面向AI的计算优化

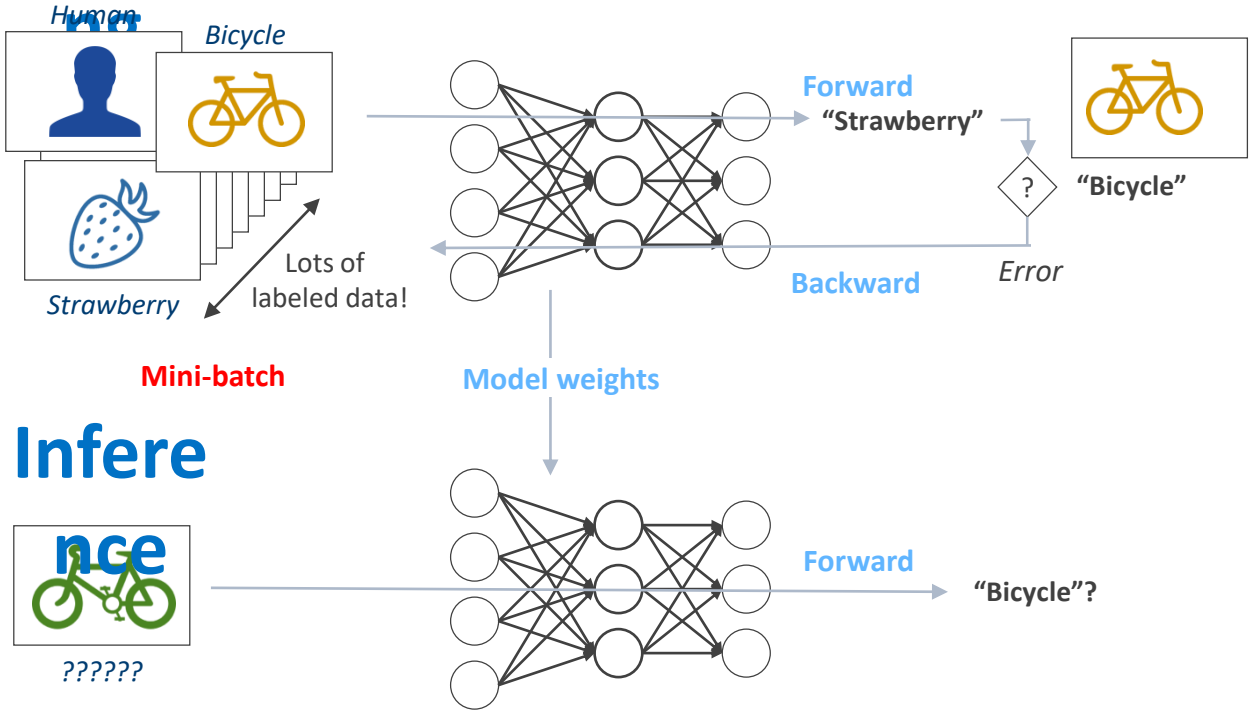
林晓东

英特尔公司

eric.lin@intel.com

AI and DL Basis

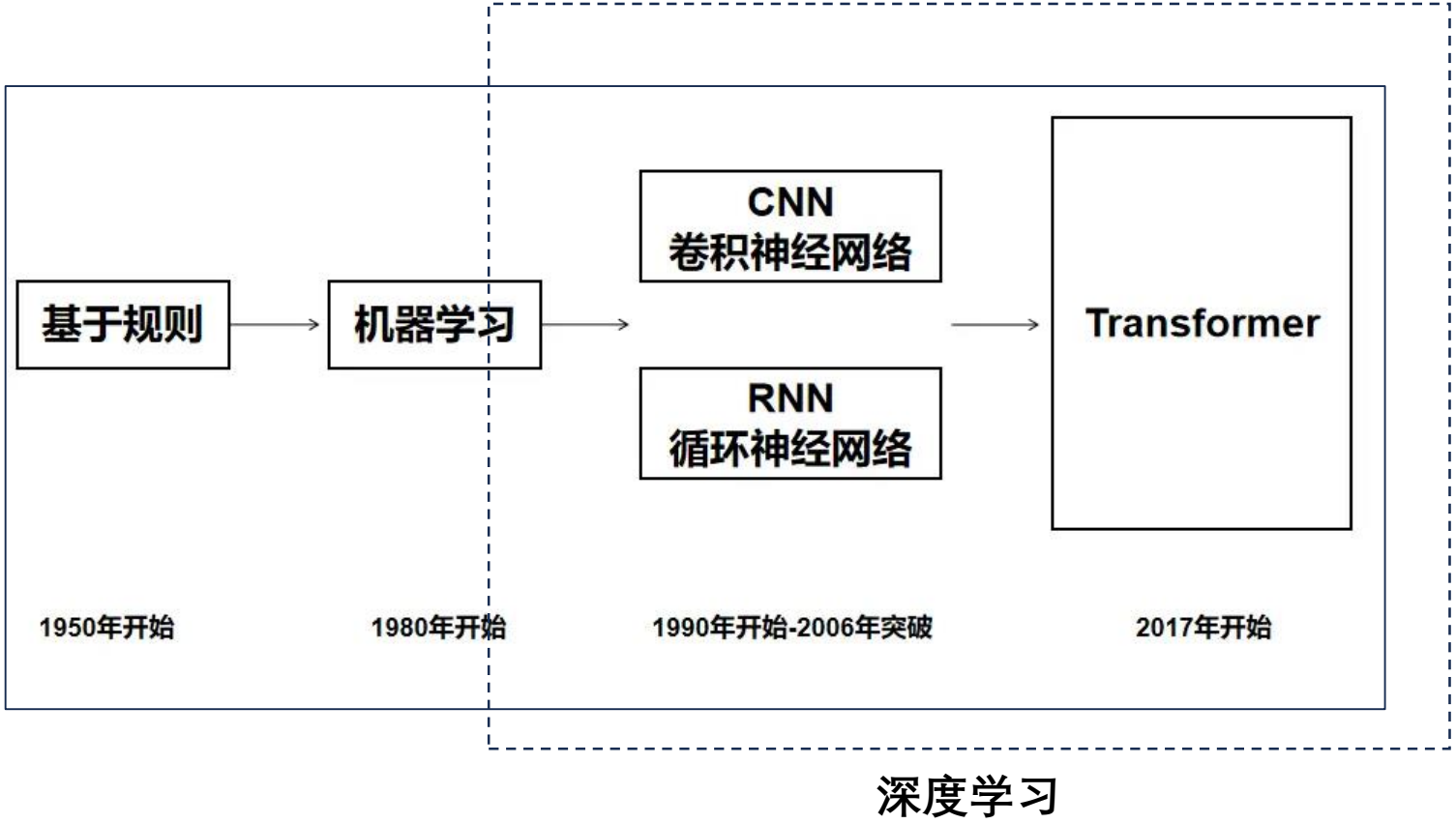
Traini



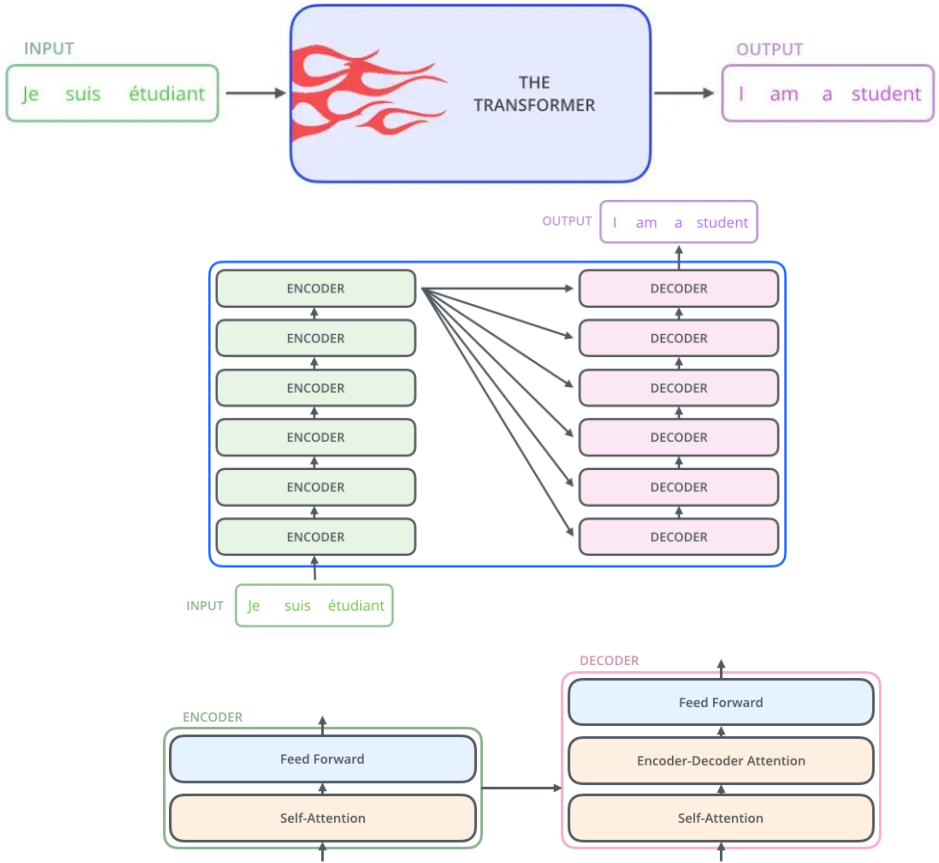
Infere

nce

AI: Architecture Evolvement

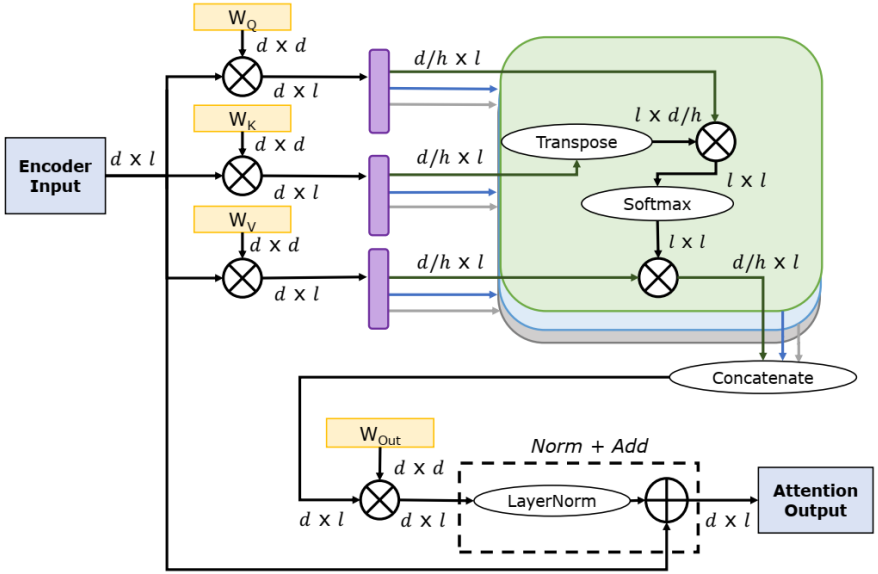


Transformer Architecture

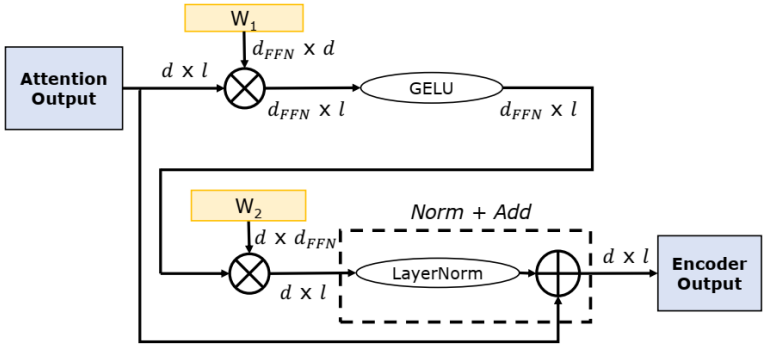


Picture: http://jalammar.github.io/illustrated-transformer/

Encoder Only: Mostly Used In LLM



Muti-Head Attention (MHA) Module



Feed-Forward Network (FFN) Module

Ops: Most Essential Building Block

```
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

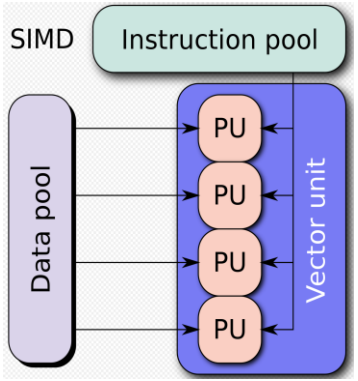
mul & add: GEMM, conv, RNNCell

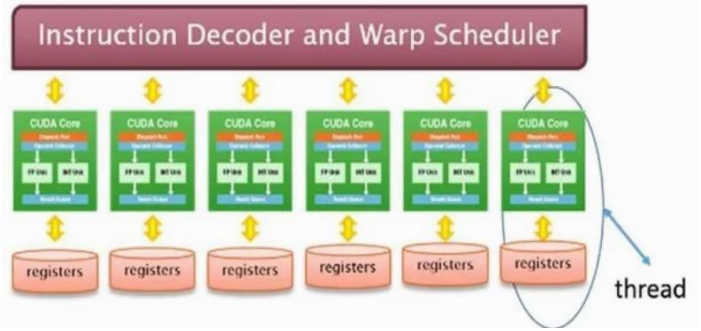
memory: embedding, softmax, transpose, concat, normalization, element-wise, dropout, transpose

```
void gemm(int M, int N, int K, float* A, float* B, float *C)
{
   unsigned int m, n, k;
   for (m = 0; m < M; m++) {
      for (n = 0; n < N; n++) {
         C[m][n] = 0.0;
         for (k = 0; k < K; k++) {
            C[m][n] += A[m][k] * B[k][n];
         }
      }
   }
}
```

Op/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

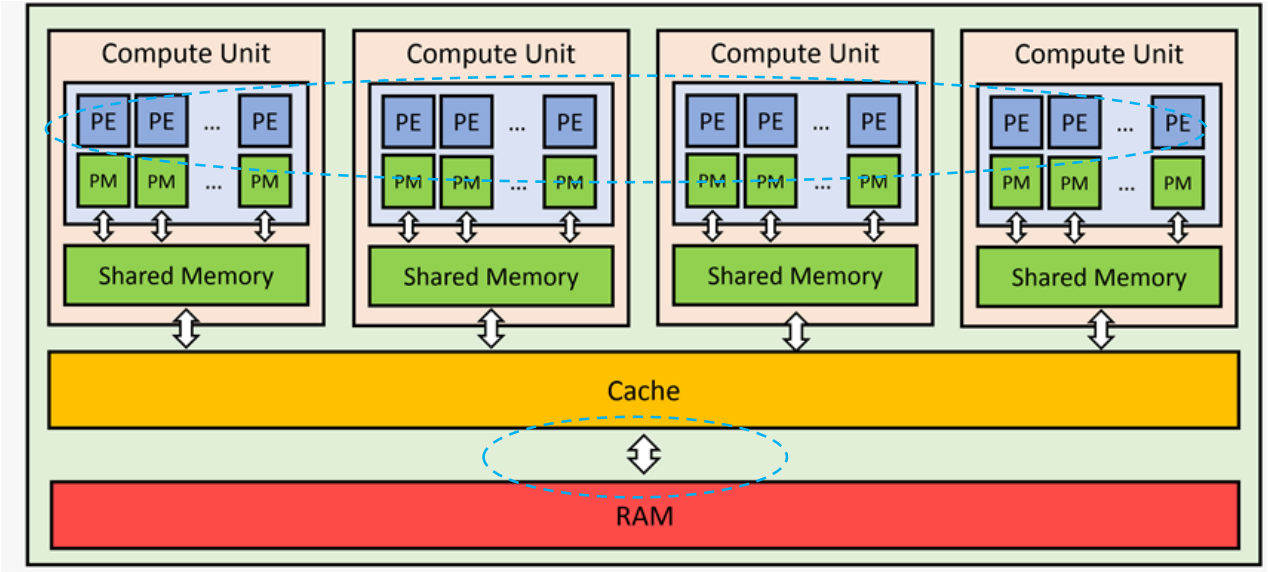
- Programming Language express the parallelism: OpenMP, SYCL/DPC++, OpenCL, CUDA …
- There are no essential difference between SIMD & SIMT on high performance code

Optimization Goal

Achieve HW peaks

- Occupancy
- 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];
}
```

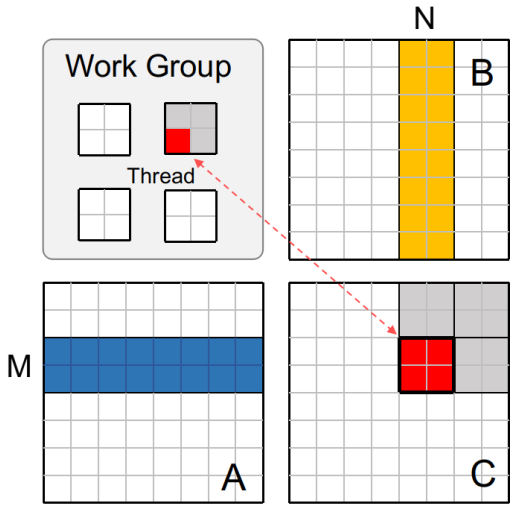
```
// DPC++/SYCL
parallel_for(range{n}, [=](id<1> i) {
   if (data[i] <= 0.0)
      result[i] = 0.0;
   else
      result[i] = data[i];
});
```

```
// 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

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];
```

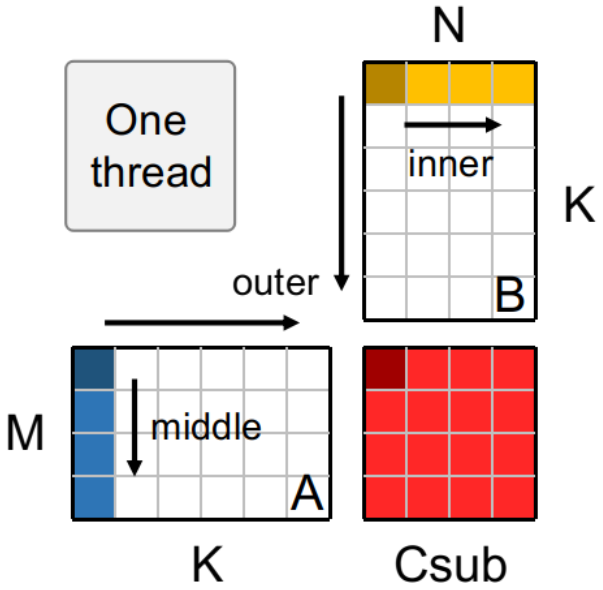



```
size_t row = index[0];
size_t col = index[1];
float csub[cm][cn] = {0.0f};
for (int m = 0; m < cm; ++m)
{
  for (int n = 0; n < cn; ++n)
  {
    for (int i = 0; i < N; i += 1)
    {
      csub[m][n] += a[row + m][i] * b[i][col + n];
    }
  }
}
for (int m = 0; m < cm; ++m)
{
  for (int n = 0; n 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



Inner product to outer product => reuse data in A

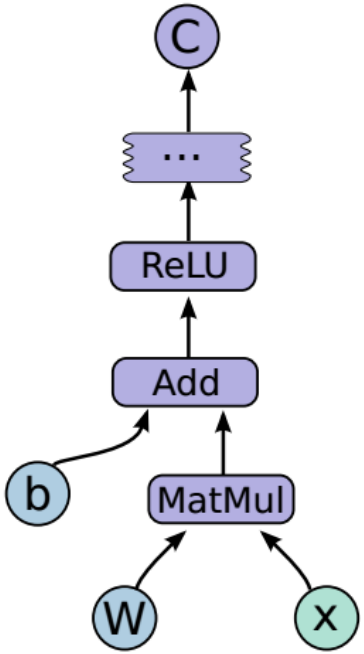
There are so many other optimizations (tiling, share local memory, K-slice, multiple tile cache locality) for GEMM, to achieve peak perf is hard. Shapes are critical!

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



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

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

Why Graph Optimization

- Ops fusion: reduce memory pressure
- Constant propagation: normalization scale become part of weight
- Layout propagation: cache friendly load/store; remove unnecessary transpose/permute
- Remove overhead caused by synchronization
- Common Subexpression Elimination
- …

Fusion

Essentially two steps

- Decide what to fuse: manual, pattern matching, automatic
- Generate code for fusion: static programming language (SYCL, CUDA), JIT language (Triton), LLVM (Legacy XLA), MLIR (OpenXLA)

Loop Fusion: GELU

GAUSSI AN ERROR LINEAR UNIT

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

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

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

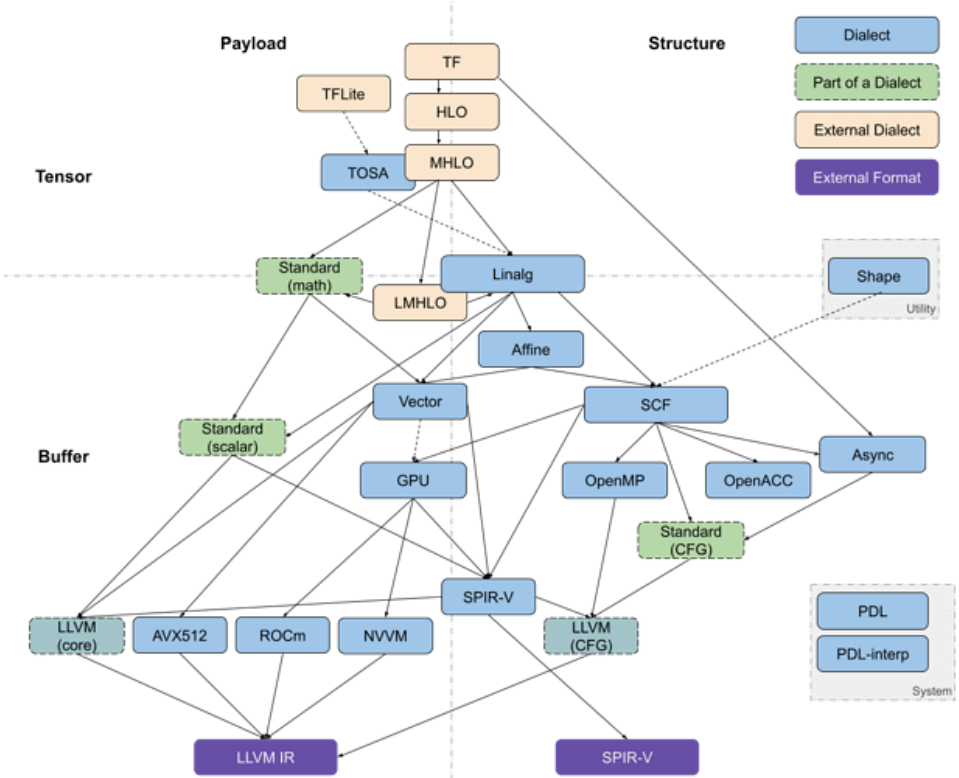
MLIR Ecosystem

It provides

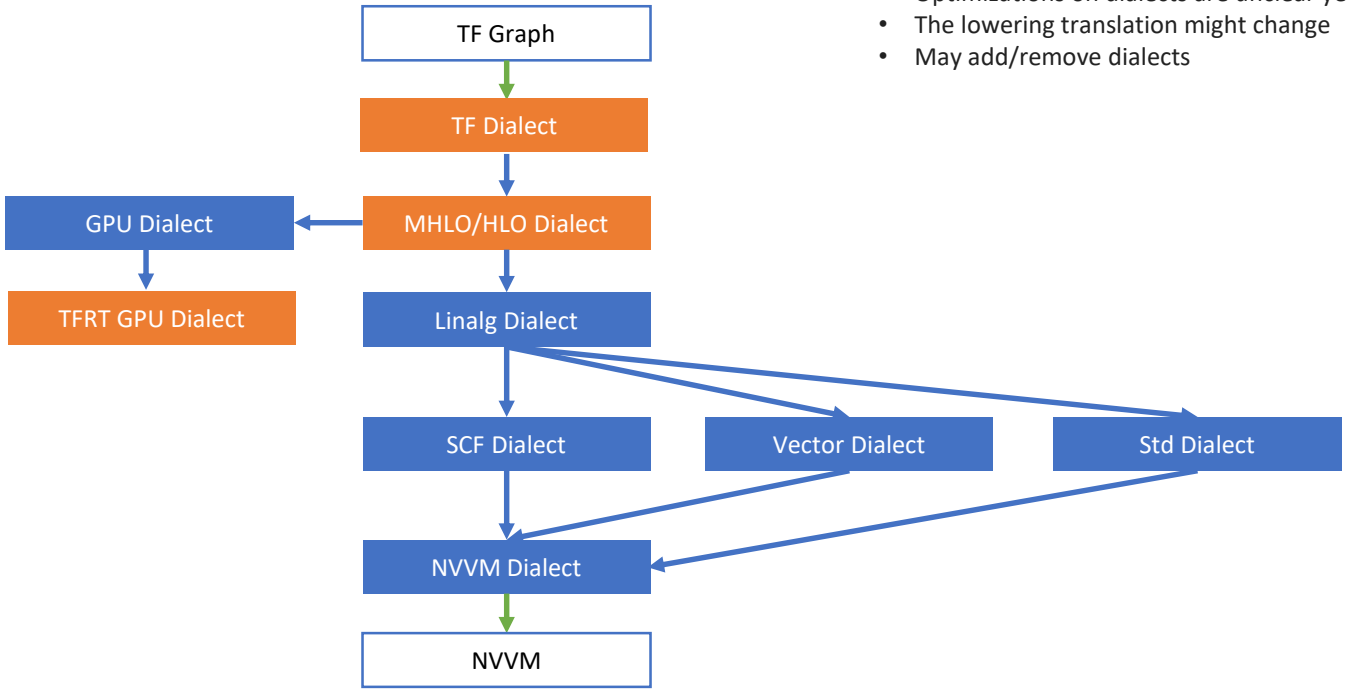
- Specification & infrastructure to build dialects & transformations
- A set of dialects
- Certain conversions: transformation between and inside dialects

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

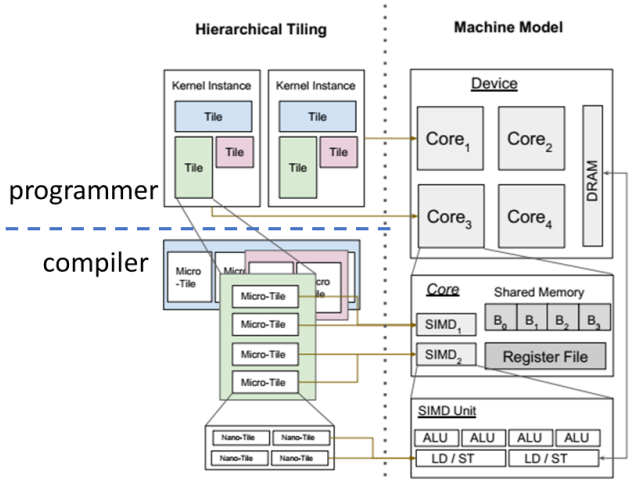
Triton Lanugage

```
@triton.jit
def add_kernel(x_ptr, y_ptr, output_ptr, n_elements, BLOCK_SIZE):
    pid = tl.program_id(axis=0)

    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    mask = offsets < n_elements

    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)
```

decorator for JIT compiler

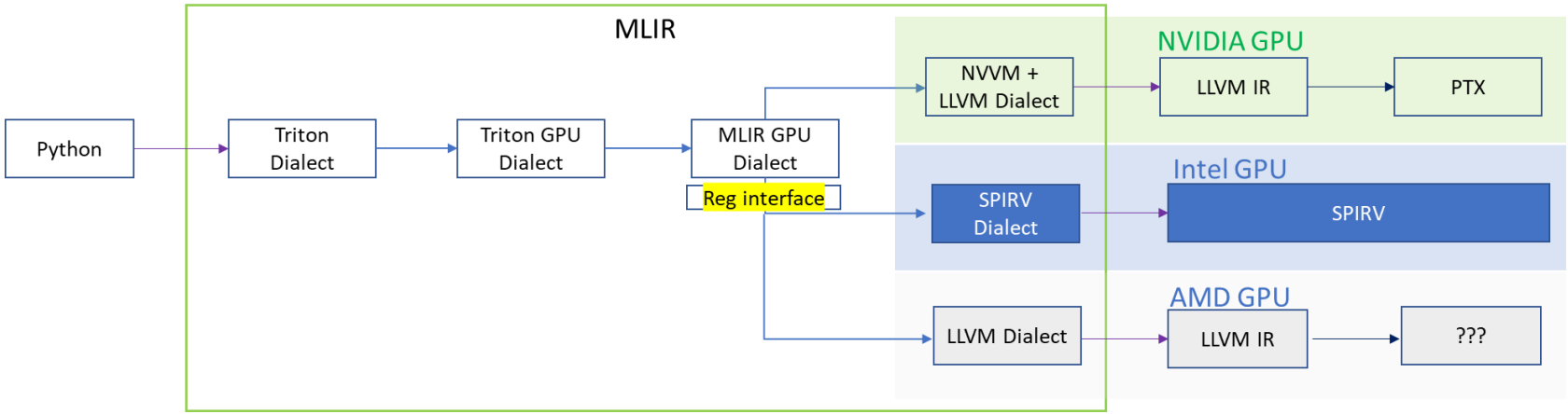
construct iteration space of a block, size must be power of 2

load blocks and compute on the block



- SPMD, block-wise programming model: programmers manipulate blocks/tiles, compiler takes care of others
- A set of built-in language APIs like memory, math, dot, reduction …
- No built-in runtime

Triton: MLIR Based Implementation



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 critical

谢谢大家！