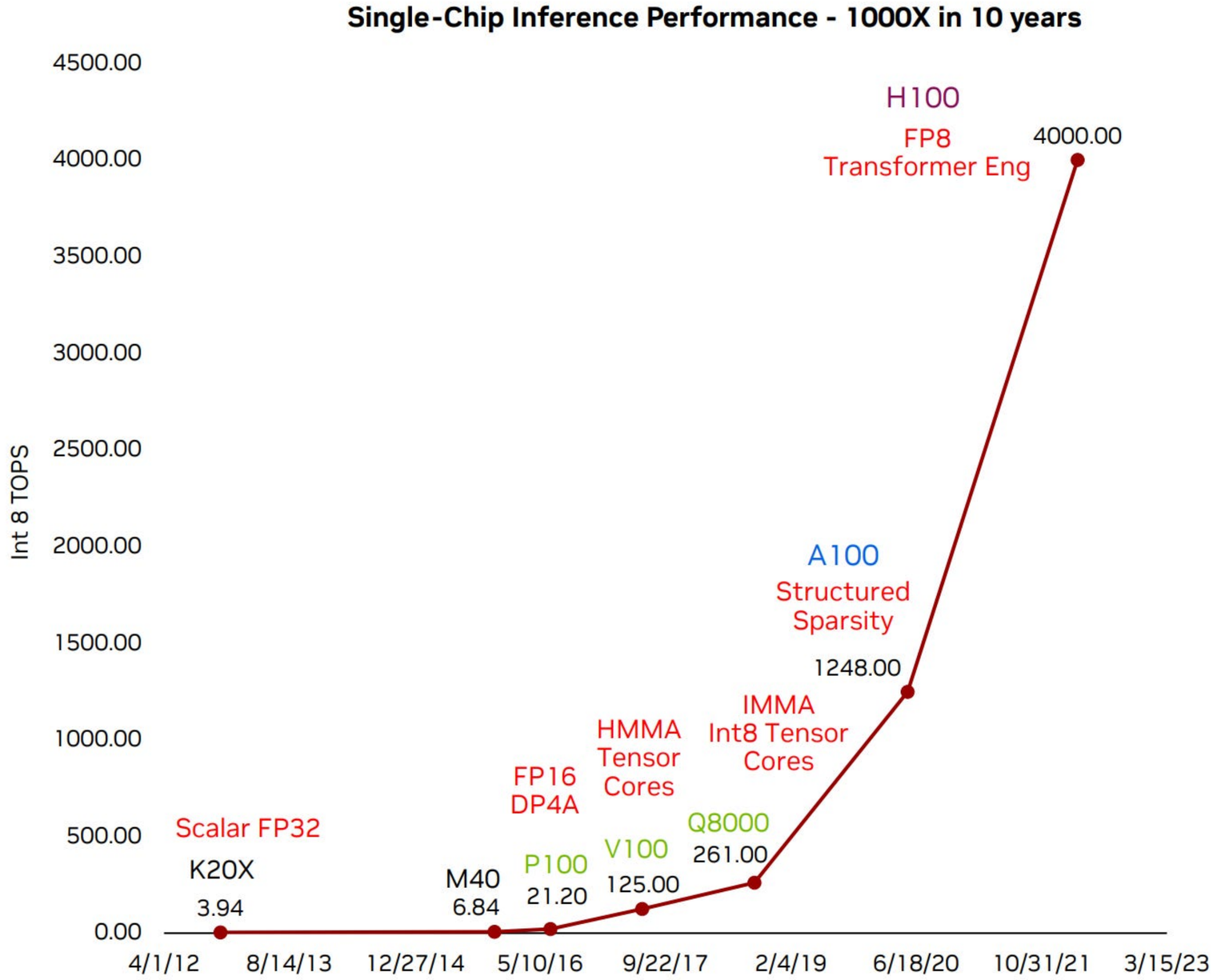
Narrow bit-width number formats for deep learning

The context: Quantization in Deep Learning

- The basic "atom" of computation in a ML model is matrix multiplication.
- We have a choice on the number of bits used to represent the "weights" in our model (i.e., the learnable parameters in the matrices that we are multiplying the input by).
- Lowering the number of bits used lowers the possible precision of our calculations, and hence the accuracy of our model.
- On the other hand, it speeds up the training and inference of the model while improve its memory and power efficiency.
- In practice, the accuracy loss can be managed and this is an acceptable tradeoff.

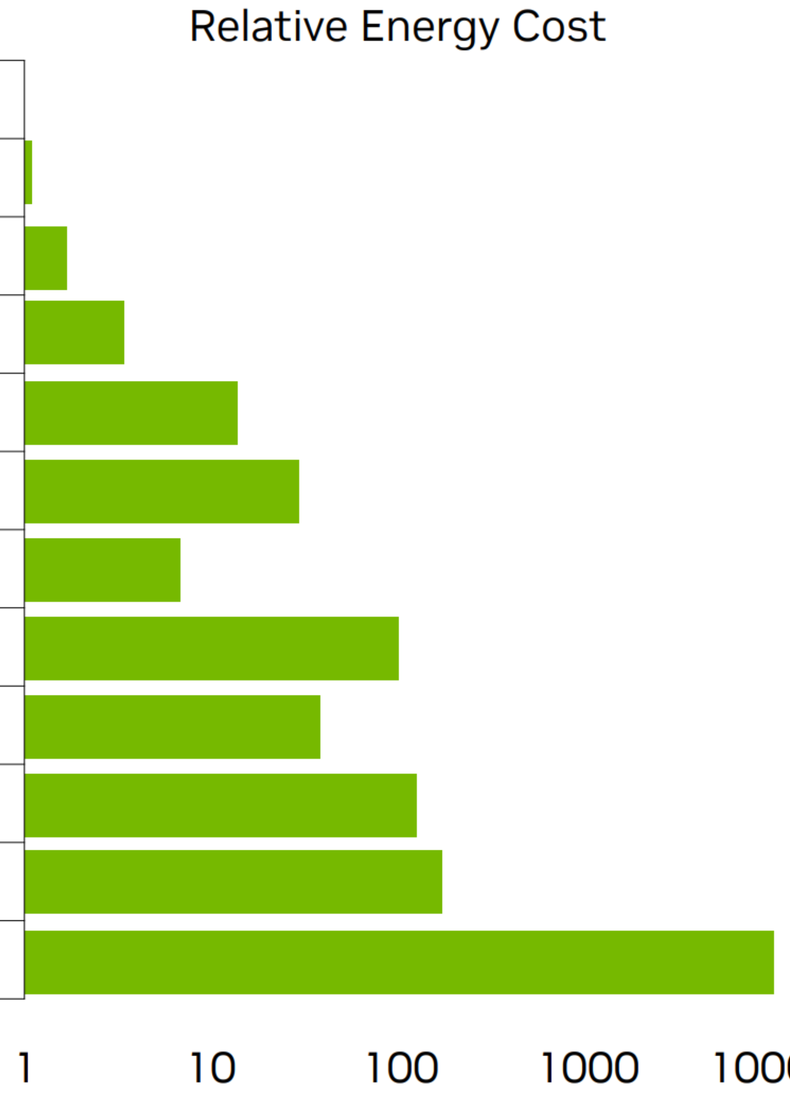
Gains from

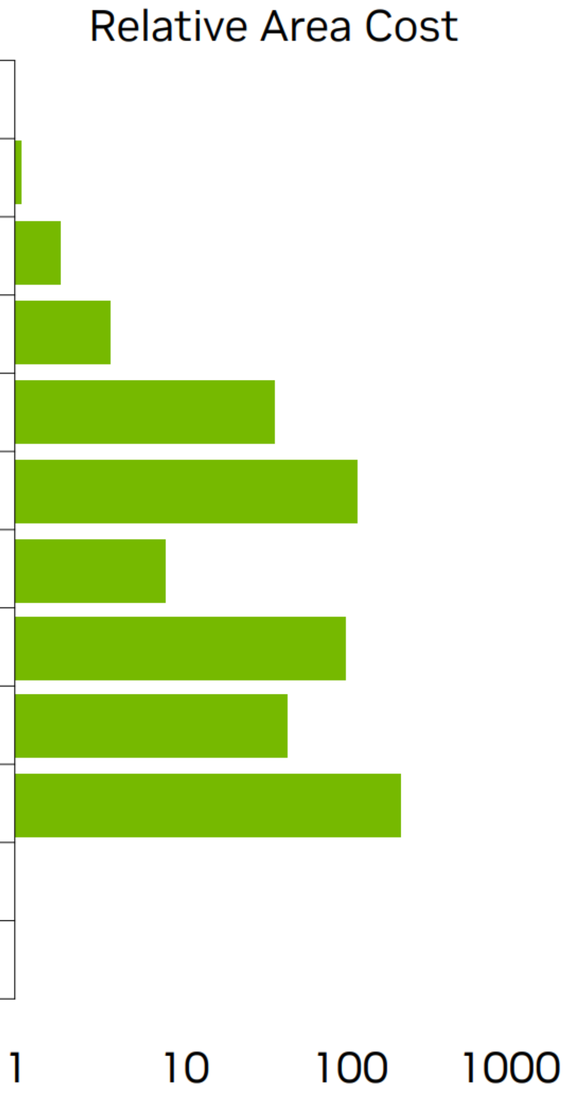
- Number Representation
  - FP32, FP16, Int8
  - (TF32, BF16)
  - ~16x
- Complex Instructions
  - DP4, HMMA, IMMA
  - ~12.5x
- Process
  - 28nm, 16nm, 7nm, 5nm
  - ~2.5x
- Sparsity
  - ~2x
- Model efficiency has also improved – overall gain > 1000x



Cost of Operations

Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A





Energy numbers are from Mark Horowitz “Computing’s Energy Problem (and what we can do about it)”, ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Basics of number formats

- Numbers are stored in binary representation on a computer.
- For a floating point number, you have a sign bit, and then bits allocated to both the *exponent* and *mantissa* (significand).
- For example, the IEEE 754 standard specifies the *single-precision floating point* format FP32 as having 1 sign bit, 8 exponent bits, and 24 mantissa bits.

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

- When reducing the number of bits used (always using a power of 2), we have a choice on how to allocate bits to the exponent and mantissa.
- Different choices may work better for training vs. inference (range vs. precision).

Basic Number Formats				
Format	Sign	Exponent	Mantissa	Total bits
FP64	1	11	52	64
FP32	1	8	23	32
TF32	1	8	10	19
FP16	1	5	10	16
BF16	1	8	7	16
UINT8	0	0	8	8
INT8	1	0	7	8
FP8 (e5)	1	5	2	8
FP8 (e4)	1	4	3	8
FP8 (e3)	1	3	4	8
UINT4	0	0	4	4
INT4	1	0	3	4

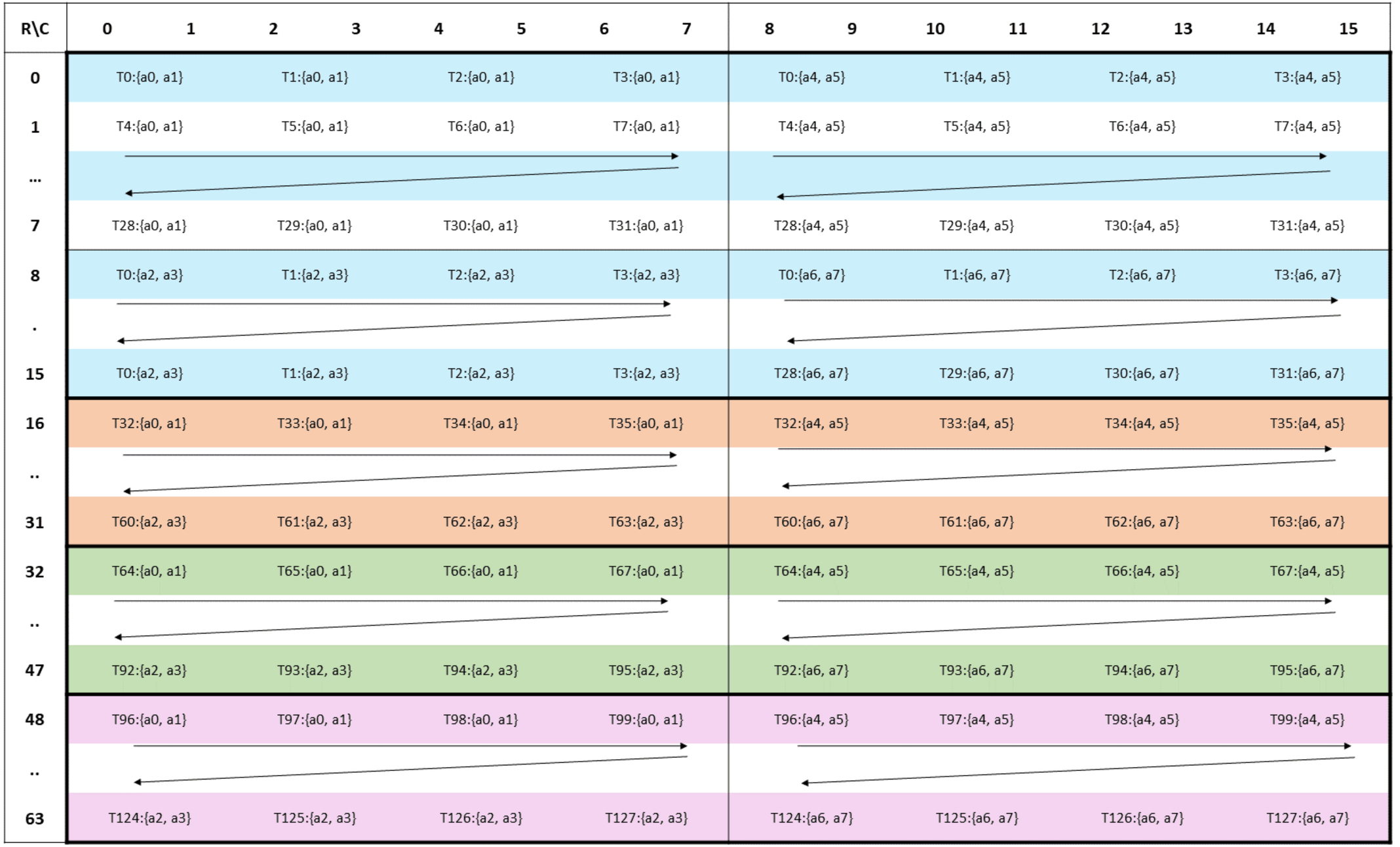
Hopper Tensor Cores by Precision Type

- The H100 GPU has native support for number formats down to 8 bits.

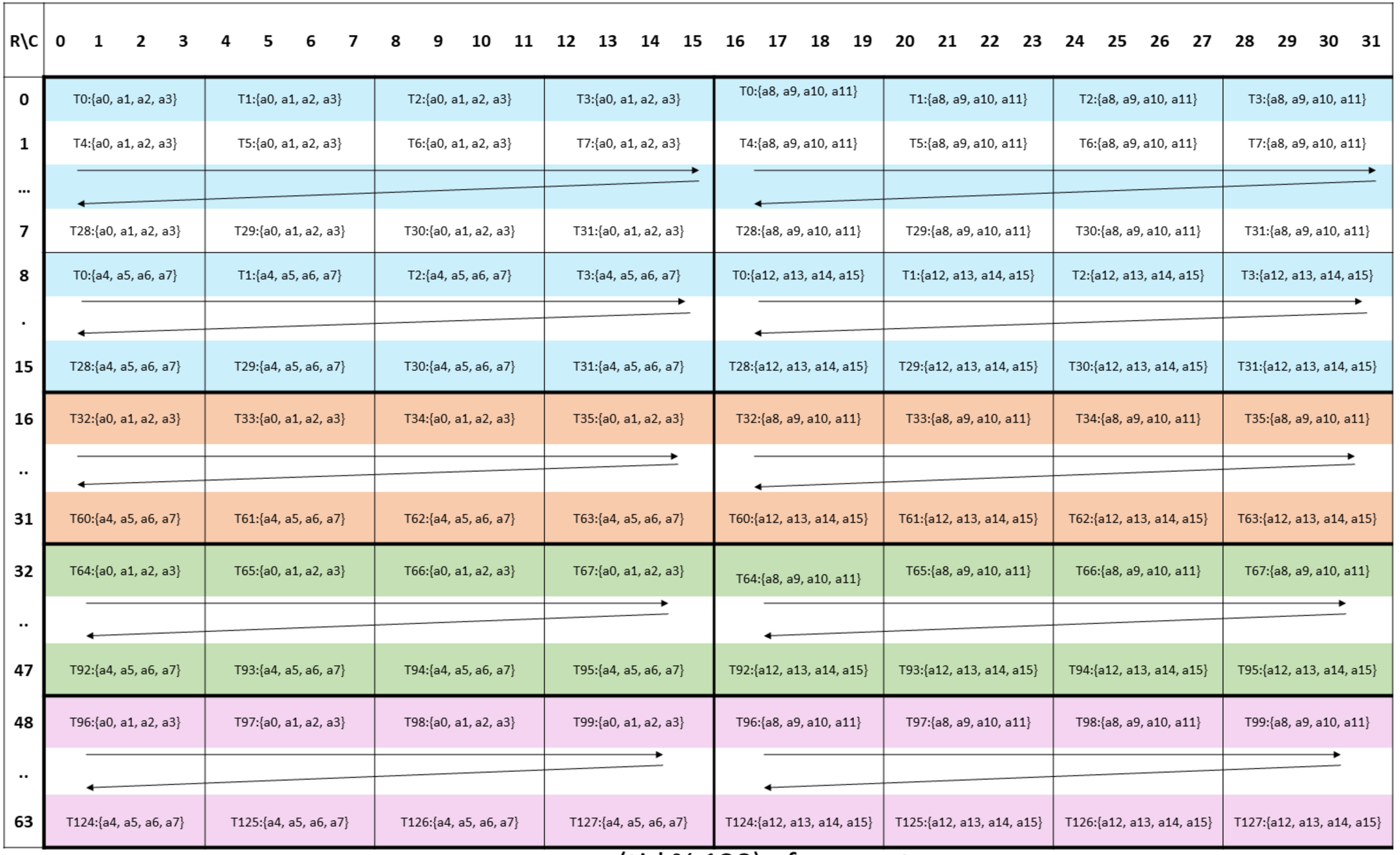
	H100 SXM	H100 PCIe	H100 NVL[1]
FP64	34 teraFLOPS	26 teraFLOPS	68 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS	134 teraFLOPS
FP32	67 teraFLOPS	51 teraFLOPS	134 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]	756 teraFLOPS[2]	1,979 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]	3,026 teraFLOPS[2]	7,916 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]	3,026 TOPS[2]	7,916 TOPS[2]

Targeting the Hopper Tensor Cores in CUDA

- On Hopper, we have WGMMA (Warpgroup Matrix Multiply Accumulate) for doing accelerated matrix multiplications C = A*B.
- WGMMA corresponds to the instruction *wgmma.mma_async* in the PTX ISA.
- In *wgmma.mma_async*, one warpgroup (=128 threads) cooperatively performs a matrix-multiplication on tiles of the matrices A and B (of certain allowed dimensions, like {M,N,K} = {64,64,16}).
- *wgmma.mma_async* makes assumptions on the layouts of the fragments of the operand matrices held by the different threads.



(tid % 128) : fragments



(tid % 128) : fragments

WGMMA continued

- WGMMA operations are *mixed-precision*: operand matrices are lower bit-width than the accumulator matrix (e.g., FP16 or FP8 operand matrices with FP32 accumulator).
- Mixed-precision should be contrasted with *mixed-input* where the A operand is of a different bit-width than the B operand, e.g. as in weight-only quantization.
- The ISA doesn't currently support mixed-input wgmma.
- However, the CUDA programmer can support mixed-input wgmma on the software side through upcasting the narrower bit-width operand.
- Then, *layout conformance* becomes a challenge with using wgmma.

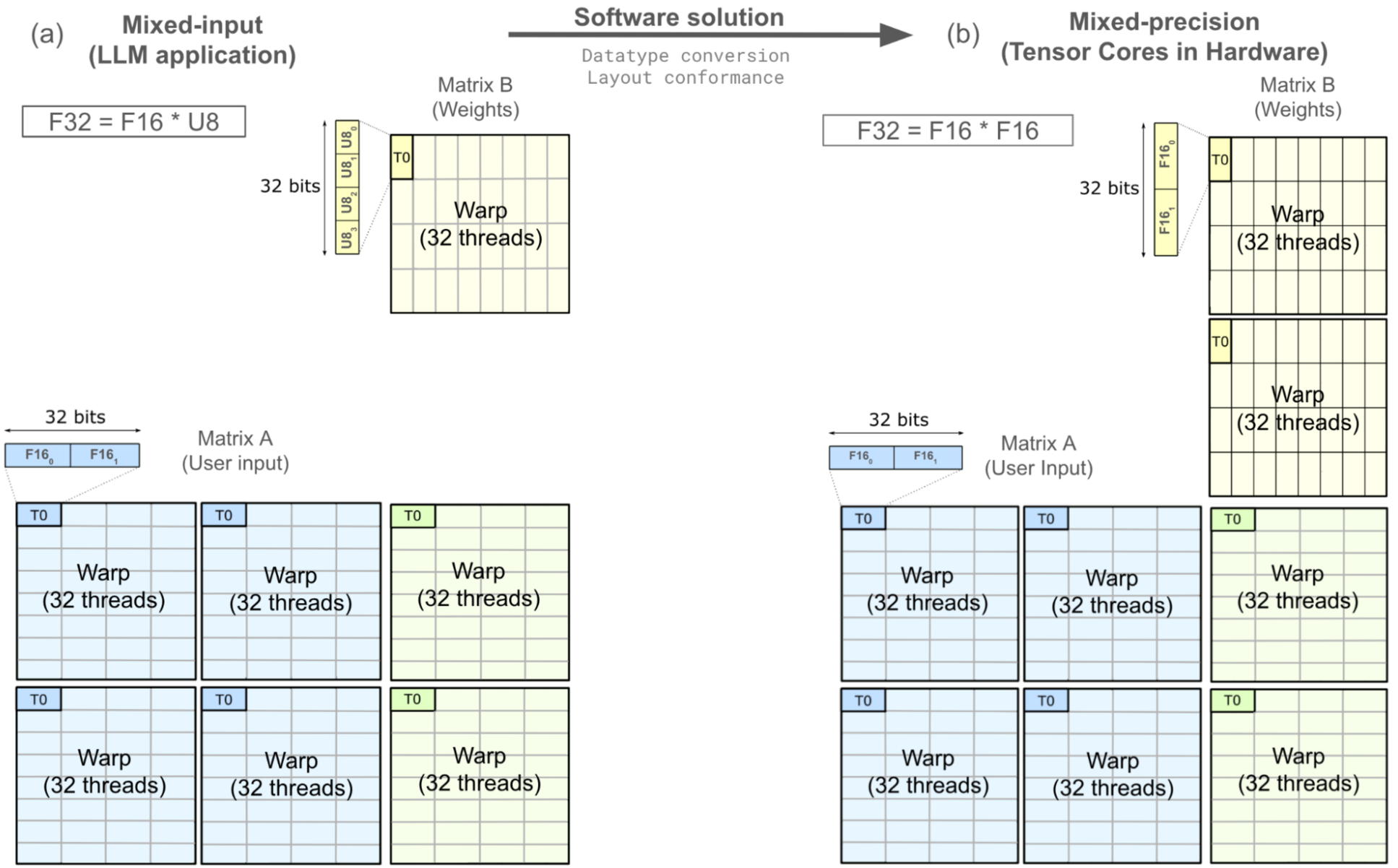


Illustration of upcasting 8 to 16 bits in mixed-input GEMM. Source: "Developing CUDA kernels to push Tensor Cores to the Absolute Limit on NVIDIA A100."

References

- "Neural Network Quantization & Number Formats From First Principles", Dylan Patel. https://www.semianalysis.com/p/neural-network-quantization-and-number

- "Mixed-input matrix multiplication performance optimizations", Manish Gupta. https://blog.research.google/2024/01/mixed-input-matrix-multiplication.html