limited flexibility due to application specific nature
- Problem needs to map well to the hardware array shape and flow
- Examples:
  - TPU[1], Eyeriss[2]

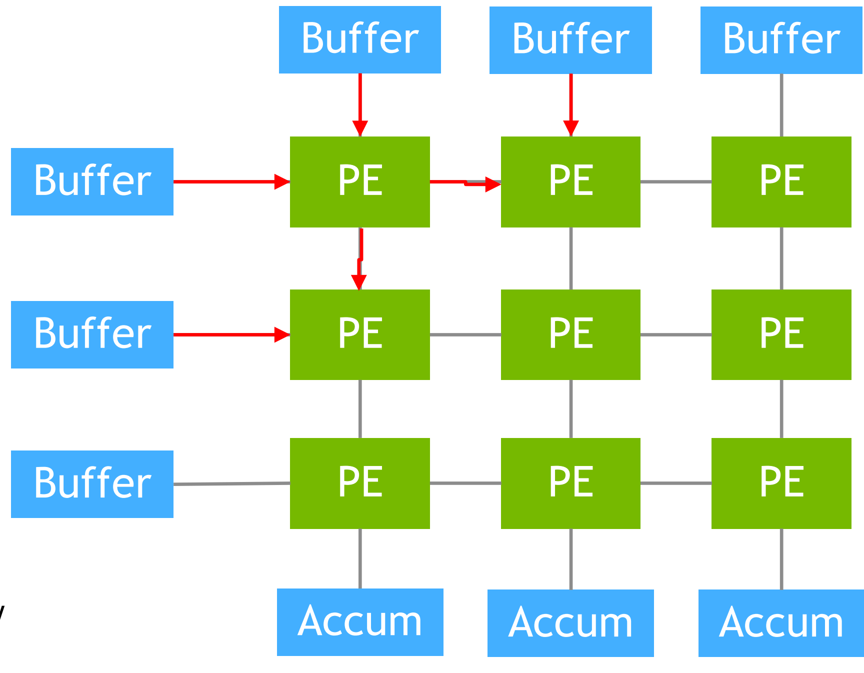


Data flow

1. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
2. Chen el al. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. ISCA 2016

Systolic Array

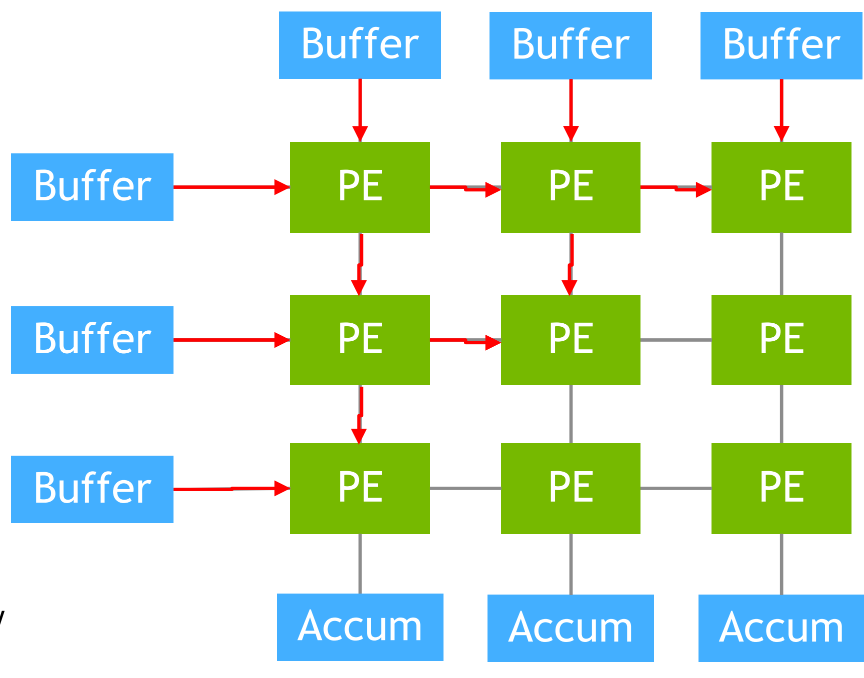
- Array of ALUs or processing elements (PEs)
- Each PE typically simple
  - Small local storage
  - Simple set of operations
- Data flows through the array
  - Each PE computes a part of the results and passes data along
- Paired with a host
  - Host interacts with a controller to decide operations
- Can be efficient if the problem maps well to the system
- Has limited flexibility due to application specific nature
- Problem needs to map well to the hardware array shape and flow
- Examples:
  - TPU[1], Eyeriss[2]



1. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
2. Chen el al. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. ISCA 2016

Systolic Array

- Array of ALUs or processing elements (PEs)
- Each PE typically simple
  - Small local storage
  - Simple set of operations
- Data flows through the array
  - Each PE computes a part of the results and passes data along
- Paired with a host
  - Host interacts with a controller to decide operations
- Can be efficient if the problem maps well to the system
- Has limited flexibility due to application specific nature
- Problem needs to map well to the hardware array shape and flow
- Examples:
  - TPU[1], Eyeriss[2]

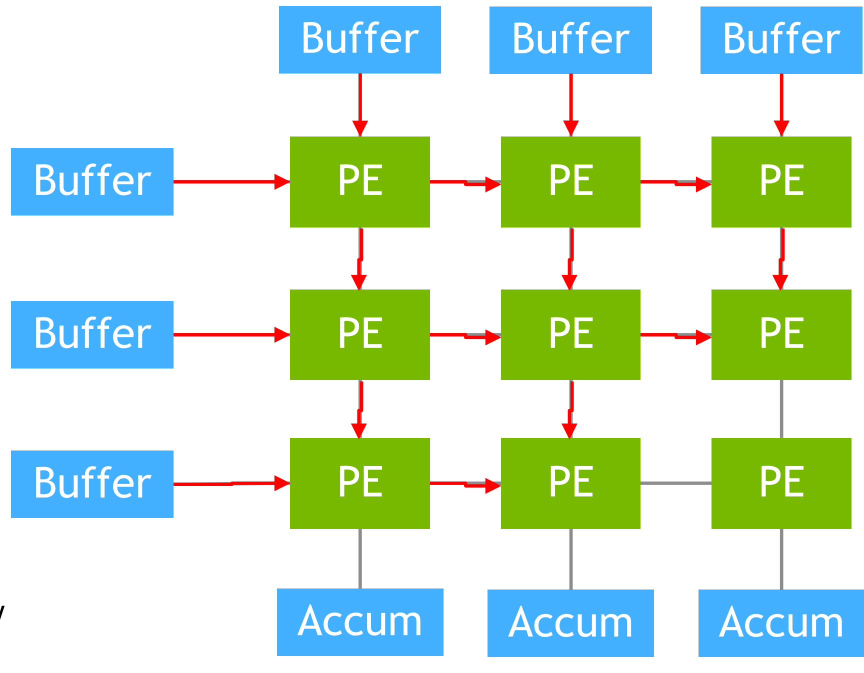


1. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
2. Chen el al. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. ISCA 2016

NVIDIA

Systolic Array

- Array of ALUs or processing elements (PEs)
- Each PE typically simple
  - Small local storage
  - Simple set of operations
- Data flows through the array
  - Each PE computes a part of the results and passes data along
- Paired with a host
  - Host interacts with a controller to decide operations
- Can be efficient if the problem maps well to the system
- Has limited flexibility due to application specific nature
- Problem needs to map well to the hardware array shape and flow
- Examples:
  - TPU[1], Eyeriss[2]

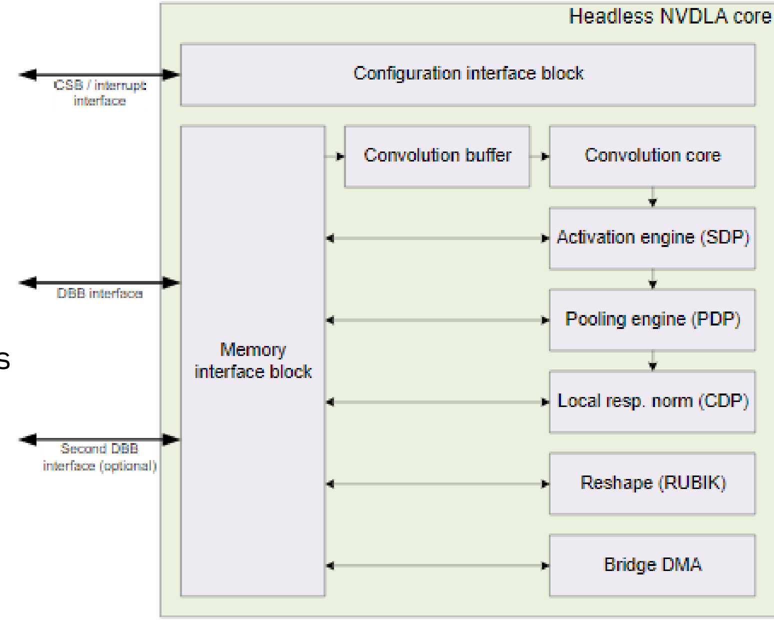


1. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
2. Chen el al. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. ISCA 2016

Custom Accelerators

NVDLA

- NVDLA is a custom accelerator for DL models
  - Example of a more specialized hardware accelerator
- Configurable instead of programmable
- High efficiency for the networks they accelerate
  - NVDLA is targeted for inference
  - NVDLA works for specific data types: int8, int16, fp16
- Custom accelerators require a good software eco system
  - NVIDIA provides tools such as TensorRT to generate network configs
- Commonly used in edge inference
  - NVDLA available in Jetson systems such as Orin



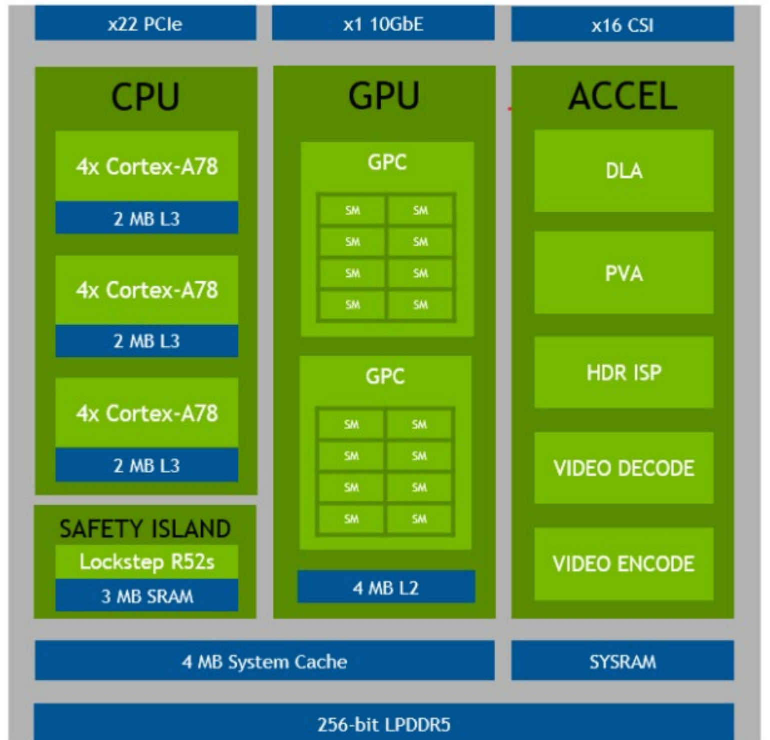
System On Chips

SoCs

- Contains multiple compute engines on single chip
  - CPUs, GPUs, and accelerators
- Shared memory system
- Common usage on edge devices
- Data flows between components
  - Application components run on most suitable engine
  - Requires more complex management (better eco system)

- Examples
  - Nvidia Jetson Orin
  - Apple M2

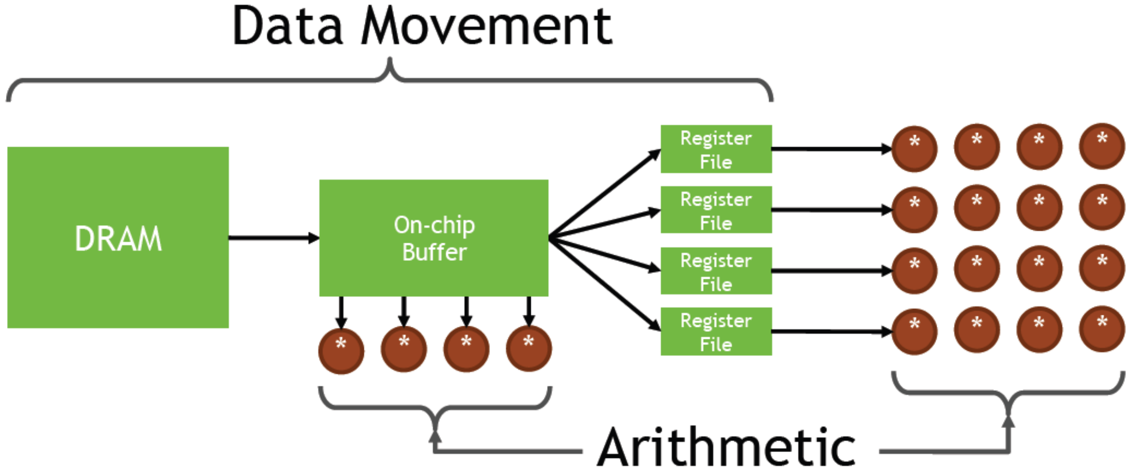
Figure 2: Orin System-on-Chip (SoC) Block Diagram



NVIDIA

Importance of Memory Accesses

General Accelerator Example



Why is
memory movement critical ?

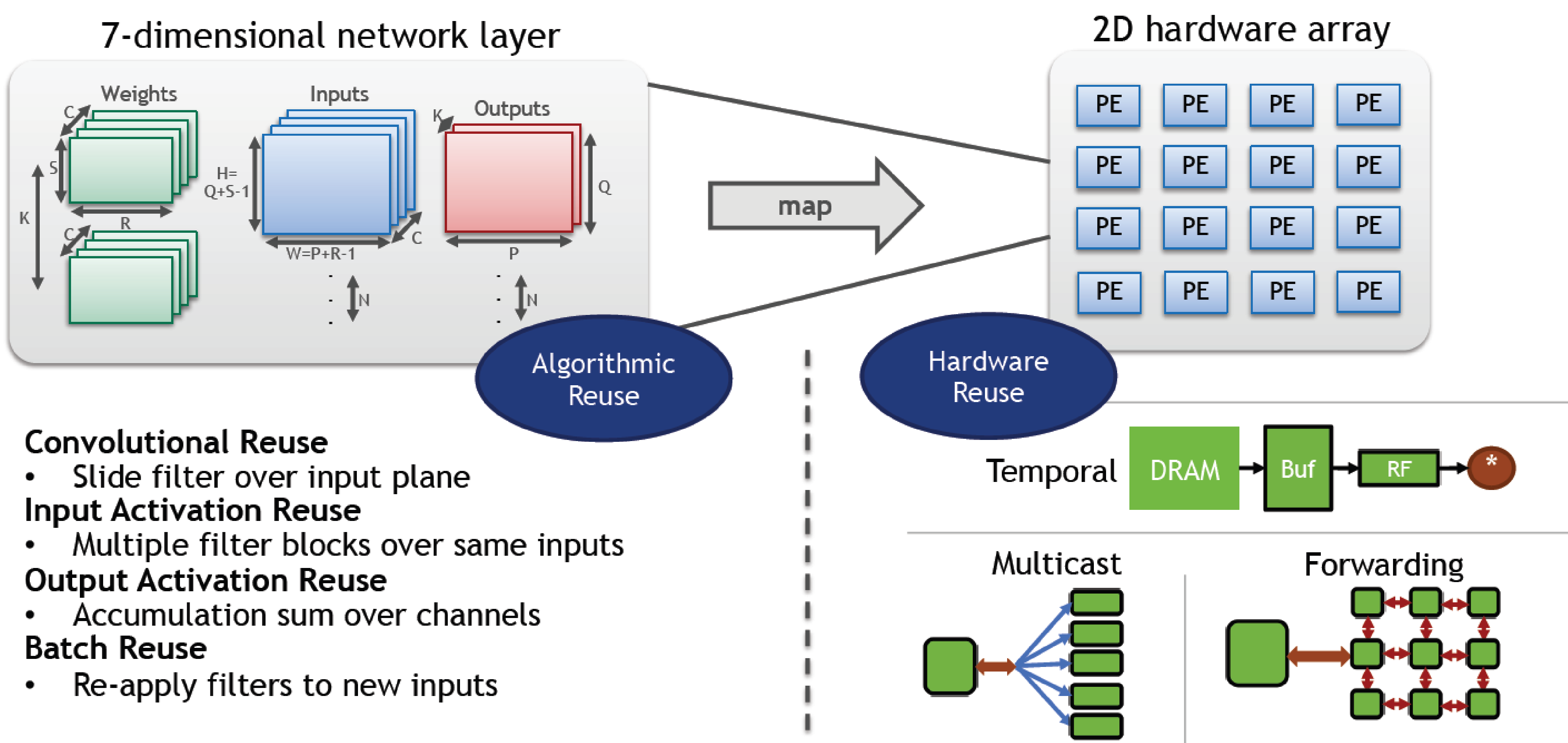
Efficiency/Energy

Energy costs	
8-bit Integer Multiply	0.2 pJ
Fetch two 8-bit operands from DRAM	128 pJ
Fetch two 8-bit operands from large SRAM	2 pJ

Figures are from [Timeloop tutorial]

Importance of Mapping

Leverage reuse for efficiency



Convolutional Reuse
- Slide filter over input plane

Input Activation Reuse
- Multiple filter blocks over same inputs

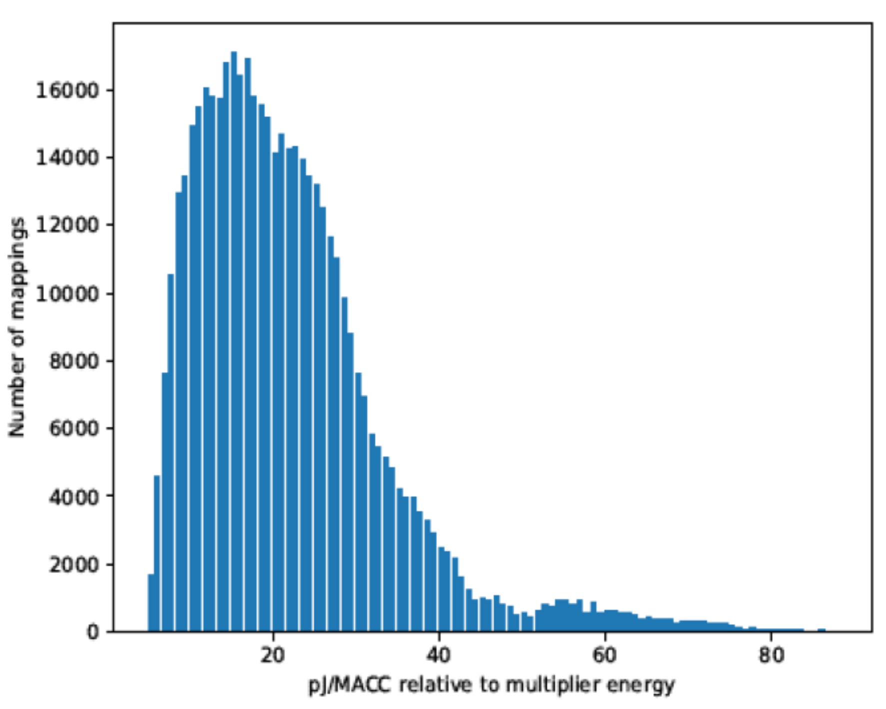
Output Activation Reuse
- Accumulation sum over channels

Batch Reuse
- Re-apply filters to new inputs

Flexible architectures may allow millions of alternative mappings of a single workload

Figures are from Timeloop tutorial

Importance of Good Mappings



480,000 mappings shown

Spread: 19x in energy efficiency

Only 1 is optimal, 9 others within 1%

A model needs a mapper to evaluate a DNN workload on an architecture

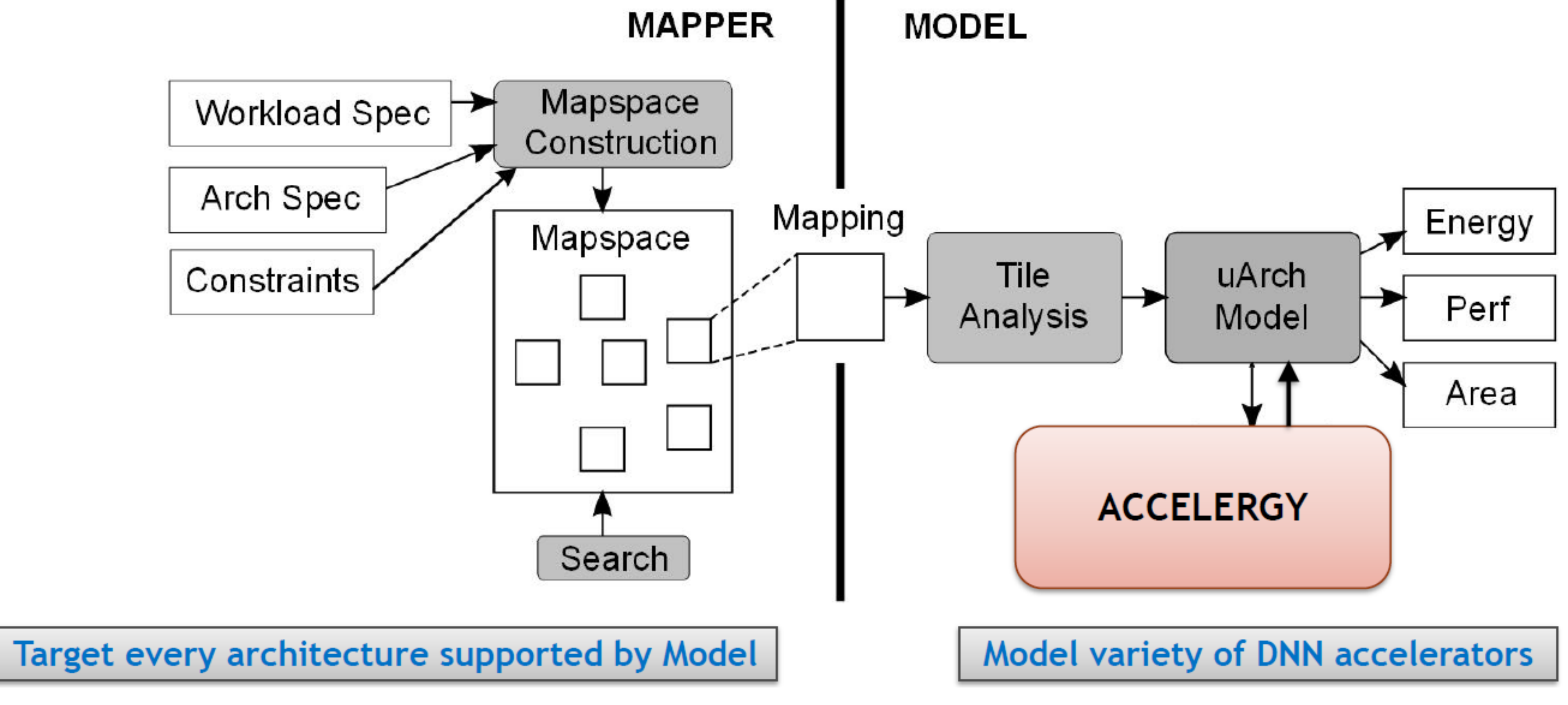
6,582 mappings have min. DRAM accesses but vary 11x in energy efficiency

A mapper needs a good cost model to find an optimal mapping

Figures are from Timeloop tutorial

Timeloop and Accelergy

Tools for determining efficient mappings



Figures are from Timeloop tutorial

More information at https://github.com/Accelergy-Project/timeloop-accelergy-tutorial

NVIDIA

Common Concepts for Performance

Performance Metrics

- Latency
  - Time to complete an operation
  - Units: time before work complete
    - Example
      - Add operation – 1 cycles
      - Divide operation – 8 cycles
      - Function call/return – 10 ns
- Energy Per Operation
  - Average energy to complete an operation
  - Example:
    - 10 pJ per Op
- Throughput
  - Rate at which a work can be completed
  - Units: work per unit time
    - Example:
      - Frames per second (FPS)
      - Bytes per second (Bps)
- Power
  - Rate at which energy is used
  - Example:
    - System TDP: 30 Watts (W)
- Perf per Watt
  - Performance metric per Watt
  - Measures efficiency
  - Example:
    - FLOPS/Watt

Latency Vs Throughput

A simple example

- System delivers throughput of 30 FPS
- What is the latency of this system?
    A. 1/30 seconds ~33 ms
    B. 1/15 seconds ~ 66ms
    C. 1 second ~ 1000 ms
    D. 2 seconds ~ 2000 ms
-

Latency Vs Throughput

A simple example

- System delivers throughput of 30 FPS
- What is the latency of this system?
    A. 1/30 seconds ~33 ms
    B. 1/15 seconds ~ 66ms
    C. 1 second ~ 1000 ms
    D. 2 seconds ~ 2000 ms

-

The answer is yes.

All of them could be right!

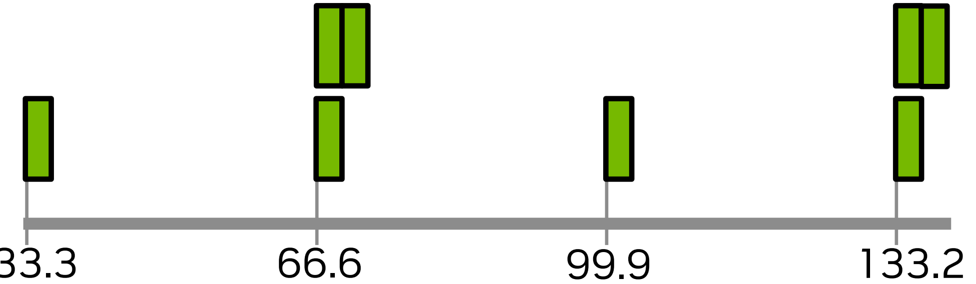
Latency Vs Throughput

A simple example

- System delivers throughput of 30 FPS
- What is the latency of this system?
  A. 1/30 seconds ~33 ms (1 frame per batch)
  B. 1/15 seconds ~ 66ms (2 frames per batch)
  C. 1 second ~ 1000 ms (30 frames per batch)
  D. 2 seconds ~ 2000 ms (60 frames per batch)
-

The answer is yes.
All of them could be right!
They all produce 30 FPS!

-1 frame



Throughput may not be enough specification!

NVIDIA.

Common Concepts for Performance

Performance Metrics

- Compute Bandwidth
  - Operations per unit time
  - Example:
    - 1024 Math units, 1 GHz clock, 1 Operation per clock
      - $1024 \times (1 \times 10^9) / 1 = 1024$ GOPs Compute Bandwidth
    - 1024 Math units, 1 GHz clock, 2 Operation per clock
      - $1024 \times (1 \times 10^9) / 1 = 512$ GOPs Compute Bandwidth
- Compute Bandwidth Utilization
  - Fraction of peak compute bandwidth achieved
  - Example:
    - Peak: 1 TFLOPS
    - Achieved: 400 GFLOPS
    - Compute Bandwidth Utilization: $400 \times 10^9 / 1 \times 10^{12} = .40$ or 40%

- Memory Bandwidth
  - Bytes per unit Time
  - Example:
    - 1 GB, in 5 seconds
      - 200 MB/s
- Memory Bandwidth Utilization
  - Fraction of peak memory bandwidth achieved
  - Example:
    - Peak: 1TB/s
    - Achieved: 200 GB/s
    - Memory Bandwidth Utilization: $200 \times 10^9 / 1 \times 10^{12} = .20$ or 20%
- Arithmetic Intensity
  - Flops per byte fetched
  - Correlates with data reuse
  - Example:
    - 32 FLOPs
    - 17 Bytes fetched
    - Arithmetic Intensity: 32 / 17 = 1.88 FLOPs/Byte

Estimating Performance

Bounding the performance

- Compute the FLOPs in the layers
  - 3×3 Conv2d with bias, Input is 1×3×416×480, Output is 1×16×208×240
    - FLOPs = ((3×3) × 3 +1) ×16 × 208 × 240 * 2= 22364160 * 2 = 44,728,320
    - FP32 Input tensor: 3×416×480*4 = 2,396,160
    - FP32 Output tensor: 16×208×240*4 = 3,194,880
    - FP32 Parameters: (3×3×3) × 16 + 16 = 448
  - Tools like torchinfo give can give you this

- Use the system peak compute and memory bandwidth to estimate performance
  - Assume peak compute bandwidth of 1 TOPS (1×10^{12} OPs)
    - Best case: 44,728,320/ 1×10^{12} = 0.00004472832 s = 44.8 ns compute
  - Assume peak memory bandwidth of 1000 GB/s
    - Best case: 559104 / 1000×10^{9} = 0.00000559104 s =5.6 ns memory transfer

- Realistically you assume some derating factor but now at least it is bounded

Understanding Performance

- Operations can be *memory-limited* and *math-limited*
  - <u>Math limited</u> – High utilization of the compute units while the memory bandwidth utilization is not high
    - Math limited if: (BWmath / PeakBWmath) ~> .7
  - <u>Memory limited</u> – High utilization of the memory bandwidth while the compute unit utilization is not high
    - Mem limited if: (BWmem / PeakBWmem) ~> .7
- Using Arithemtic intensity for estimating limitations
  - Math limited if: #ops / #bytes > PeakBWmath / PeakBWmem

Table 1. Examples of neural network operations with their arithmetic intensities. Limiters assume FP16 data and an NVIDIA V100 GPU.

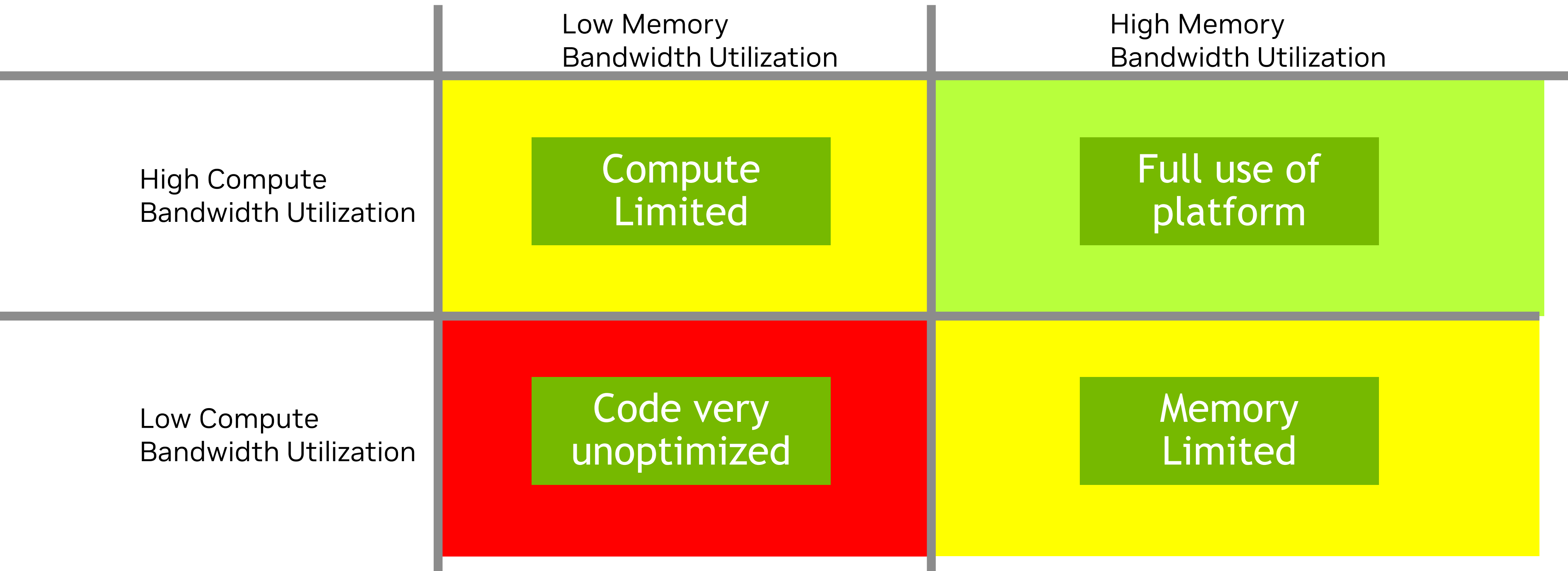
Operation	Arithmetic Intensity	Usually limited by...
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 FLOPS/B	arithmetic
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 FLOPS/B	memory
Max pooling with 3x3 window and unit stride	2.25 FLOPS/B	memory
ReLU activation	0.25 FLOPS/B	memory
Layer normalization	< 10 FLOPS/B	memory

B – byte

b - bit

4 Quadrants of Bandwidth

Top Right is optimal



NVIDIA

Hints on handling issues

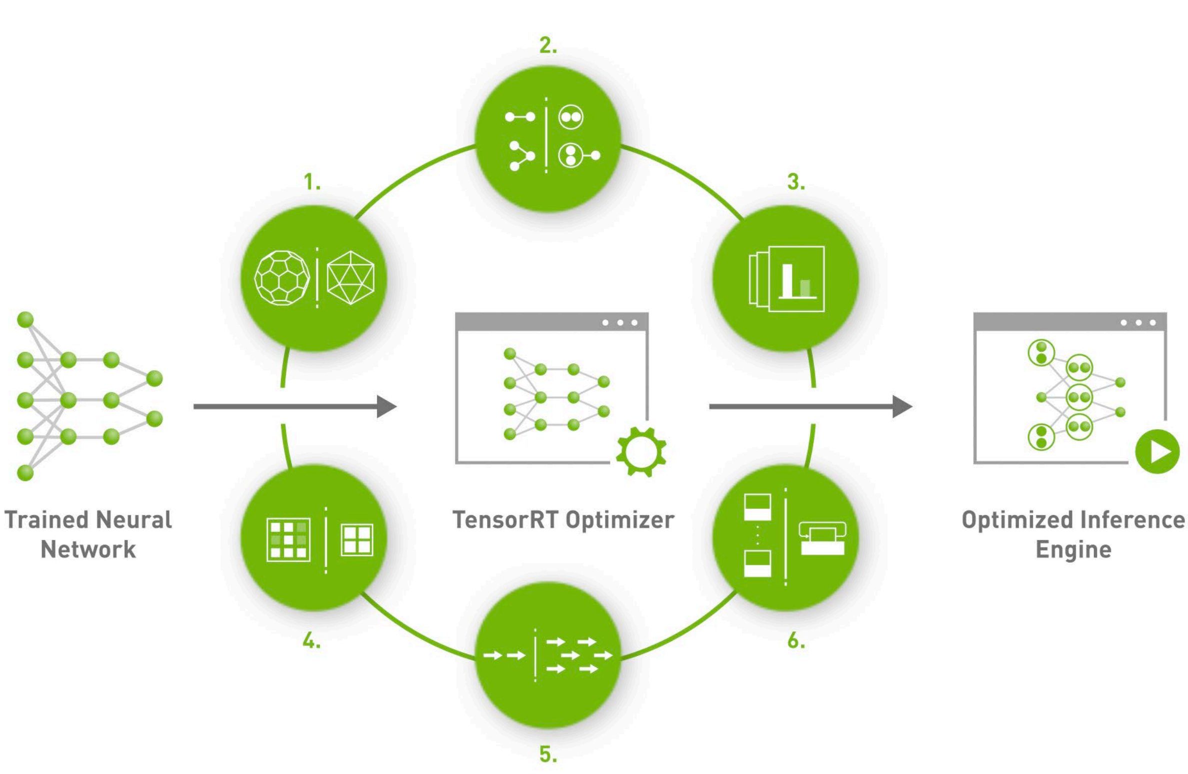
- Compute bandwidth limited
  - Move to a different numerical format
    - Some hardware supports more compute with smaller format
  - Rework the algorithm to use less compute
    - Due to parallelism small changes may not impact performance
  - Use structured sparsity
    - 2:4 structured sparsity approx. doubles the FLOPSs
  - Use specialized hardware
    - Tensor cores
- Memory bandwidth limited
  - Move to a different numerical format
    - You can move more operands in the same memory bandwidth
  - Overlap memory transfers
  - Restructure your data
    - Some data structures make better use of the memory transfers
  - Resize the problem
- Compute and Memory bandwidth limited
  - Reformulate the problem
  - Refactor the code

	NVIDIA H100 SXM5[1]	NVIDIA H100 PCIe[1]
Peak FP64[1]	30 TFLOPS	24 TFLOPS
Peak FP64 Tensor Core[1]	60 TFLOPS	48 TFLOPS
Peak FP32[1]	60 TFLOPS	48 TFLOPS
Peak FP16[1]	120 TFLOPS	96 TFLOPS
Peak BF16[1]	120 TFLOPS	96 TFLOPS
Peak TF32 Tensor Core[1]	500 TFLOPS | 1000 TFLOPS[2]	400 TFLOPS | 800 TFLOPS[2]
Peak FP16 Tensor Core[1]	1000 TFLOPS | 2000 TFLOPS[2]	800 TFLOPS | 1600 TFLOPS[2]
Peak BF16 Tensor Core[1]	1000 TFLOPS | 2000 TFLOPS[2]	800 TFLOPS | 1600 TFLOPS[2]
Peak FP8 Tensor Core[1]	2000 TFLOPS | 4000 TFLOPS[2]	1600 TFLOPS | 3200 TFLOPS[2]
Peak INT8 Tensor Core[1]	2000 TOPS | 4000 TOPS[2]	1600 TOPS | 3200 TOPS[2]

Table 1. NVIDIA H100 Tensor Core GPU preliminary performance specs

Remember: With NVIDIA GPUs, you can always write a custom kernel to for greater efficiency and control

What is TensorRT?



1. **Weight & Activation Precision Calibration**
   Maximizes throughput by quantizing models to INT8 while preserving accuracy
2. **Layer & Tensor Fusion**
   Optimizes use of GPU memory and bandwidth by fusing nodes in a kernel
3. **Kernel Auto-Tuning**
   Selects best data layers and algorithms based on target GPU platform
4. **Dynamic Tensor Memory**
   Minimizes memory footprint and re-uses memory for tensors efficiently
5. **Multi-Stream Execution**
   Scalable design to process multiple input streams in parallel
6. **Time Fusion**
   Optimizes recurrent neural networks over time steps with dynamically generated kernels

Figures are from [TensorRT SDK | NVIDIA Developer]

TensorRT

What does it do?

1. **Weight & Activation Precision Calibration**
   Maximizes throughput by quantizing models to INT8 while preserving accuracy

   Convert the model to int8 using common static techniques. It does not require retraining to generate the new operands

2. **Layer & Tensor Fusion**
   Optimizes use of GPU memory and bandwidth by fusing nodes in a kernels

   Combine common operations to avoid overheads of launching separate kernels or taking round trips to memory

3. **Kernel Auto-Tuning**
   Selects best data layers and algorithms based on target GPU platform

   Explores various transforms that may be more efficient ways of performing the operations of a layer. For example, it may transform a convolution into a Winograd convolution if that is more efficient. It may change the way it tiles the problem based on the hardware it is targeting.

4. **Dynamic Tensor Memory**
   Minimizes memory footprint and re-uses memory for tensors efficiently

   Release memory that may not be used after the current operation

5. **Multi-Stream Execution**
   Scalable design to process multiple input streams in parallel

   Independent data streams can share the hardware spatially (run at the same time) on different pieces of the hardware.

6. **Time Fusion**
   Optimizes recurrent neural networks over time steps with dynamically generated kernels

   Optimize the time step expansion of RNNs

TensorRT

Summary

- Tool to optimize networks
- Takes a network and generates a highly optimized version
- Can target specific hardware, including DLA
  - Can use both GPU and DLA to run network
  - Can determine which is best for a given layer and set it to run there
- Uses a special runtime engine for optimized network execution
- Integrations into PyTorch and TF
  - TF: tensorflow-gpu has integrations and just needs tensorRT to be installed
  - PyTorch: torch-tensorrt is available.
  - Also, NVIDIA provides NGC docker images with these integrations ready to use (ngc.nvidia.com)

NVIDIA Profiling Tools

Suite Of Applications

- Nsight Systems (nsys)
  - Provides primarily system level information
    - Timing performance
    - Memory Bandwidth
    - Tensor core utilization
    - And many more
- Nsight Compute
  - Provides kernel level information
    - Kernel FLOPS
    - Memory Usage
    - SM Occupancy
    - And many more

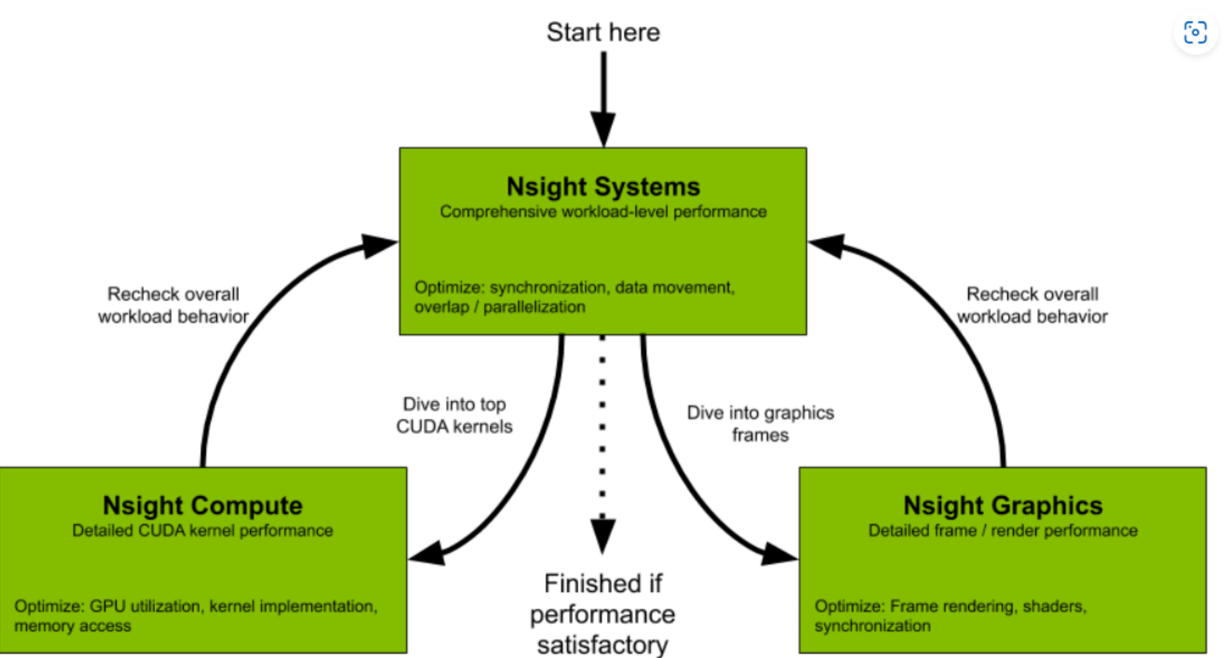
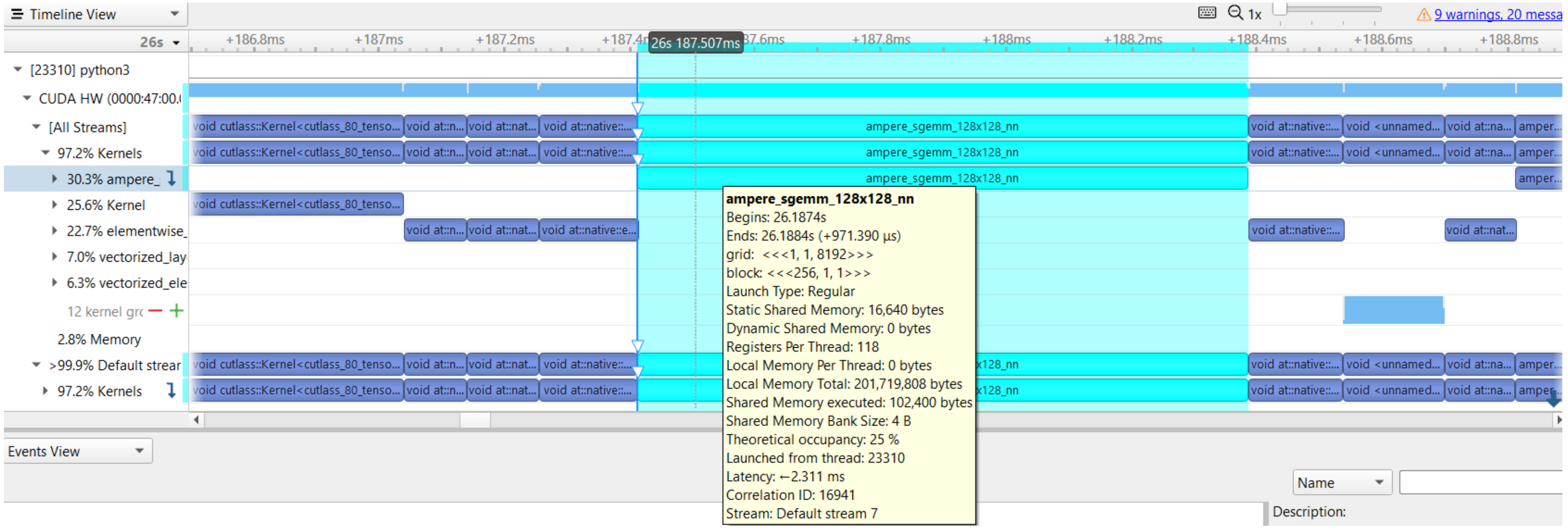


Figure 1. Flowchart describing working with new NVIDIA Nsight tools for performance optimization

[NVIDIA Developer Tools Overview | NVIDIA Developer]

Nsight System

- Used to get system information during execution
  - Kernel latency, timeline etc
- Can be used as a gui



Nsight System

Command Line: Generating a report

- nsys profile -w true -t cuda,nvtx,osrt,cudnn,cublas -s none -o report_output_name -f true -x true --capture-range=cudaProfilerApi --capture-range-end=repeat python3 train.py
  - nsys profile – runs nsight system
  - -w true: send output to std out
  - -t cuda,nvtx,osrt,cudnn,cublas: trace these api calls
  - -s none: disable cpu sampling. Could turn on if care about CPU code
  - -o report_output_name: the name of the report file to make
  - -f true: force the overwrite if filenames exist
  - -x true: stop profiling on exit
  - --capture-range=cudaProfilerApi: how to capture. This mode is set up for a call to start in code. Could be from the beginning
  - --capture-range-end=repeat: what to ends the capture range. This says we can repeatedly turn it on and off with a call in the python code
  - python3 train.py: command to run nsys profile on

Nsight System

Generate Stats

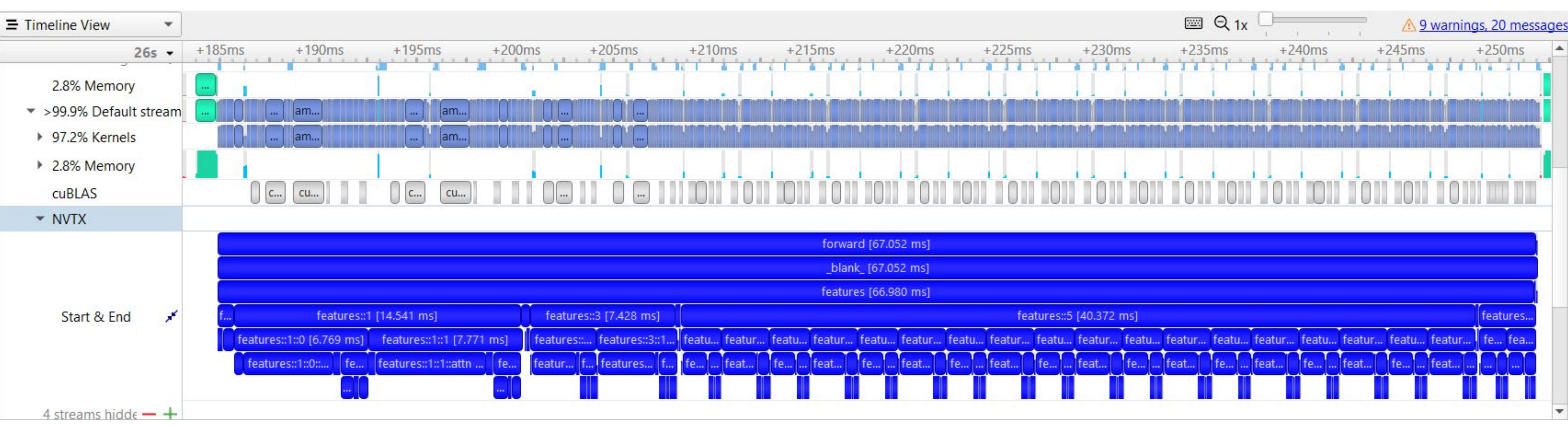
- nsys stats --force-export --force-overwrite --format csv --output . --report cudaapisum,cudaapitrace,nvtxkernsum,nvtxsesum,gpumemsizesum,gpumemtimesum,gputrace,gpukernsum report_output_name.nsys-rep
  - nsys stats: run stats generation
  - --force-export: force to read report file and not use pregenerated sql file
  - --force-overwrite: force overwriting output if already exists
  - --format csv: output format
  - --output .: the output files
  - --report cudaapisum,cudaapitrace,nvtxkernsum,nvtxsesum,gpumemsizesum,gpumemtimesum,gputrace,gpukernsum: the reports to generate
  - report_output_name.nsys-rep: nsys profile result to use for generating stats

Time (%)	Total Time	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns	Name
22	2.39E+08	5300	45135.4	40960	10977	153919	39208	void cudnn::bn_fw_inf_1C11_kernel_NCHW<float, float, (bool)1, (int)1>(T2, T2, cudnnTensorStruct, const T1 *, cudnnTensorStruct, T1 *, cudnnTensorStruct, const T2 *
15.9	1.72E+08	2300	74873.1	64383.5	34528	121599	19662.9	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_16x4_nn_align4>(T1::Params)
12.6	1.36E+08	4900	27840.4	10816	5345	128256	32642.9	void at::native::vectorized_elementwise_kernel<(int)4, at::native::<unnamed>::launch_clamp_scalar(at::TensorIteratorBase &, c10::Scalar, c10::Scalar, at::native::deta
11.5	1.24E+08	4800	25914.9	11904	5408	130528	29270.2	void cudnn::ops::nchwToNhwcKernel<float, float, float, (bool)0, (bool)1, (cudnnKernelDataType_t)2>(cudnn::ops::nchw2nhwc_params_t<T3>, const T1 *, T2 *)
11.2	1.21E+08	1600	75810	45216	16736	178304	54248	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float>>, at::detail::Array<char *, (int)3>>(int

Gpukernsum csv output example for ResNet50. Top 5 kernels.

NSYS GUI

w/NVTX Regions



NVIDIA.

Nsight Compute

Detailed kernel info

- Used to get kernel specific information
- High overhead
  - only use on hot spots or be ready to wait. Can take over 10x longer to execute
  - Serializes some execution meaning can't give full picture of parallel utilization
- NCU Commands
  - Generate analysis example:
    - ncu --set full -f -o ncu_report_output_file_name python train.py
  - Query metrics that can be gathered:
    - ncu --query-metrics
  - Gathered specific kernel metrics example:
    - ncu --csv --nvtx -f --print-nvtx-rename kernel --target-processes all --fp --print-kernel-base demangled --print-units base --profile-from-start off --clock-control base --print-summary per-nvtx --log-file profile_name_ncu_metrics.csv --metrics sm__cycles_elapsed.sum,sm__cycles_active.sum,dram__bytes_read.sum,dram__bytes_write.sum python train.py

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Domain	Range:PL_	Kernel Nar	Block Size	Grid Size	Device Id	Invocation	Section Na	Metric Name	Metric Uni	Minimum	Maximum	Average
2	<default d	"forward:	void at::na	512	4096	0	1	Command	dram__bytes_read.sum	byte	12894208	12894208	12894208
3	<default d	"forward:	void at::na	512	4096	0	1	Command	dram__bytes_write.sum	byte	43264	43264	43264

NVTX Regions

Annotation Tool for Profiling Code in Python and C/C++

- Allows you to insert named regions into code
- Python package available (https://pypi.org/project/nvtx/): pip install nvtx

```
rng = nvtx.start_range(message="my_message", color="blue")
# ... do something ... #
nvtx.end_range(rng)
```

- Nsight tools understand these regions
  - Can generate reports for nvtx regions
  - Helps you pinpoint areas that need improvement

layer1	StartEnd	727882	727882	1	10	817501	81750.1	41280	39424	144896	52978.6	void cudnn::bn_fw_inf_1C11_kernel_NCHW<float, float, (bool)1, (int)1>(T2, T2, cudnnTensorStruct, const T1 *, cudnnTe
layer1	StartEnd	727882	727882	1	7	573215	81887.9	88448	34976	93920	20902	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_16x4_nn_align4>(T1::Params)
layer1	StartEnd	727882	727882	1	3	527775	175925	176032	175520	176223	363.5	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BinaryFunctor<float, float, float, at::native::AddFunc
layer1	StartEnd	727882	727882	1	9	523325	58147.2	26336	24928	123296	48642.6	void at::native::vectorized_elementwise_kernel<(int)4, at::native::<unnamed>::launch_clamp_scalar(at::TensorIterator
layer1	StartEnd	727882	727882	1	3	303231	101077	100864	100767	101600	455.5	sm80_xmma_fprop_implicit_gemm_indexed_tf32f32_tf32f32_f32_nhwckrsc_nchw_tilesize256x64x32_stage3_warpsiz
layer1	StartEnd	727882	727882	1	6	107456	17909.3	17520	5760	31392	13317.9	void cudnn::ops::nchwToNhwcKernel<float, float, float, (bool)0, (bool)1, (cudnnKernelDataType_t)2>(cudnn::ops::nchw

Nvtxkernsum csv output example for ResNet50.

Profiling Tips

- GPU Considerations
  - Query the available clocks
    - nvidia-smi -q -d SUPPORTED_CLOCKS -i 0
  - Lock the clocks on the gpu
    - `nvidia-smi -lgc <minGPUclock>, <maxGPUclock>`
  - Release the clocks
    - nvidia-smi –rgc
- System Considerations
  - Stop nonessential tasks
  - Warmup the system
    - Run a few iterations to instantiate buffers and first-time setup things
  - Profile only the regions of interest
    - For GPU use cudaProfilerStart() and cudaProfilerStop() (Profiler Users Guide (nvidia.com)) (bshillingford/python-cuda-profile (github.com))
  - Proper synchronizations to ensure done
    - In torch call `torch.cuda.synchronize() to ensure GPU done for benchmarking`

Thank you

Questions?

Jason Clemons

jclemons@nvidia.com

Disclaimer: Results, numbers and performance are reported from the research perspective
For the exact performance please contact NVIDIA product managers