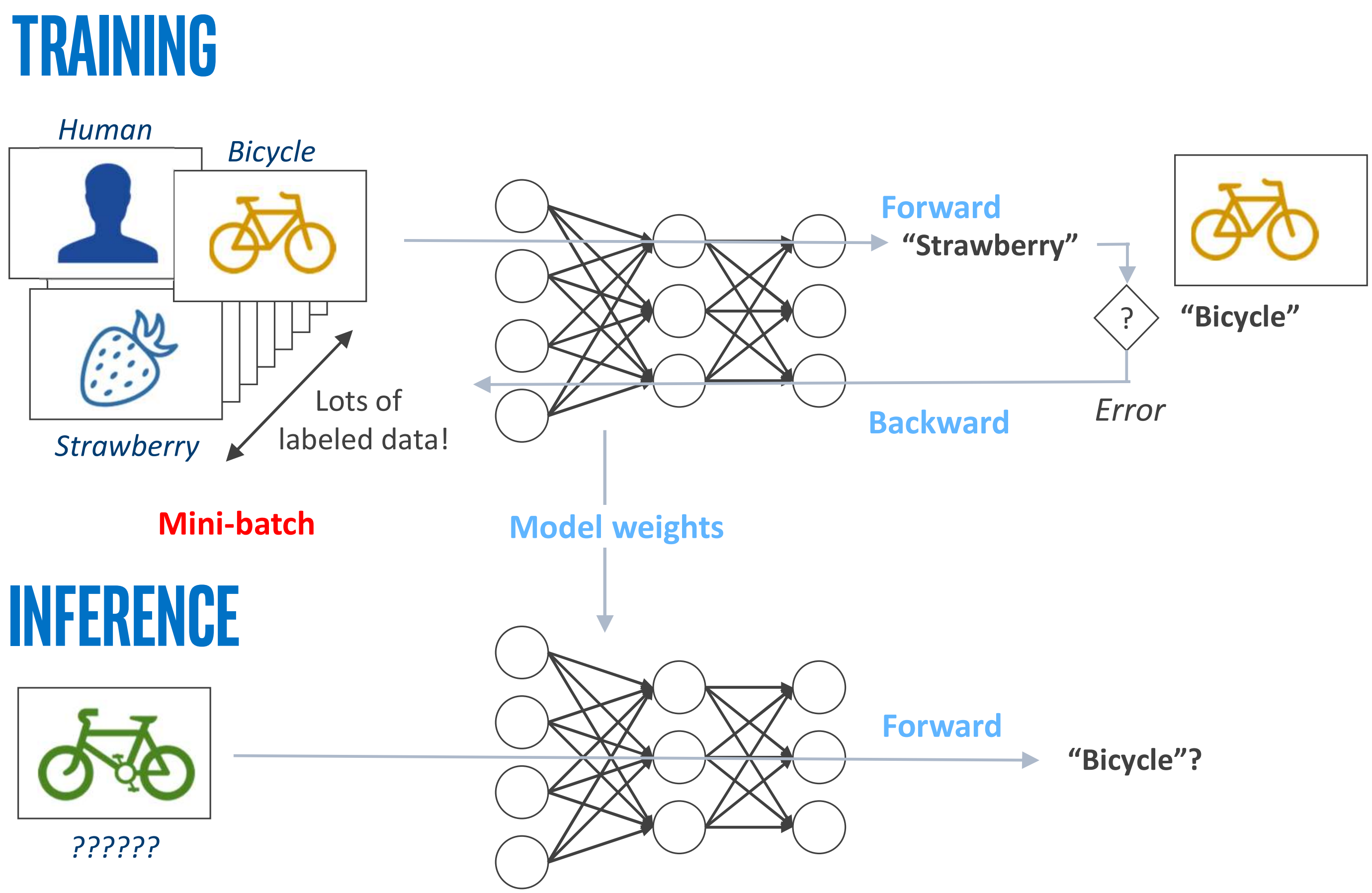
AI计算优化简介

林晓东

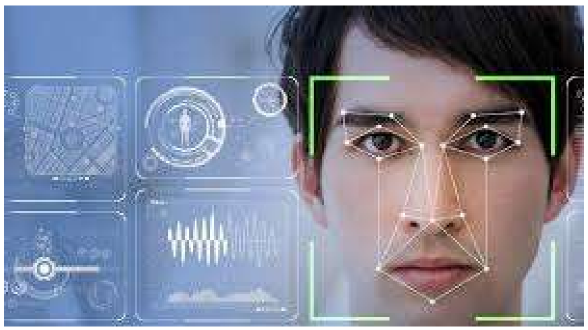
英特尔公司

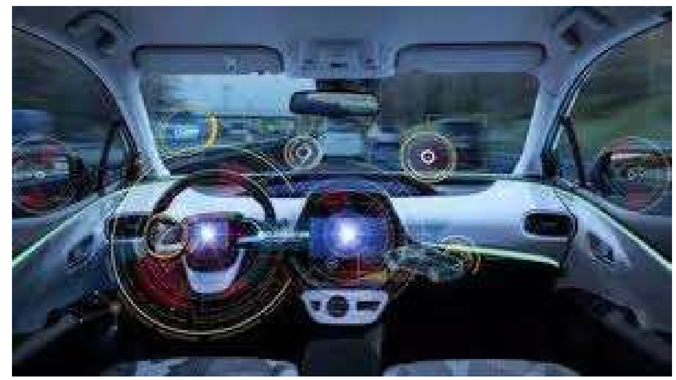
eric.lin@intel.com

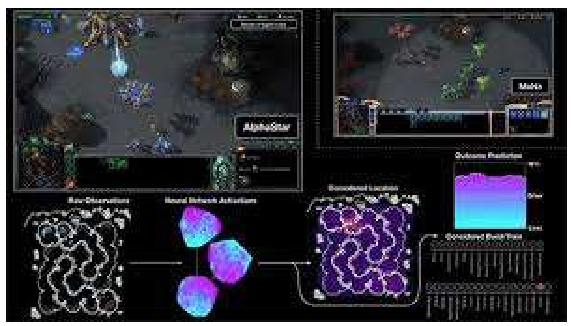
Deep Learning Basics



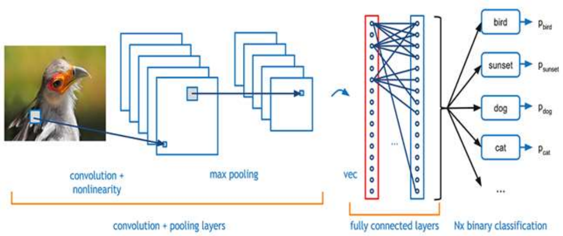
Deep Learning: What End Users See





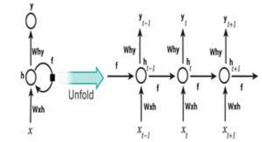


Deep Learning: What Data Scientists See



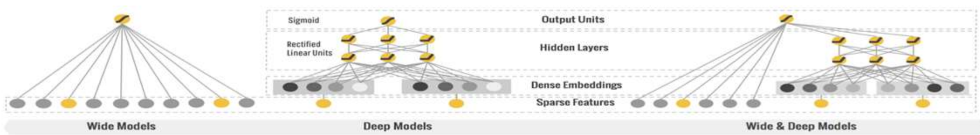
https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

Convolutional Neural Network (CNN)

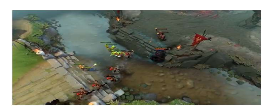


https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/

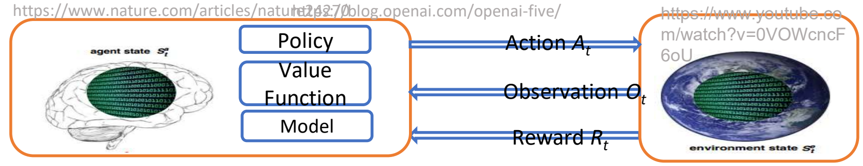
Recurrent Neural Network (RNN)



[Cheng et al. Wide & Deep Learning for Recommender Systems. DLRS @ RecSys 2016.]



https://www.nature.com/articles/nature24270 https://blog.openai.com/openai-five/ https://www.youtube.com/watch?v=0VOWcncF6oU



Recommendation Systems

Reinforcement Learning

Deep Learning: What DL Engineers See

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
   for(int m = 0;  m < M;  m++)
      for(int h = 0; h < H/K; h++)
        for(int w = 0; w < W/K; w++) {
          S[m, x, y] = 0.;
          for(int p = 0; p < K; p++) {
             for(int q = 0; q < K; q++)
                S[m, h, w] += Y[m, K*h + p, K*w + q] /(K*K);
          }
          S[m, h, w] = sigmoid(S[m, h, w] + b[m])
        }
}
```

DL Ops are just normal codes, except they are hungrier for TFLOPS & memory bandwidth

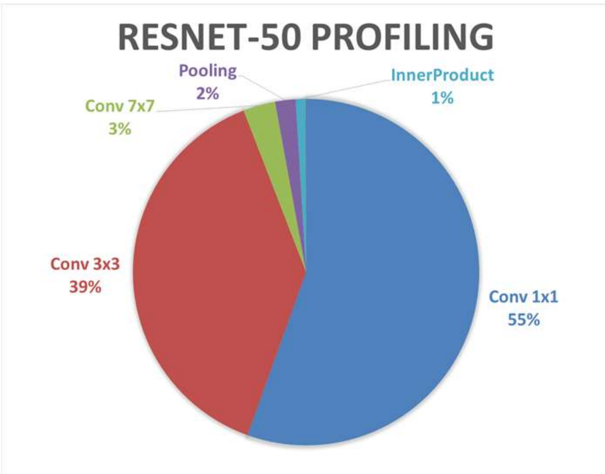
```
void conv_forward(int B, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
  int H_out = H - K + 1;
  int W_out = W - K + 1;

  for (int b = 0; b < B; ++b)
    for(int m = 0;  m < M;  m++)
      for(int h = 0; h < H_out; h++)
        for(int w = 0; w < W_out; w++) {
          Y[b, m, h, w] = 0;
          for(int c = 0;  c < C; c++)
            for(int p = 0; p < K; p++)
              for(int q = 0; q < K; q++)
                Y[b, m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];
        }
}
```

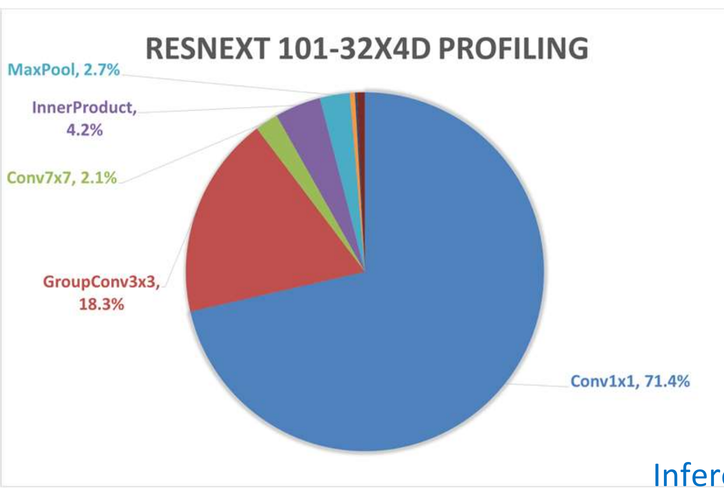
mul & add: GEMM, conv, RNNCell

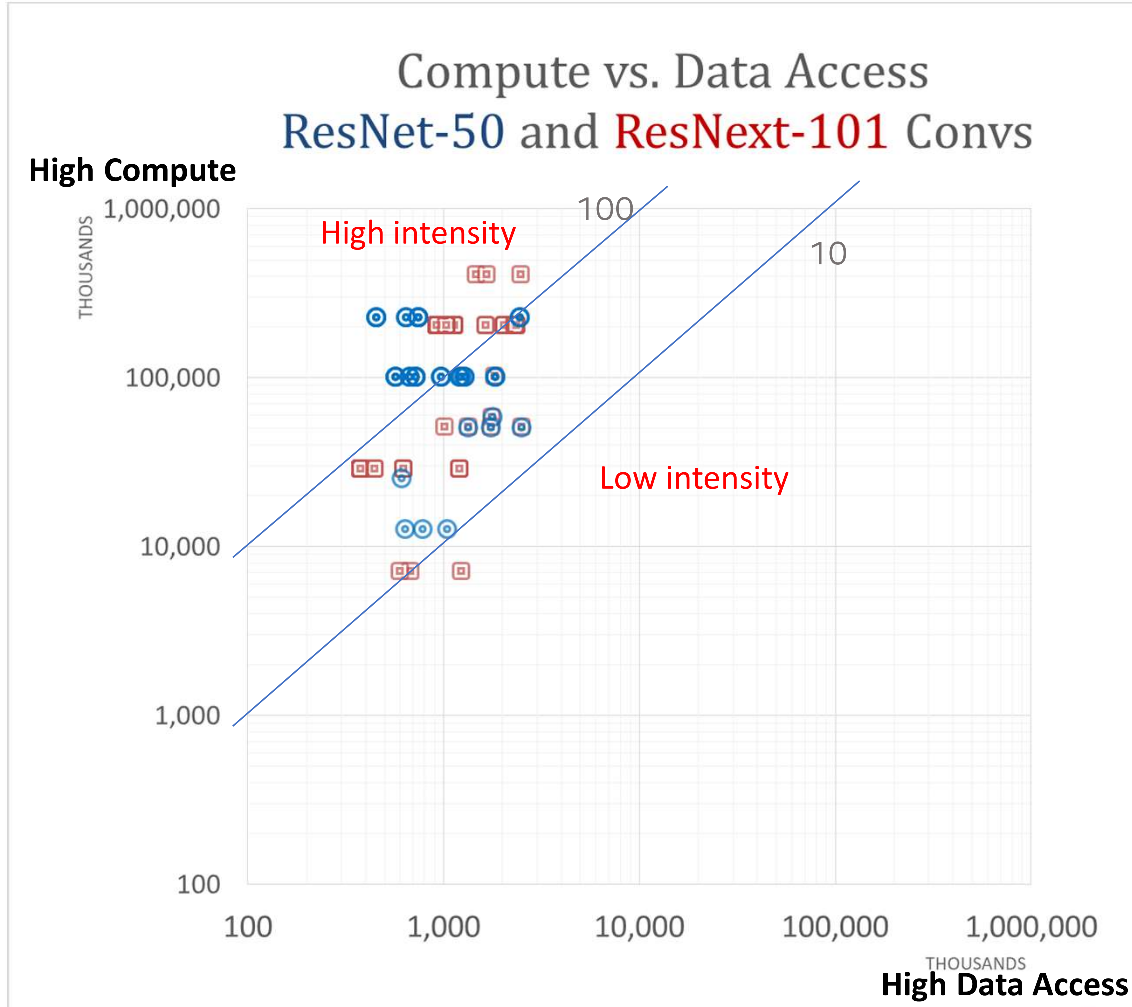
memory: embedding, transpose, concat, normalization, element-wise, broadcast, transpose

Typical Workload

Computer Vision

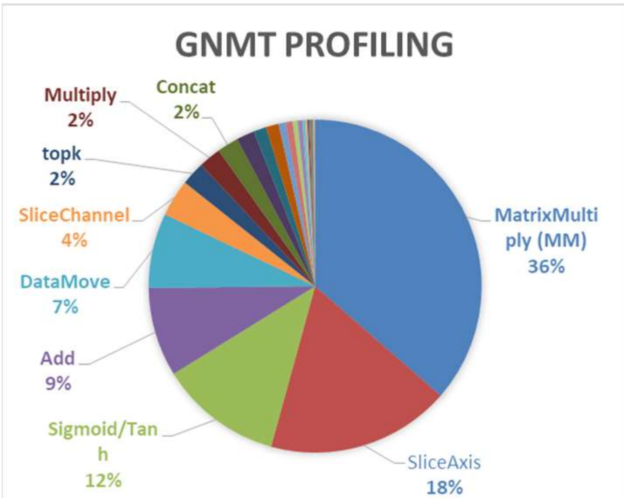


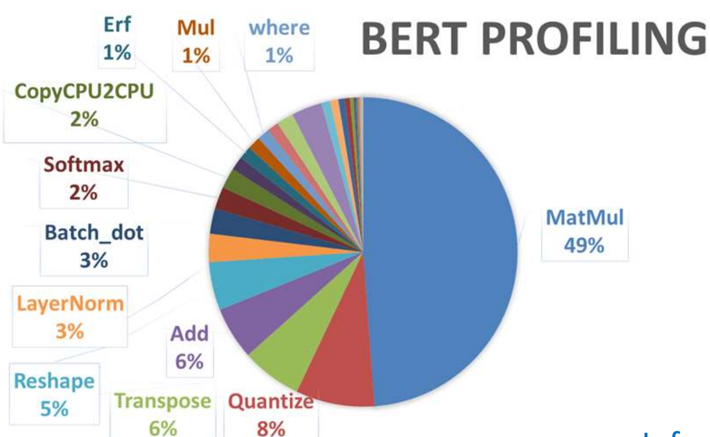


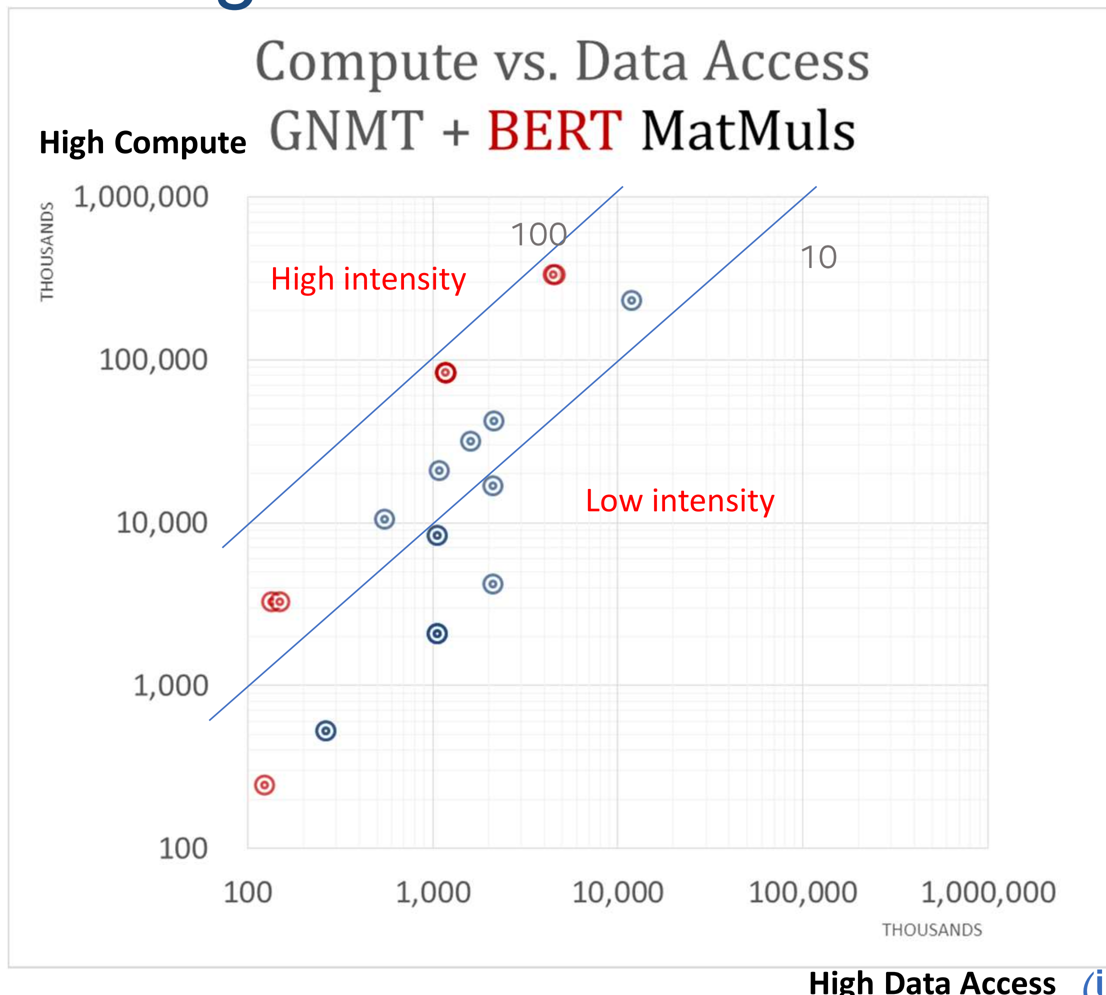


Inference, BS = 1

Natural Language Processing

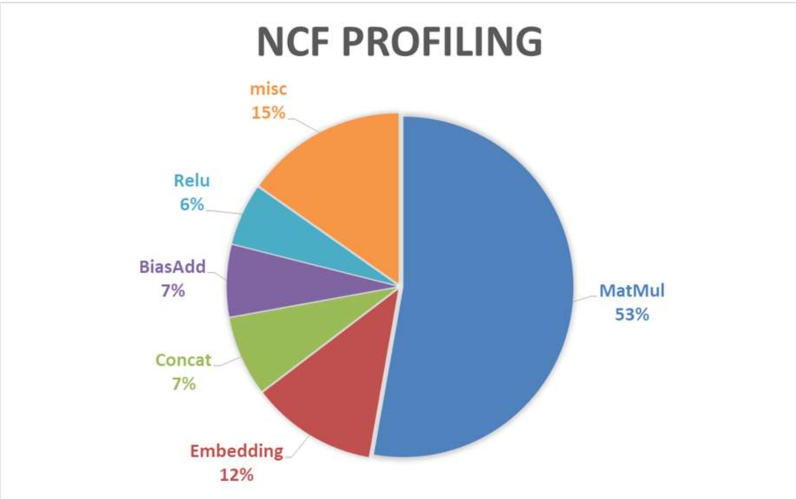


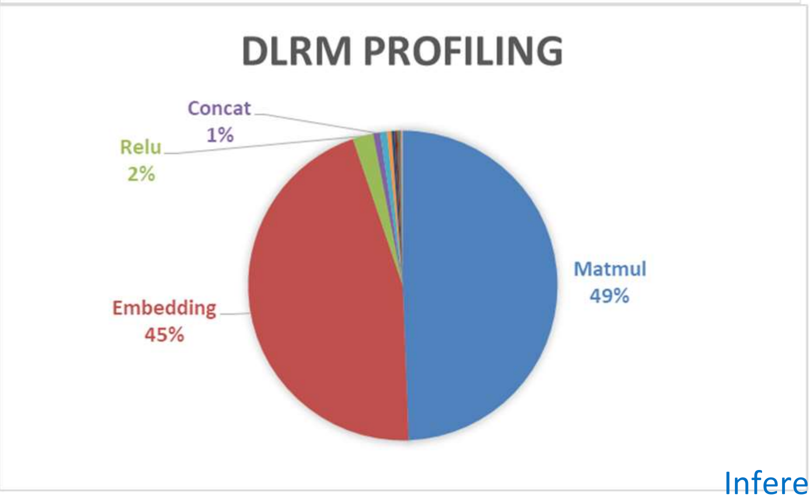


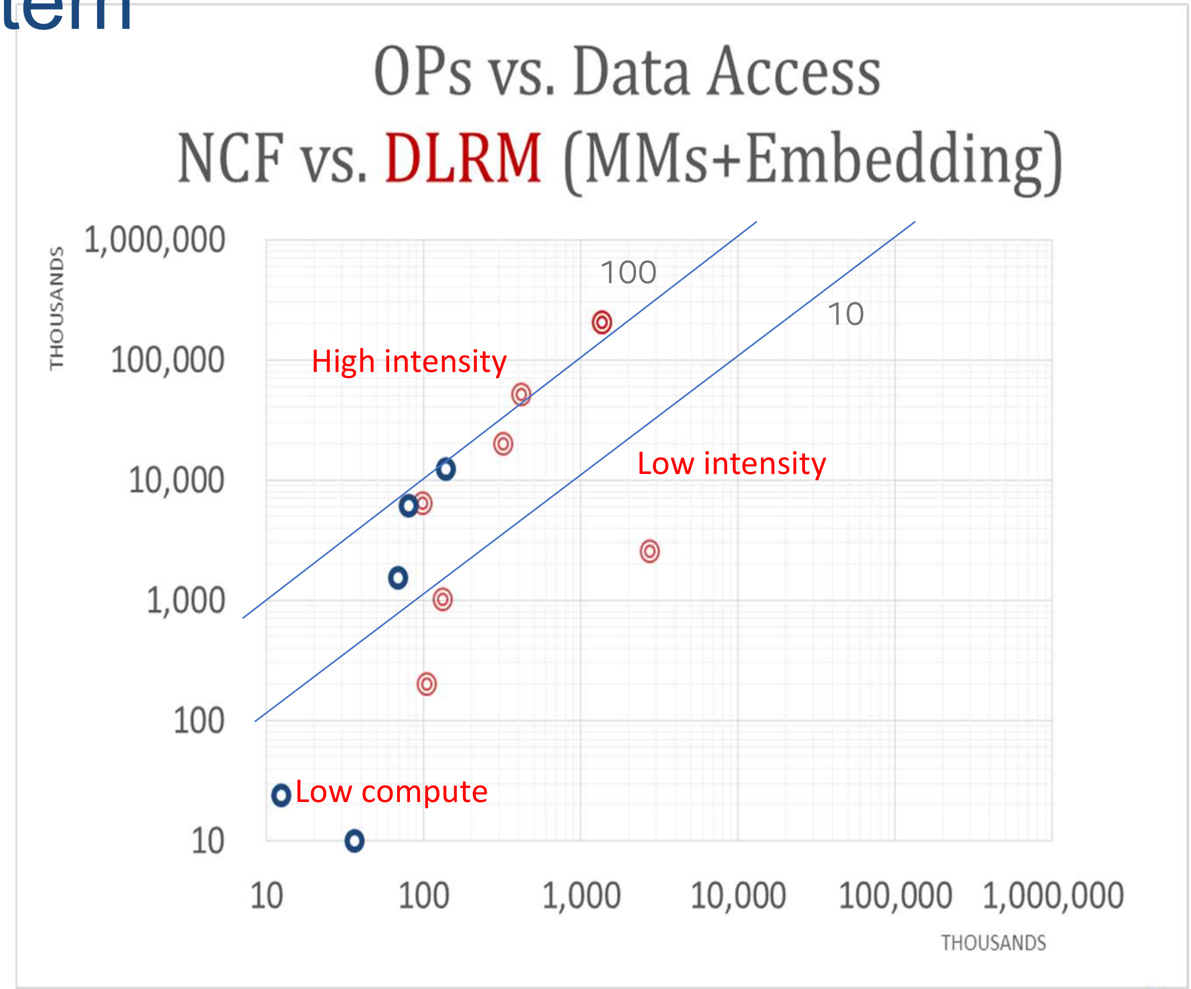


Inference, BS = 1

Recommendation System

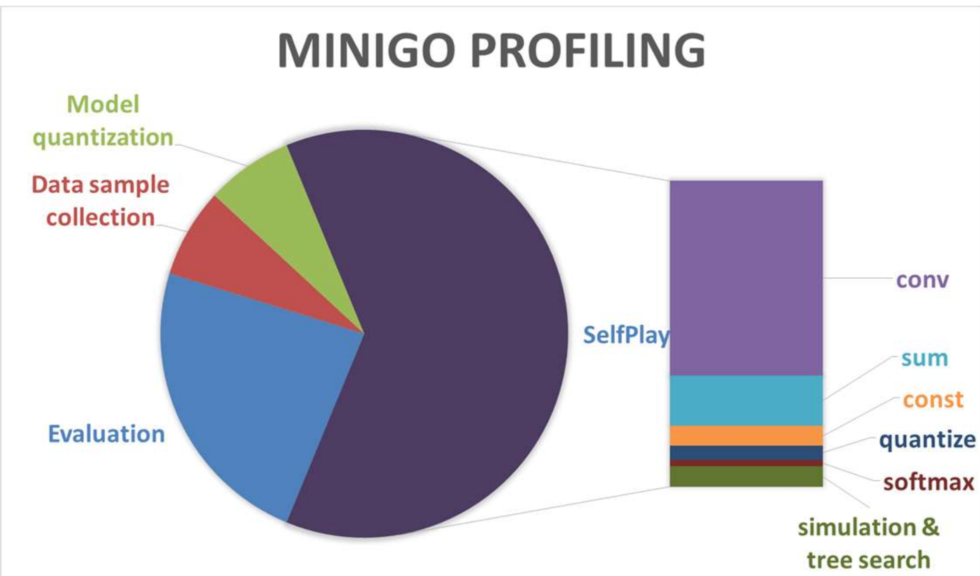


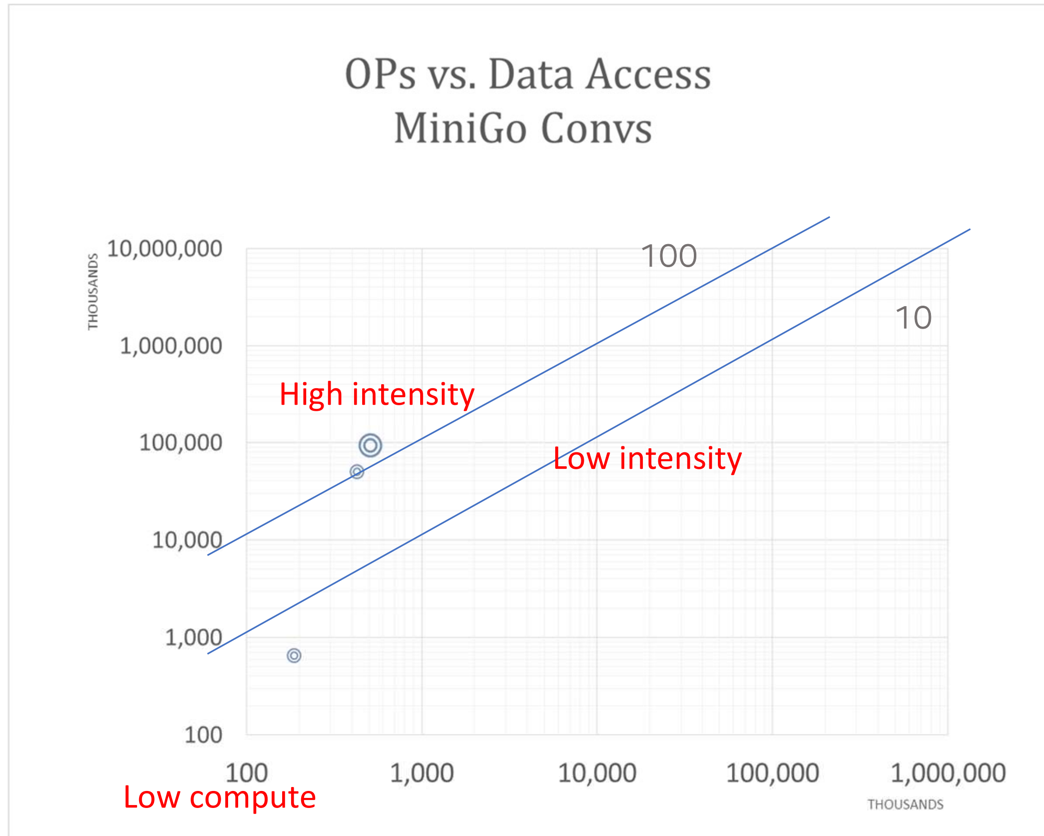




Inference, BS = 100

Reinforcement Learning

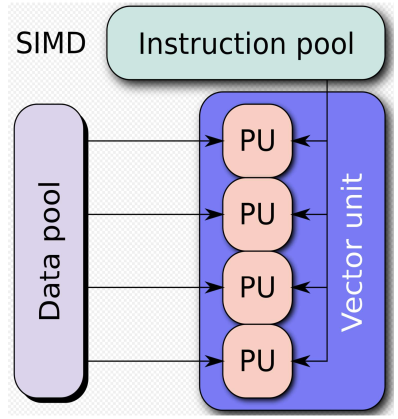


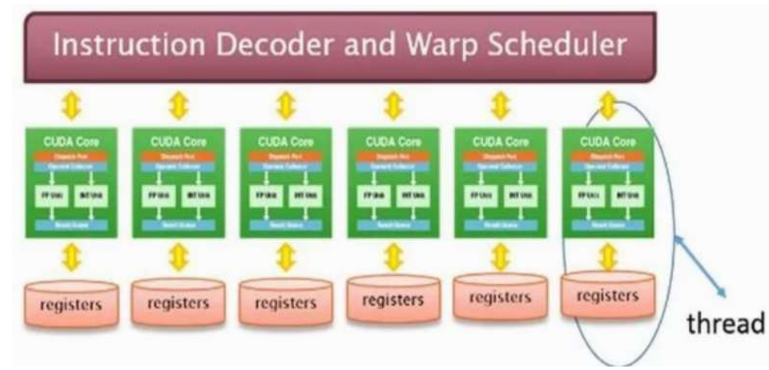


Inference, BS = 64

Primitive Level Optimization

Optimization Basis

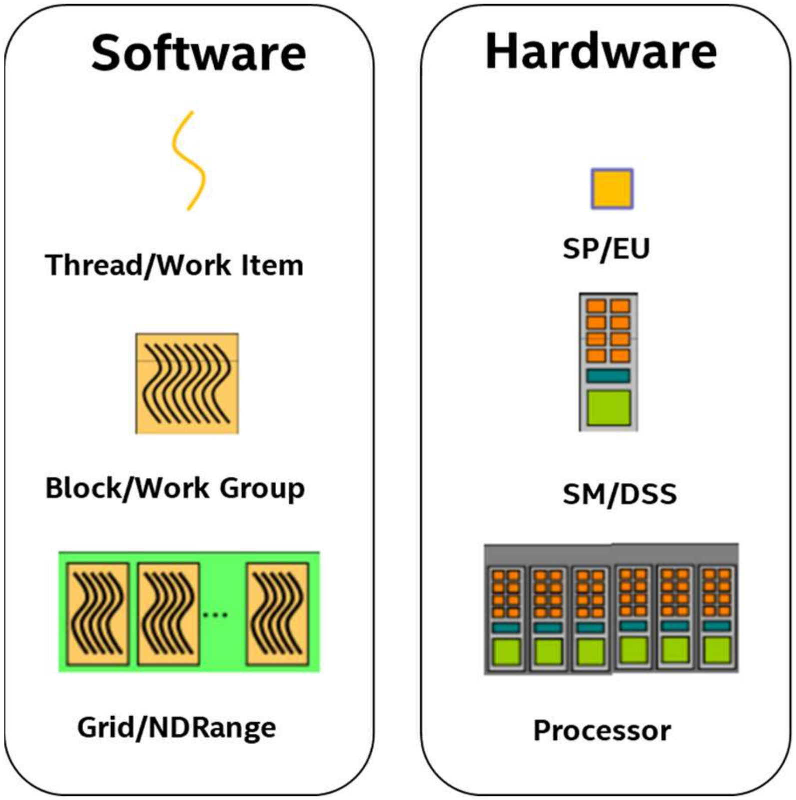




- Optimize for parallelism
  - Vectorization (SIMD)
  - Multiple thread (SIMT)
  - SIMT + SIMD Combination
  - Multiple processors (cores, SMs)
- Optimize for memory hierarchy: reduce & hide the latency; utilize the bandwidth
  - Multiple level cache
  - Local memory (addressable cache)
  - Memory Coalescing
  - Avoid bank conflict
  - NUMA

- Programming Language express the parallelism: OpenMP, SYCL/DPC++, OpenCL, CUDA …
- There are no essential difference between SIMD & SIMT on high performance code

Parallel Programming Language



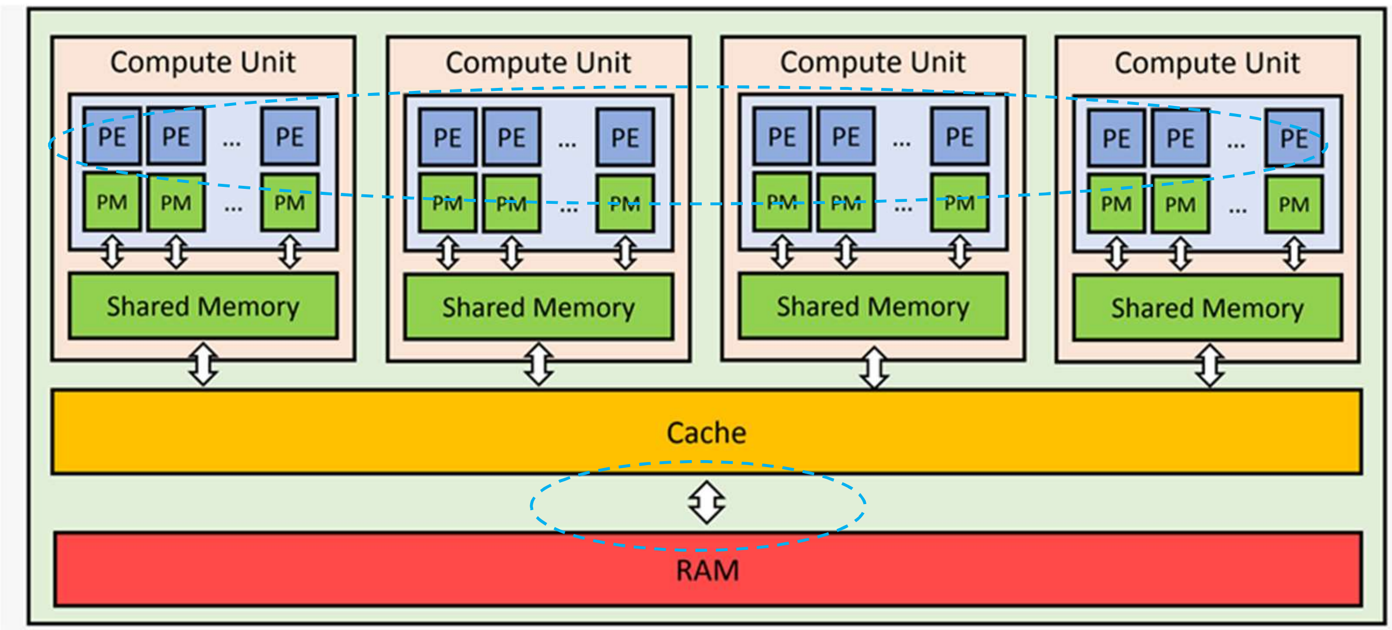
CUDA	SYCL/DPC++
SP	Process Element
SM	Compute Unit
Thread	Work item
Block	Work group
Grid	NDRange

Optimization Goal

Achieve HW peaks

- TFLOPS/s: for compute bounded ops/kernels
- Memory bandwidth: for memory bounds ops/kernels

TFLOPS



Memory bandwidth

Make it Parallel: ReLU

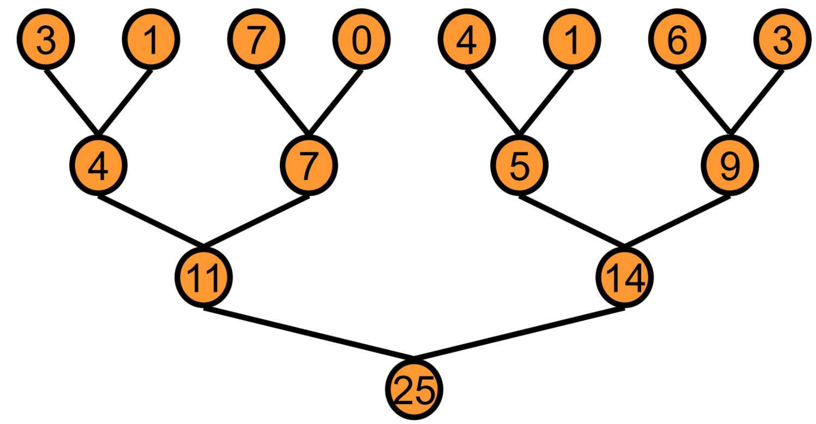
```
// Normal C++
for (i = 0; i < n; i++) {
   if (data[i] <= 0.0)
      result[i] = 0.0;
   else
      result[i] = data[i];
}
```

```
// DPC++/SYCL
parallel_for(range{n}, [=](id<1> i) {
   if (data[i] <= 0.0)
      result[i] = 0.0;
   else
      result[i] = data[i];
});
```

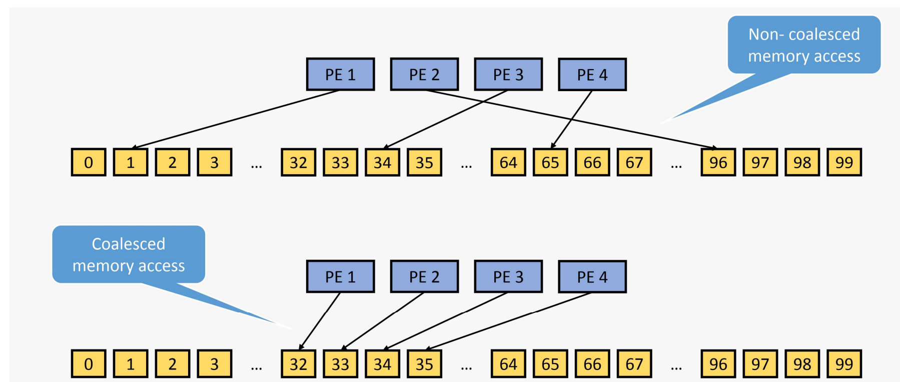
```
// CUDA
__global__ void ReLU(float *data, float *result)
{
   int i = threadIdx.x;
   if (data[i] <= 0.0)
      result[i] = 0.0;
   else
      result[i] = data[i];
}

ReLU<<<1, n>>>(data, result);
```

Parallel programming languages express parallelism explicitly so that compiler can do SIMT or SIMD optimization freely

Make it Parallel: Reduction



- Processing large dataset with associative and commutative operations (sum, product, max/min……): normalization, softmax are all reduction based
  - Partition the data set into smaller chunks
  - Each work-item/thread to process a chunk
  - Reduction tree to summarize the results from each chunk into the final answer
- log(N) steps, for data size N
  - Memory coalescing
  - Maximize HW utilization for each step (ensure there are enough tasks)
  - Stop recursive and unroll the loop when there is no enough tasks
  - Use shared local memory to hold partial result



Pitfall: memory ordering

When the partial result of other work-group is visible?

- Remember: partial result are in share local memory, need to store global memory to do final reduction
- How work-item 0 know all the store complete?

Seems a simple producer/consumer issue while we want it to be lock free, in parallel programming you always want to avoid heavy mutex/lock/semaphore overhead

```
// WG0, producer
payload = 1;
guard = 1;

// WG1, consumer
while (guard != 1)
assert(payload = 1)
```

assert(payload == 1) might fire!

Threads don't have to agree on the order of events; Operations on distinct variables can appear in different orders on different threads

- *Compiler may reorder instructions*
- *Processors may reorder instructions*
- *Processors may have caches, or other buffers*
- *Cache might not be consistent*

The correct version of producer/consumer

```
// SYCL
int payload = 0; atomic<int> guard(0);

WG0:
payload = 1;
guard.store(1, memory_order::release, memory_scope::device);

WG1:
while (guard.load(memory_order::acquire, memory_scope::device) != 1);
assert(payload == 1)
```

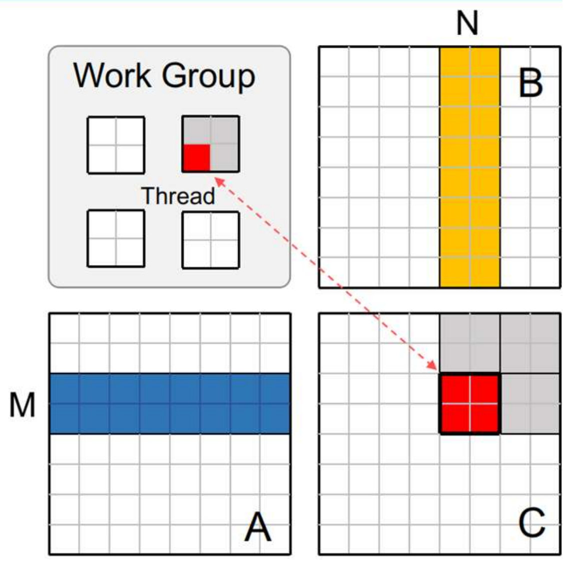
```
// CUDA
volatile int payload = 0;
volatile int guard = 0;

TG0 :
payload = 1;
__threadfence();
guard = 1;

TG1:
while (guard != 1);
__threadfence();
assert(payload == 1);
```

Compiler ensures it works as expected

There are other complexities on synchronization like barrier, independent forward progress/lockstep. Yes, parallel programming is difficult☺

GEMM: simple optimization

```
// naive implementation
for (i = 0; i < M; ++i)
   for (j = 0; j < N; ++j)
      for (k = 0; k < L; ++k)
         c[i][j] += a[i][k] * b[k][j];
```

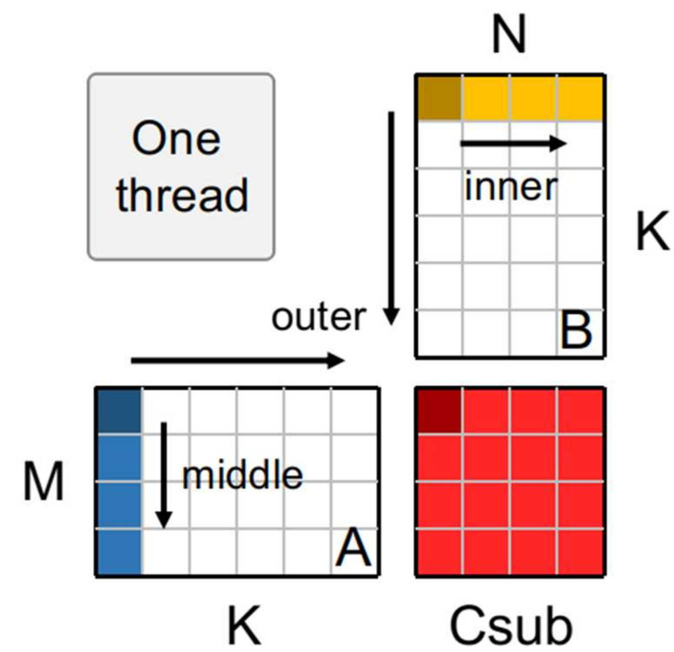



```
size_t row = index[0];
size_t col = index[1];
float csub[cm][cn] = {0.0f};
for (int m = 0; m < cm; ++m)
{
  for (int n = 0; n < cn; ++n)
  {
    for (int i = 0; i < N; i += 1)
    {
      csub[m][n] += a[row + m][i] * b[i][col + n];
    }
  }
}
for (int m = 0; m < cm; ++m)
{
  for (int n = 0; n < cn; ++n)
  {
    c[row + m][col + n] += csub[m][n];
  }
}
```

Use A & B data in the case; B load is coalesced

GEMM: loop exchange

```
for (int m = 0; m < cm; m++) {
    for (int n = 0; j < cn; n++) {
        for (int i = 0; i < K; i++) {
            csub[m][n] += a[row + m][i] * b[i][col + n];
        }
    }
}
```



There are so many other optimizations (share local memory, K-slice, multiple tile cache locality, avoid bank conflict) for GEMM, to achieve peak perf is hard

Optimized Algorithm: Winograd GEMM

$$\begin{bmatrix} i0 & i1 & i2 \\ i1 & i2 & i3 \end{bmatrix} \times \begin{bmatrix} F0 \\ F1 \\ F2 \end{bmatrix} = \begin{bmatrix} Y0 \\ Y1 \end{bmatrix}$$

$$Y0 = i0 * F0 + i1 * F1 + i2 * F2$$

$$Y1 = i1 * F1 + i2 * F2 + i3 * F3$$

$\Rightarrow$

$$Y0 = X0 + X1 + X2$$

$$Y1 = X1 - X2 - X3$$

$$X0 = (i0 - i2) * F0$$

$$X1 = (i1 + i2) * (F0 + F1 + F2)/2$$

$$X2 = (i1 - i3) * F2$$

$$X3 = (i2 - i1) * (F0 - F1 + F2) / 2$$

Compute Operations decrease, memory accesses increase

Optimized Algorithm: one pass Variance

$$\bar{x} = \frac{\sum_{j=1}^{n} x_j}{n}, \quad \text{sample variance} = s^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1},$$

Var(X) = E((X - Mean(X))^2)

Naïve algorithm, x is read in two pass: one pass for mean, one pass to get the sum of the squares of the differences from the mean

Var(X) = E(X^2) - (E(X))^2

Read X once, calculate the square and sum together

There might be numerical instability issue, while thanks to deep learning tolerance, it's OK in reality

Graph Level Optimization

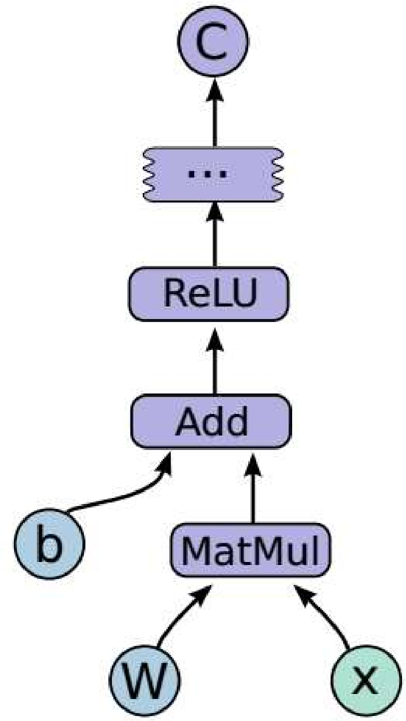
DL Computation Graph

A way to represent a math function in the language of graph theory.

- Every neural network represents a single mathematical function
- These functions are often very complex
- Graph transformation = Optimization

TensorFlow

- Graph **nodes** represent operations "**Ops**" (**Add**, **MatMul**, **Conv2D,** …)
- Graph **edges** represent "data" flowing between ops




```
relu = tf.nn.relu(tf.matmul(w, x) + b)
```

Loop Fusion: GELU

GAUSSI AN ERROR LINEAR UNIT

$$\mathrm{GELU}(x) = xP(X \le x) = x\Phi(x)$$
$$\approx 0.5x \left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

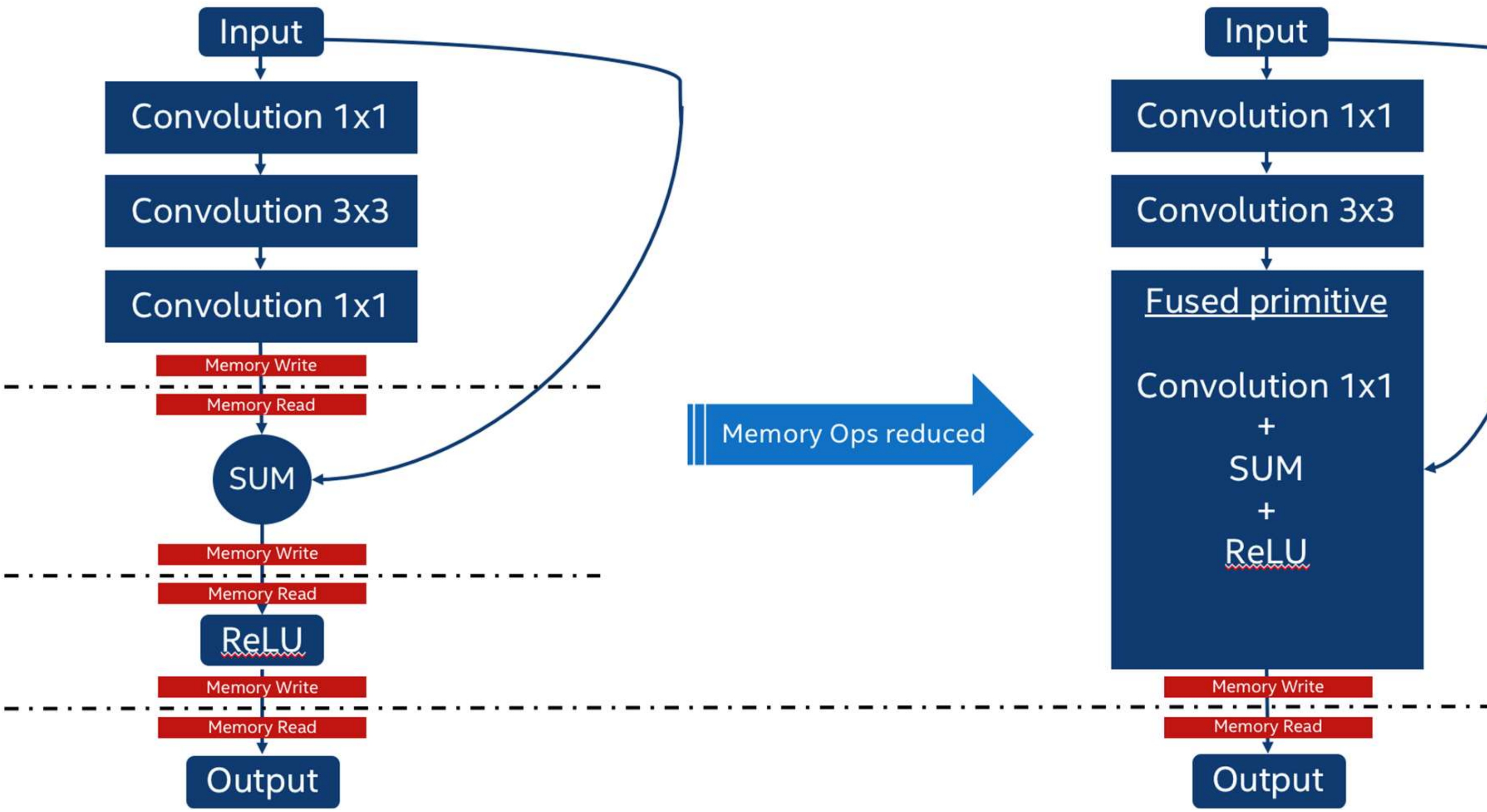
There are 7 ops in the computation graph, too much memory read and write

```
// DPC++/SYCL
parallel_for(range{n}, [=](id<1> i) {
  result[i] = data[i] * data[i] * data[i];
});

parallel_for(range{n}, [=](id<1> i) {
  result[i] = 0.044715 * data[i];
});
……
```

```
// DPC++/SYCL
parallel_for(range{n}, [=](id<1> i) {
  result[i] = (data[i] * data[i] * data[i]) * 0.44715 + data[i])……
});
```

All intermediate are in registers

Fuse Matrix Multiply and Activation



DL Compilation Technology

DL Complier: from computation graph to optimized code for different HW backend

- MLIR
- OpenXLA
- Triton
- TVM

Overview: MLIR & MLIR based Compiler

- MLIR: DL compiler infrastructure, which provides reusable and extensible compiler components. Support developers to write end-to-end compiler
- End-to-end (domain specific) compiler: take framework graph as input, compiled to independent executable with optimizations
  - XLA: starting from TensorFlow using its own IR (XLA HLO). Gradually moving to MLIR based
  - IREE: MLIR based, including compiler and runtime (still under development, especially on training side)
  - Others: BladeDisc (Alibaba), OneFlow, ByteIR (ByteDance) …

MLIR Core: programming language

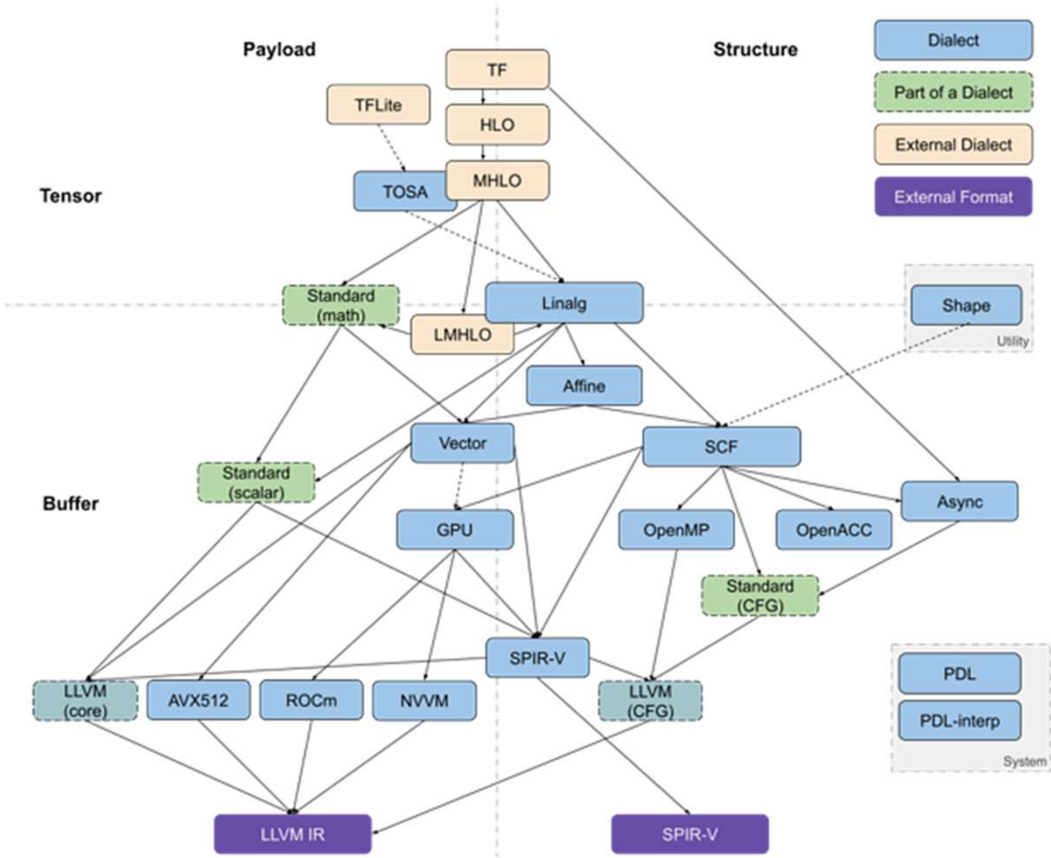
MLIR in-tree dialect: standard library

E2E Compiler: applications

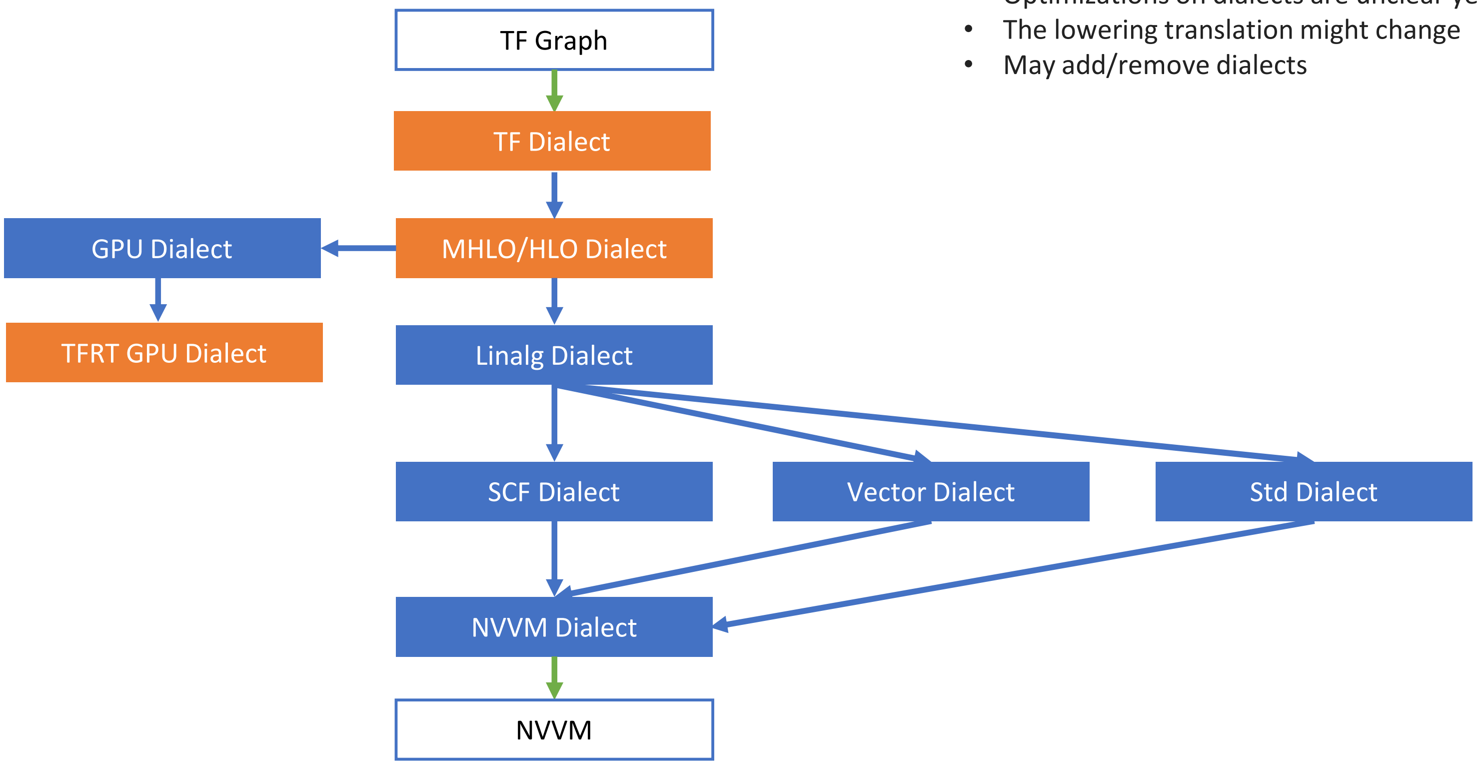
MLIR Ecosystem

It provides
- Specification & infrastructure to build dialects & transformations
- A set of dialects
- Certain conversions: transformation between and inside dialects

Out of scope
- Translation: dialects to/from external formats
- Runtime
- All dialects
- HW specific optimizations
- Full codegen capability
- Compile & linkage



MLIR based XLA



Not finalize yet

- Optimizations on dialects are unclear yet
- The lowering translation might change
- May add/remove dialects

The Trend of DL Computation

- HW adds more powerful instruction to improve throughput (CPU, GPU, accelerators)
  - VNNI (dot product)
  - AMX/Tensor Core (small matrix mul)
  - TPU, Cambricon, Habana…… (bigger matrix mul)
- Sparse linear algebra, sparse algorithm
- Low latency, high bandwidth: bigger SRAM, high bandwidth memory
- Non uniform memory architecture is more common
- Low precision: INT8, INT4, INT2 on inference, BF16/FP8 on training

SW optimization are even more critial

Thank You