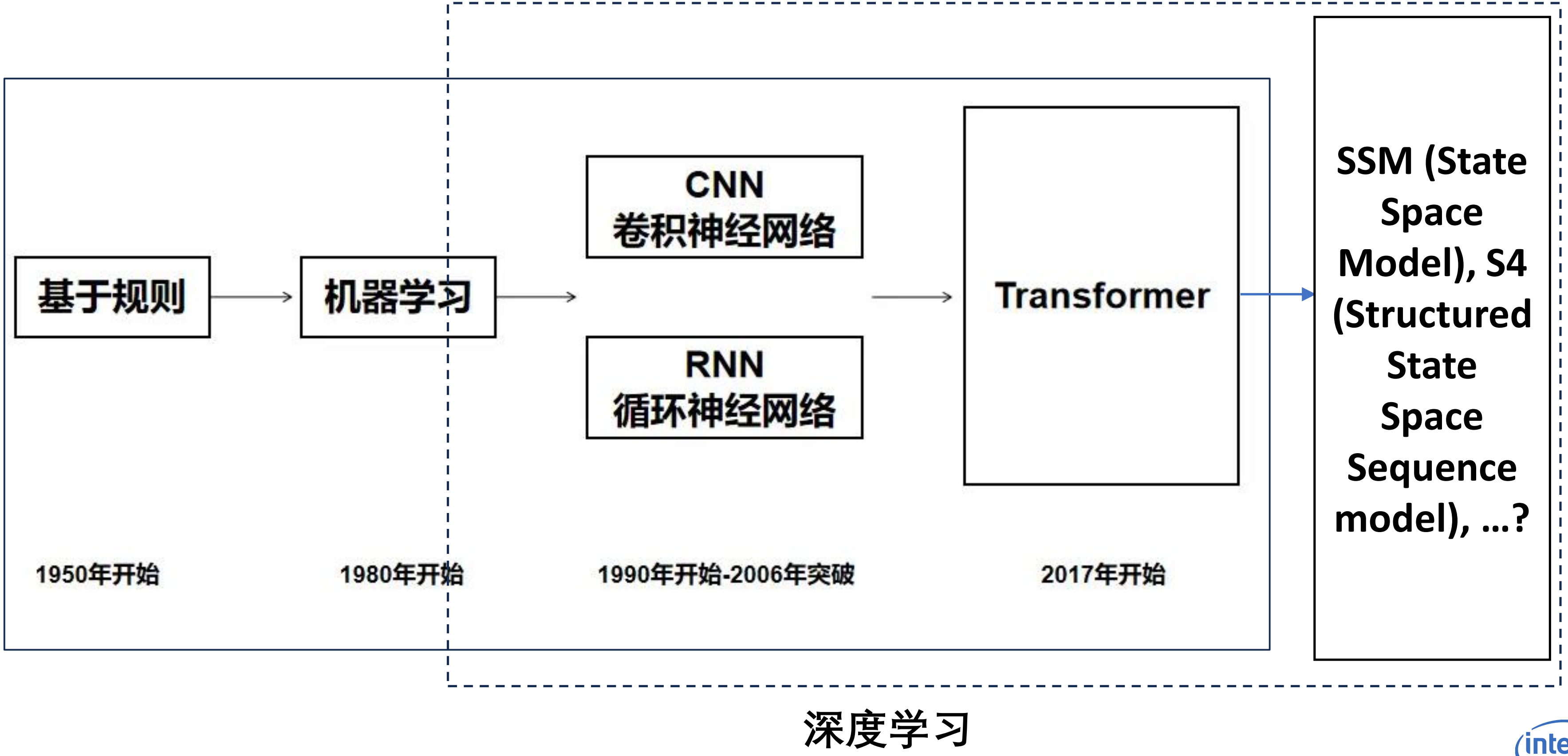
# 面向AI的计算优化

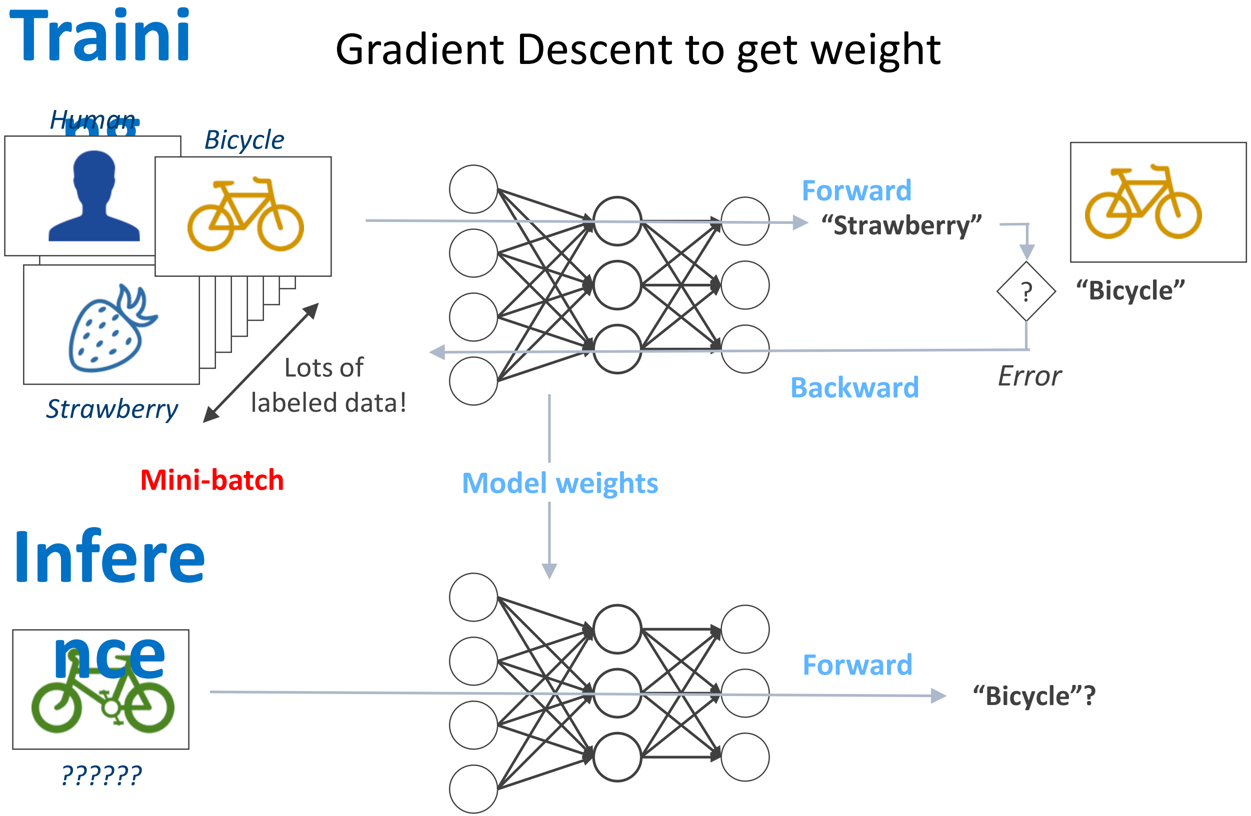
林晓东
英特尔公司
eric.lin@intel.com

# AI: Architecture Evolvement

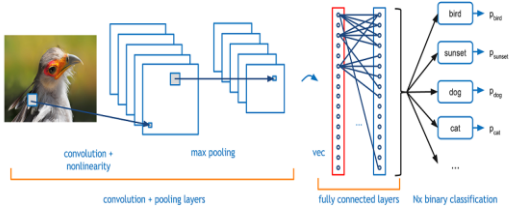


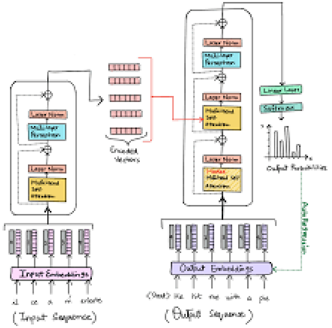
# DL Basis

Gradient Descent to get weight



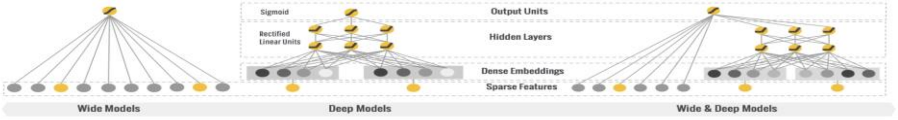
# Deep Learning:  What Data Scientists See



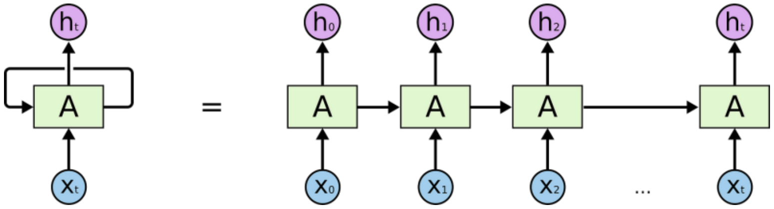



Convolutional Neural Network (CNN)

Transformer




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




Recommendation Systems

RNN

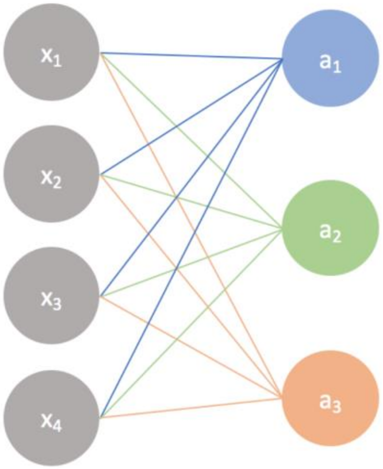
# Network Architecture

All Architectures are essentially "optimization" from fully connected NN

- CNN
  - Vision & image
- RNN
  - Temporal sequence (text, speech …)
  - Attention was introduced
- Transformers
  - "Attention is all your need": global & accurate "understanding"
  - Computationally efficient and more scalable Easy for parallel computing
  - Claim "can be purposed for any task"
  - Compute hungry

Some models might have multiple building blocks

# Simplest Neural Network




$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow[activation]{} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

A matrix vector multiplication with bias

If there are multiple batches, then a matrix multiplication plus bias add (fit GEMM)

# DL Compute Building Block: Op/Kernel

```
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])
        }
}
```

Compute bound: GEMM, conv, RNNCell

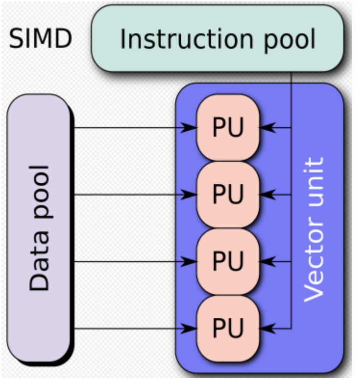
Memory bound: embedding, softmax, transpose, concat, normalization, activation, element-wise, dropout, transpose

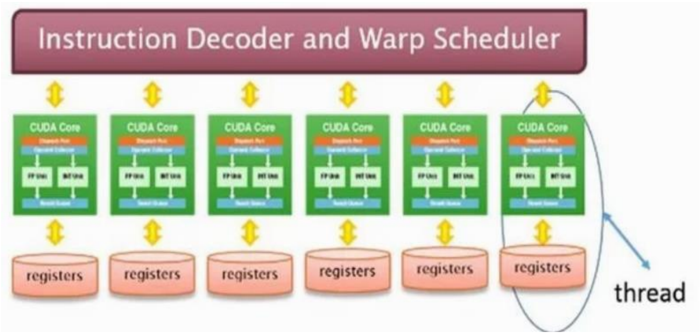
DL Ops are just normal codes, hungrier for TFLOPS & memory bandwidth

```
void matmul(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/Kernel Level Optimization

# Optimization Basis




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

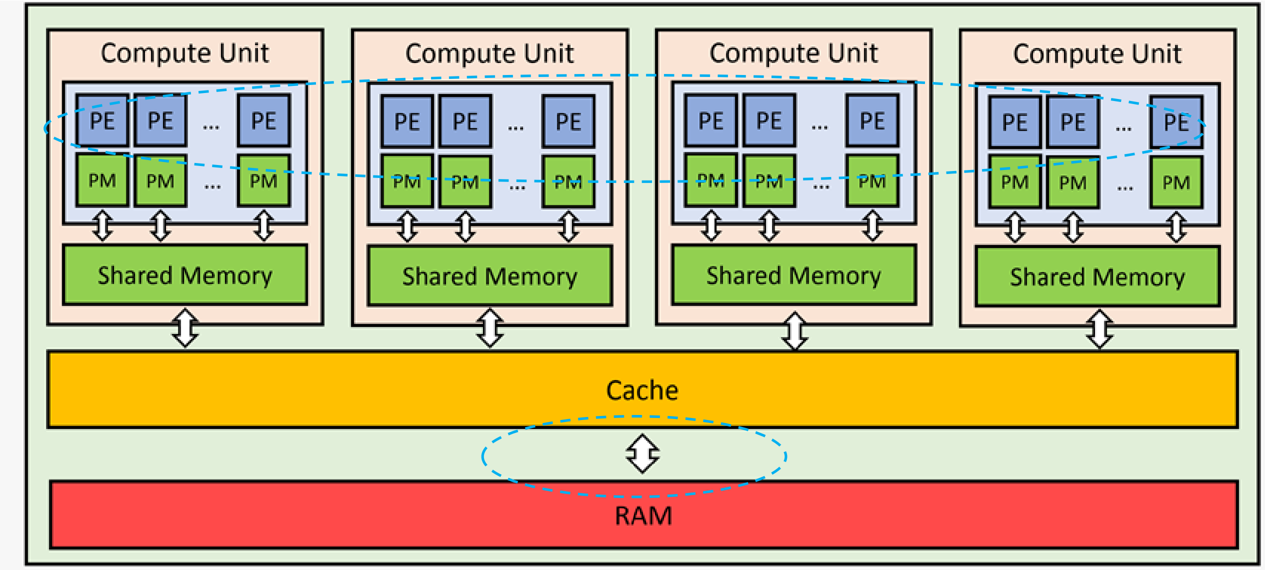
- Programming Language express the parallelism: OpenMP, SYCL/DPC++, OpenCL, CUDA …
- There are no essential difference between SIMD & SIMT on high performance code though SIMT is usually easy to program

# 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];
}
```

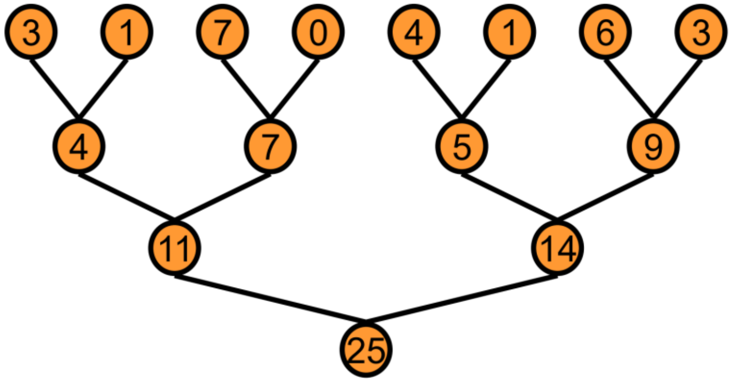
```
// DPC++/SYCL
parallel_for(range{n}, [=](id<1> i) {
    if (data[i] <= 0.0)
        result[i] = 0.0;
    else
        result[i] = data[i];
});
```

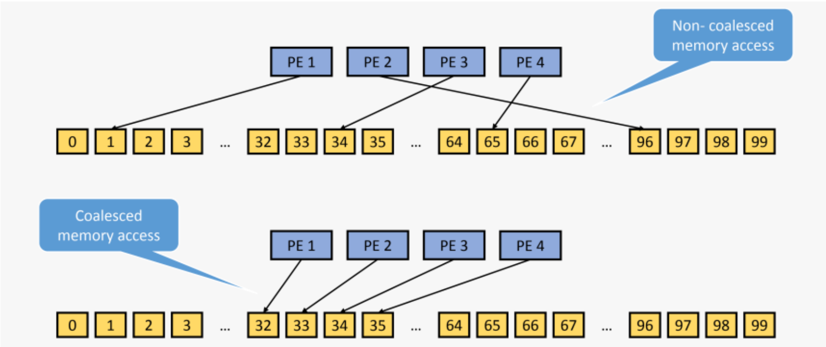
```
// CUDA
__global__ void ReLU(float *data, float *result)
{
    int i = threadIdx.x;
    if (data[i] <= 0.0)
        result[i] = 0.0;
    else
        result[i] = data[i];
}

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

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

# Make it Parallel: Reduction

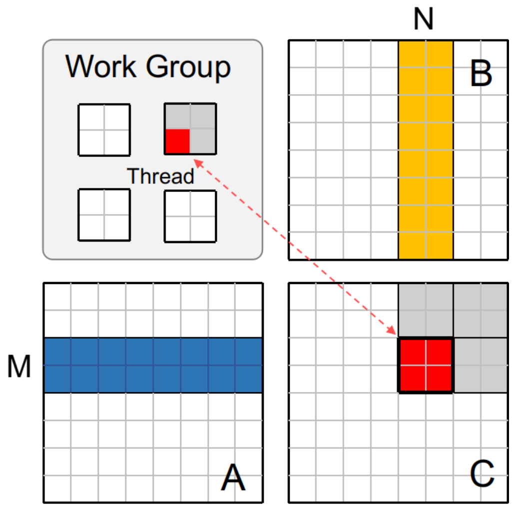
- Processing large dataset with associative and commutative operations (sum, product, max/min……): normalization, softmax are all reduction based
  - Partition the data set into smaller chunks
  - Each work-item/thread to process a chunk
  - Reduction tree to summarize the results from each chunk into 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



# GEMM: Simple Tiling

```
// 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];
```



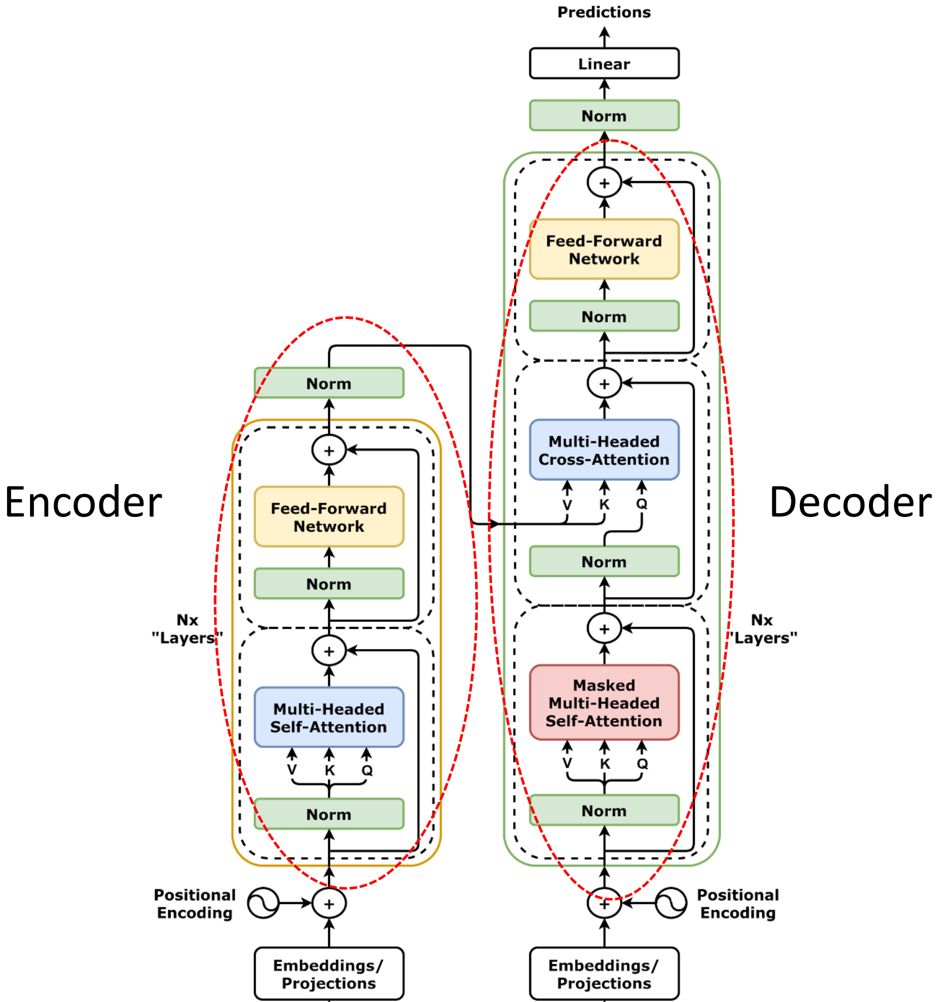

```
size_t row = index[0];
size_t col = index[1];
float csub[cm][cn] = {0.0f};
for (int m = 0; m < cm; ++m)
{
  for (int n = 0; n < cn; ++n)
  {
    for (int i = 0; i < N; i += 1)
    {
      csub[m][n] += a[row + m][i] * b[i][col + n];
    }
  }
}
for (int m = 0; m < cm; ++m)
{
  for (int n = 0; n 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

# GEMM is most important OP in DL

While compute optimization is not easy

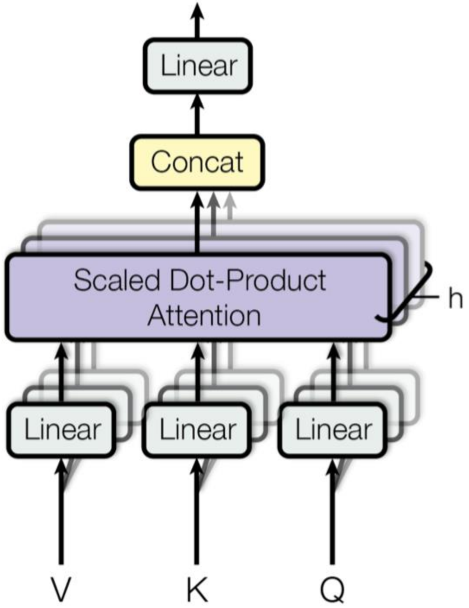
- GEMM shape: M, K, N
- SW & HW data layout, transpose
- HW: number of PE, FLOPS, memory bandwidth, cache, SLM, HW matmul capability, sync/async with scalar/vector unit, synchronization mechanism
- Algorithm: naïve C tiling, K-slicing, stream K, software pipelining…

# Transformer: The LLM Building Block



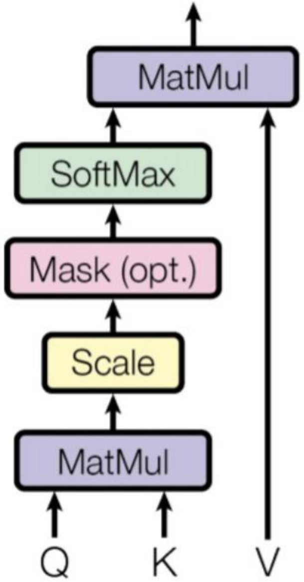

- Post & Pre LayerNorm is all possible
- Decoder only architecture is used to build most LLMs
- Multiple decoder layers
- Major ops
  - embedding
  - add
  - layer normalization
  - FFN is mainly GEMM & activation
  - Linear is GEMM

# Multi Head Attention




$$\mathrm{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\mathrm{head}_1, \ldots, \mathrm{head}_h]\mathbf{W}_0$$

$$\text{where } \mathrm{head}_i = \mathrm{Attention}\left(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V\right)$$




$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Ops

- Linear: GEMM
- (batch) Matmul: (batch) GEMM
- Scale: mul
- Softmax

# GEMM Shape in Transformers*

## Linear (M, K, N)

- Q, K, V projection: b * s, h, h
- Attention output (post attention projection): b * s, h, h
- FFN/MLP: b * s, h, h_ffn; b * s, h_ffn, h (quite a few models, h_ffn == 4h)

## (Batch) MM: [batch, M, K], [batch, K, N]

- Q X K^T = Score : [b * n_heads, s, head_dim], [b * n_heads, head_dim, s]
- Score X V: [b * n_head, s, s], [b * n_head, s, head_dim]

*b: batch size, number of sequence*
*s: sequence length (number of tokens)*
*h, h_ffn: hidden size, hidden size of FFN*
*n_heads: number of heads*
*head_dim: dimension of head, query size/key size/value size*

*assume n_kv_heads == n_heads, aka, no grouping

# FLOPS & Memory in Transformer Based Model

## Memory

- Weight per Transformer Block
  - Wq, Wk, Wv, Wo: 4 * h * h
  - Wffn: 2 * h * h_ffn
- Embedding weight: vocab_size * h
- Weight lm_head: h * vocab_size
- Training specific : Gradient (1X weight) & Optimizer State (2X weight); Activations
- Inference specific: KV cache (2 * b * s * h)

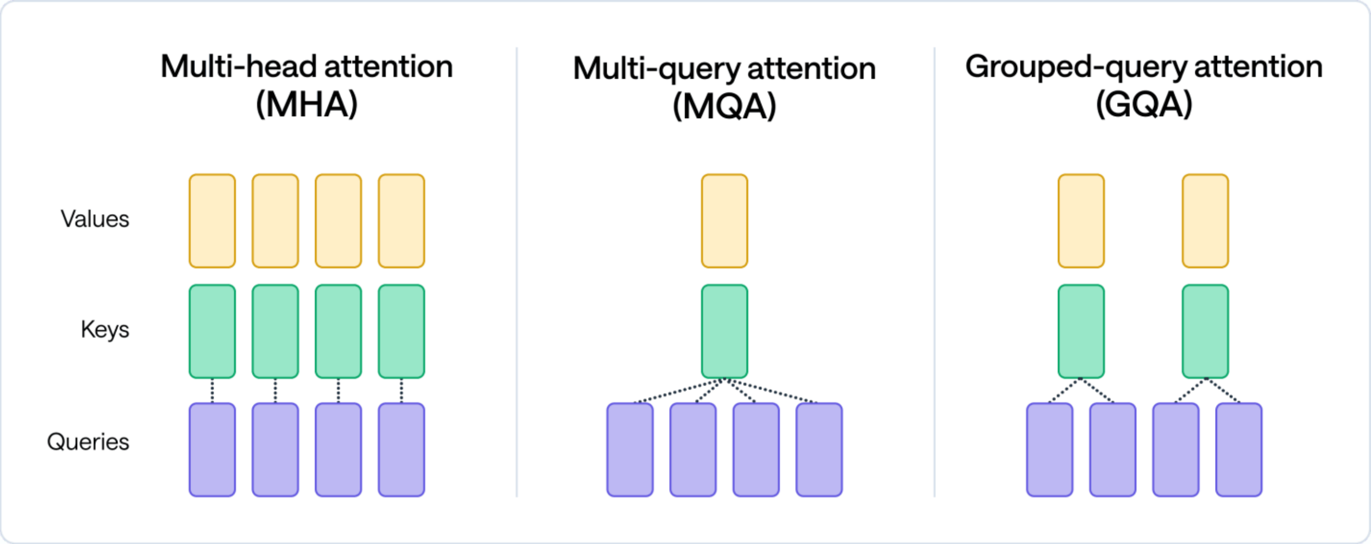
## FLOPS Forward

- Per Transformer Block
  - Q, K, V, Output: 8 * b * s * h * h
  - Attention: 4 * b * h * s * s
  - FFN: 4 * b * s * h_ffn * h
- lm_head: 2 * b * s * h * vocab_size

* Backward FLOPS is 2 – 3X of Forward

That’s why LLM is hungry for memory & compute

# Attention Variants: MHA/MQA/GQA




- Less compute
- Less memory, both capacity & bandwidth

How many Query heads share one Key & Value heads

# Example: Llama-3 70B Config

```
{
  "_name_or_path": "meta-llama/Meta-Llama-3-70B-Instruct",
  "architectures": [
    "LlamaForCausalLM"
  ],
  "hidden_size": 8192,              # h
  "intermediate_size": 28672,       # h_ffn
  "model_type": "llama",
  "num_attention_heads": 64,        # n_heads, h / head_dim
  "num_hidden_layers": 80,          # number of transformer layer
  "num_key_value_heads": 8,         # number of KV heads, number of groups
  "vocab_size": 128262
}
```

# (Scaled Dot Product) Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Safe or Stable softmax: exp(x – max(x)) to avoid overflow

1. Q X K^T, BMM with M, K, N: s, head_dim, s; B (b * n_heads)
2. max(x)
3. sum (exp(x-max(x))
4. divide
5. Score X V, BMM with M, K, N: s, s, head_dim; B (b * n_heads)

Attention computational & space complexity: O(h * S^2)

Optimization: how to reduce memory access/how to tile softmax

- Naturally fuse all ops into one
- But we need all x to get max(x) and sum(exp(x-max(x))

# FlashAttention

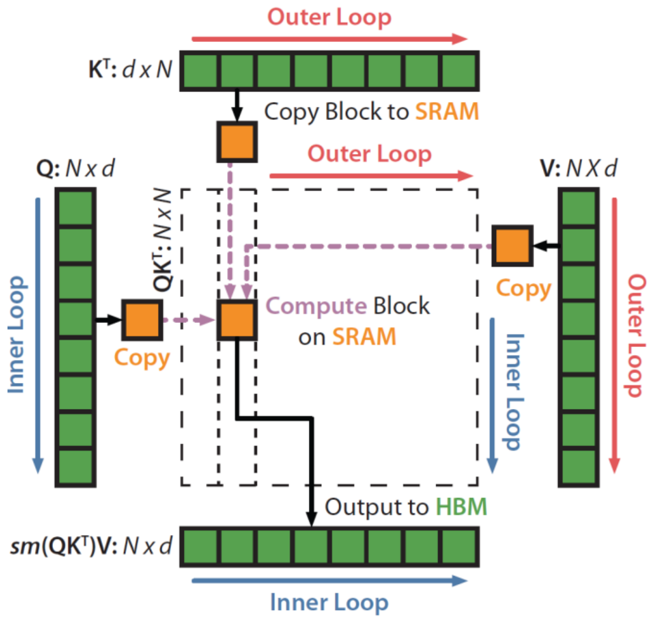
- Compute one tile of Q X K^T
- Find **local** max & **local** sum, calculate **local** softmax of every tile
- Compute one tile of “score X V” (partial output)

After completing all tile

- Get global max & global sum from all local max & local softmax, remediate all the partial output

Algorithm based on

- exp(x - m) = exp(x - t) × exp(t – m)
- matmul is linear transformation



# LLM Inference Optimization: KV Cache

- KV Cache: optimization for decoder only models => auto-regressive or causal (attention of a token only depends on its preceding tokens)
- Conceptually for every iteration the input increases by 1
  - Q, K, V of previous tokens are same => no need to recompute so caching Key & Value vectors
  - The output representation for particular tokens will be also identical for all subsequent iterations => no need to compute their attention => no Q cache needed
  - Example: iteration 2, only compute attention of “My” with K vector and V vector of “what” “is” “your name” “?” “My”

**Inference Iteration**

W/o KV Cache:
Iteration 1: what is your name?
Iteration 2: what is your name? My
Iteration 3: what is your name? My name

W/ KV Cache:
Iteration 1: what is your name?
Iteration 2: My
Iteration 3: name

W/ KV cache: iteration to produce first token is usually called *prefill*; iteration to produce subsequent tokens is called *decoding/generation* where attention complexity is O(h * s)

# LLM Inference Optimization For Deployment

- Serving framework: vLLM, TGI
- KV cache memory increases due to longer & longer context: PagedAttention to manage KV cache efficiently borrowing the idea of page table for memory management
- To utilize FLOPS for decoding (s == 1) => larger batch size while sequence length in one batch varies: continuous batching/dynamic batching
- Compuet prefill & decoding together => chunked prefill

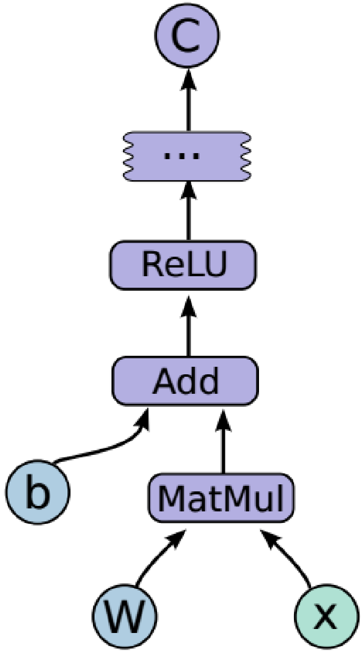
# Graph Level Optimization

# 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

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

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

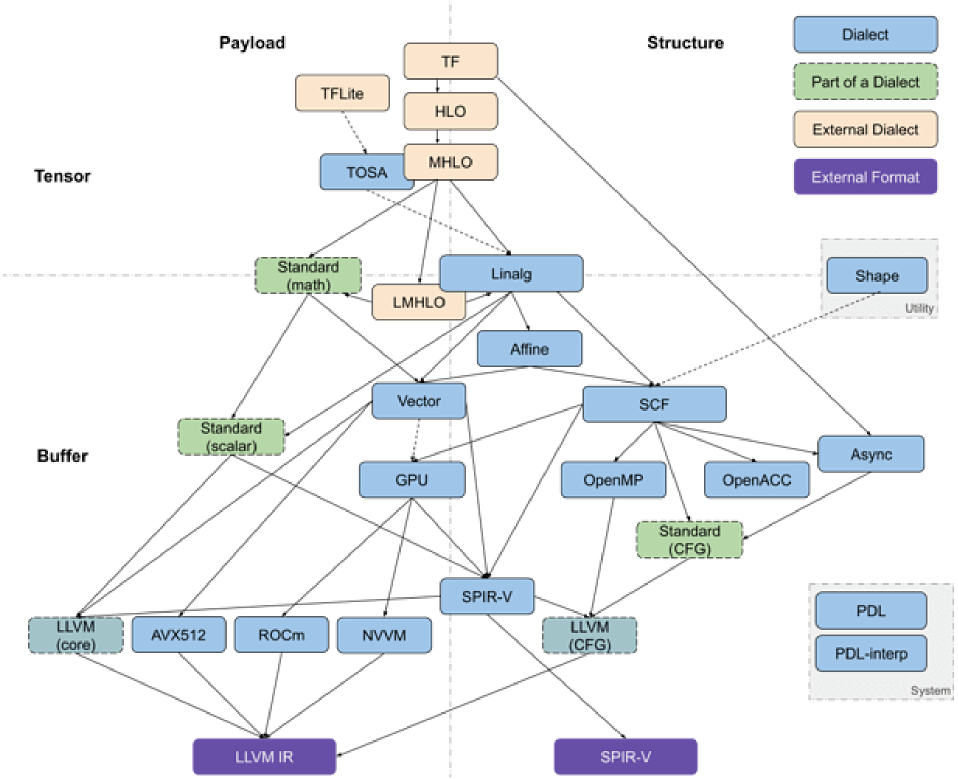
MLIR in-tree dialect: standard library

E2E Compiler: applications

# MLIR Ecosystem

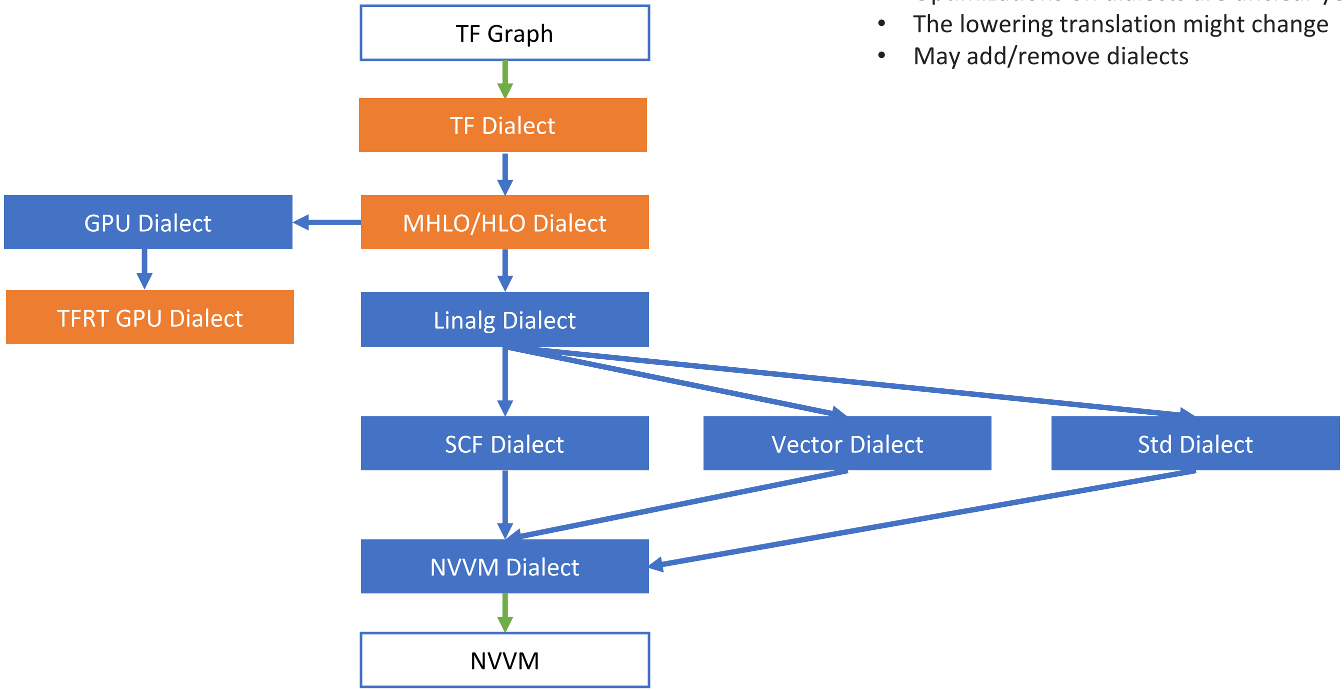
It provides
- Specification & infrastructure to build dialects & transformations
- A set of dialects
- Certain conversions: transformation between and inside dialects

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




intel

# MLIR based XLA




Not finalize yet

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

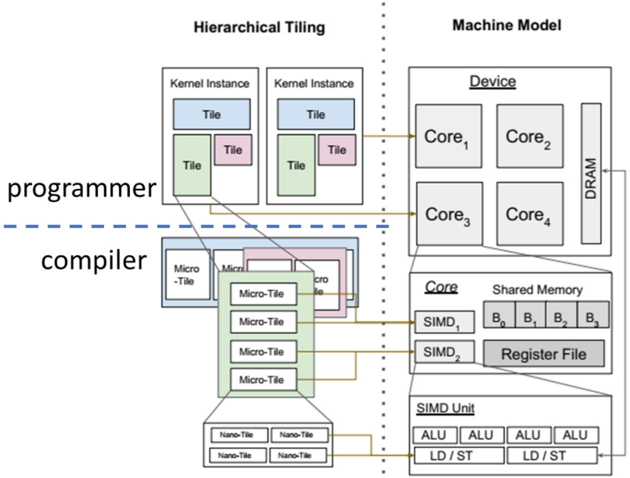
# Triton Lanugage

decorator for JIT compiler

```
@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)
```

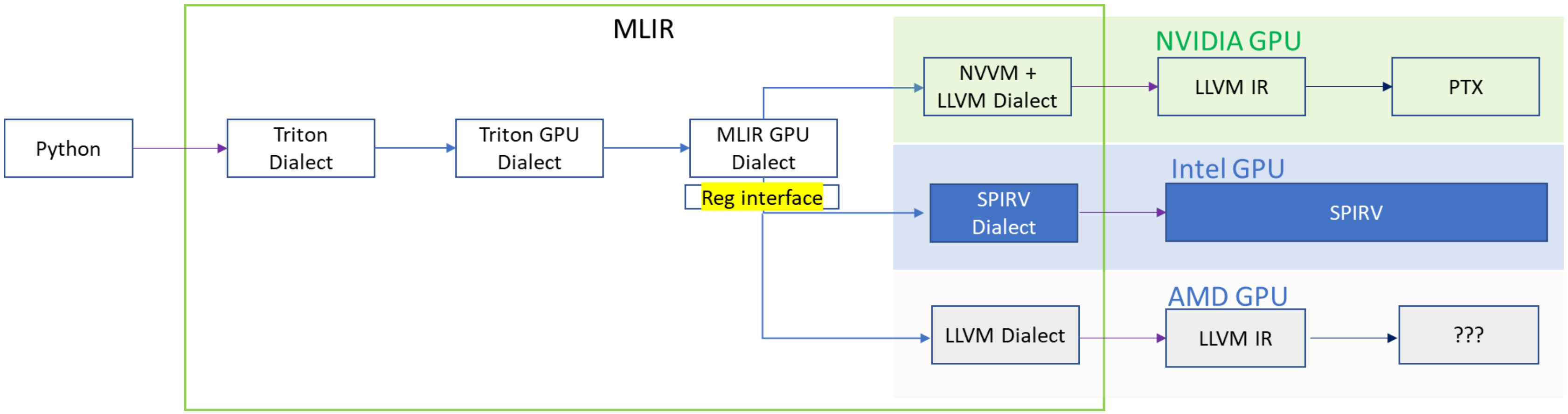
construct iteration space of a block, size must be power of 2

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)
- More memory hierarchy: SLM, DSLM
- Memory capacity
- Non uniform memory architecture: high bandwidth connection, parallel
- Low precision: INT8, INT4, BF16, FP8, MXFP4, MXFP6 ...

SW optimization are even more critical