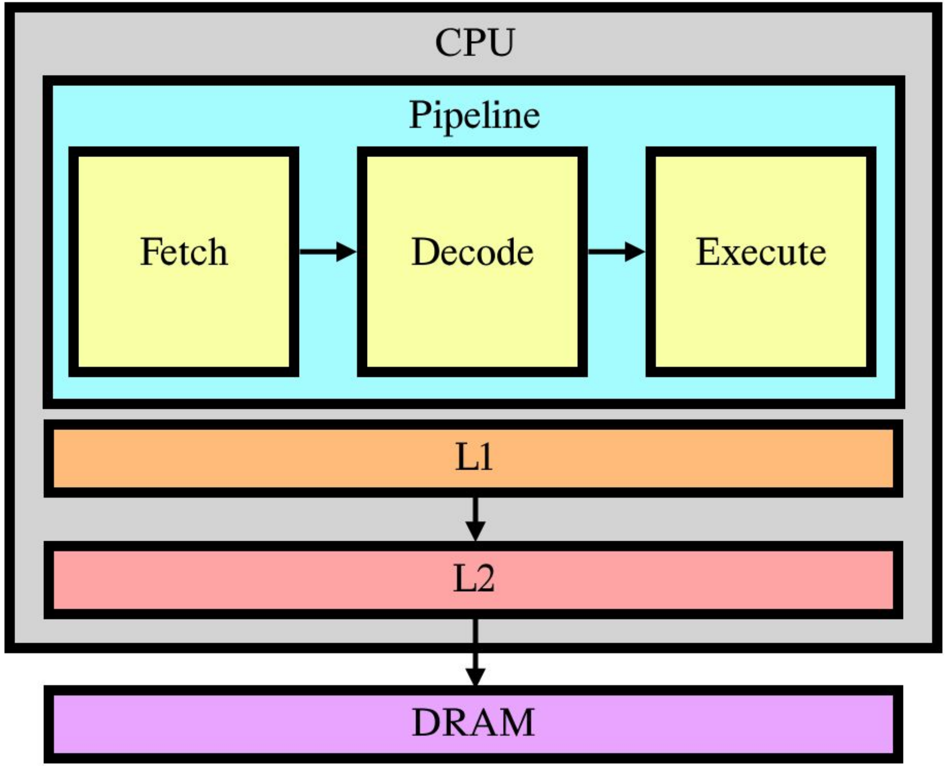
Hardware and Software Optimizations for Deep Learning Workloads on Graphics Processing Units

Justin Garrigus

UNT COLLEGE OF ENGINEERING

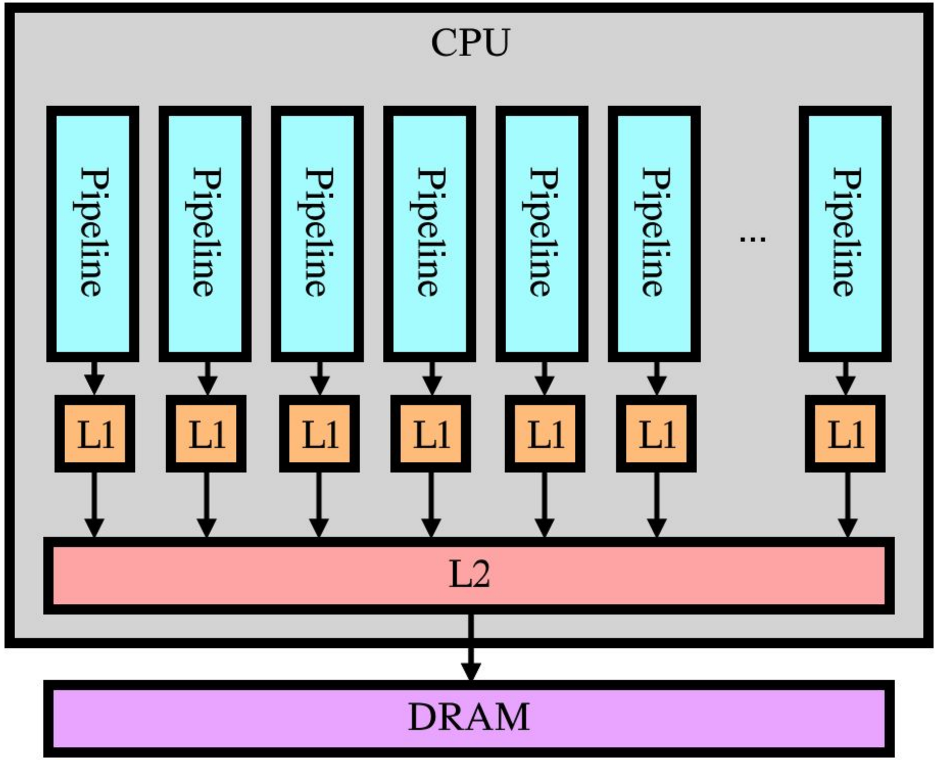
Single-core Processor

- Single pipelines
- One instruction type executed at a time



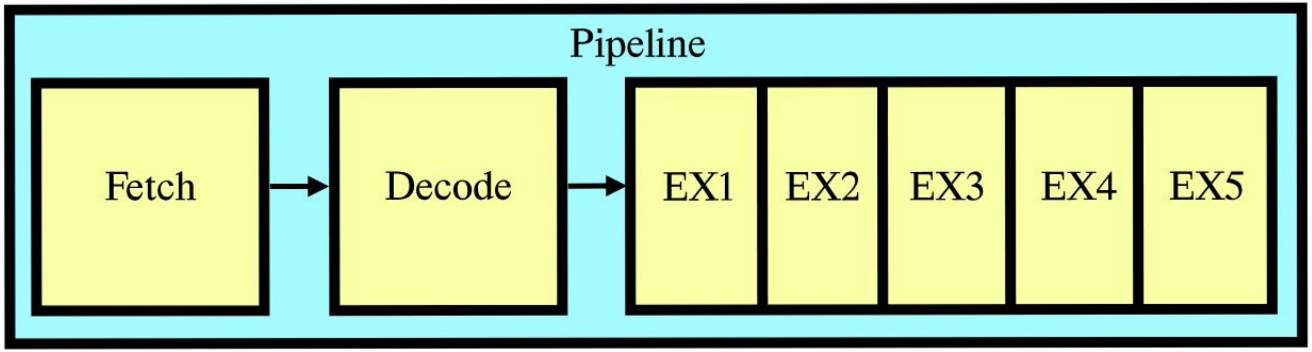
Multi-core Processor

- Multiple pipelines
- Multiple instruction types executed at a time



Vector Processor: Pipelined Execution Unit

- Sub-pipelines
- Execution phase split
- 1 CPI on average for vectors

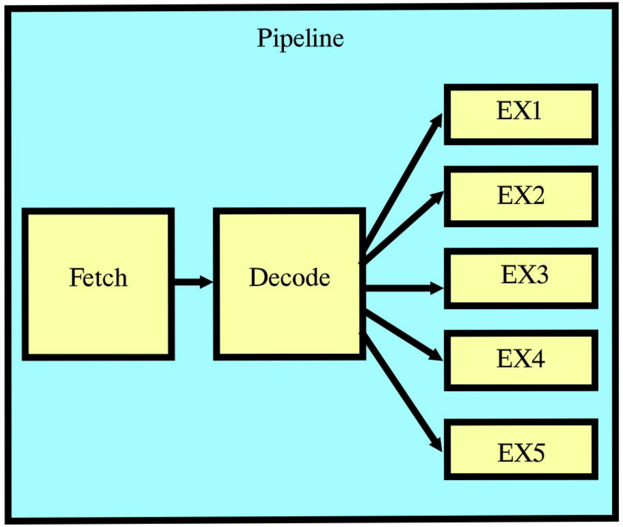


MUL[0] = 6 cycles
MUL[1] = 7 cycles
MUL[2] = 8 cycles
MUL[3] = 9 cycles
MUL[4] = 10 cycles
MUL[5] = 11 cycles
…
MUL[99] = 105 cycles

105 / 99 ~= 1 CPI

Vector Processor: Duplicated Execution Units

- No sub-pipelines
- Duplicated pipelines, equal to startup time
- 1 CPI on average for vectors

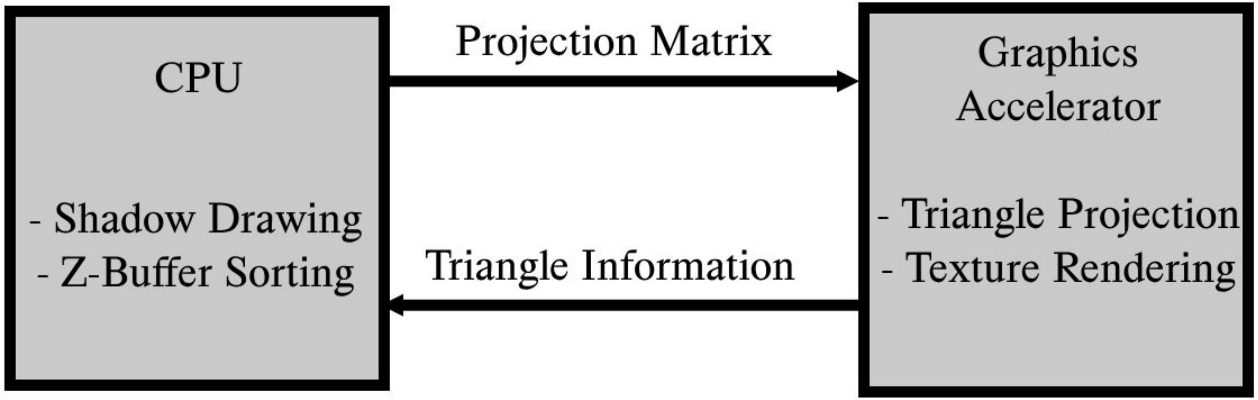


MUL[0] = 6 cycles
MUL[1] = 6 cycles
MUL[2] = 6 cycles
MUL[3] = 6 cycles
MUL[4] = 6 cycles
MUL[5] = 12 cycles
MUL[6] = 12 cycles
MUL[7] = 12 cycles
…
MUL[99] = 120 cycles

120 / 99 ~= 1 CPI

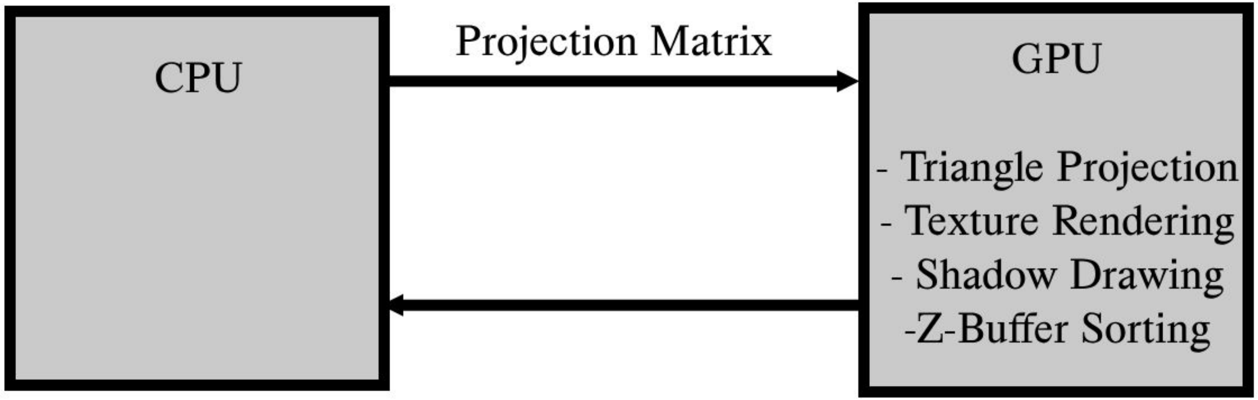
3D Graphics Accelerator

- Polygon rendering
  - Vector translator
- Shadows and textures still on host



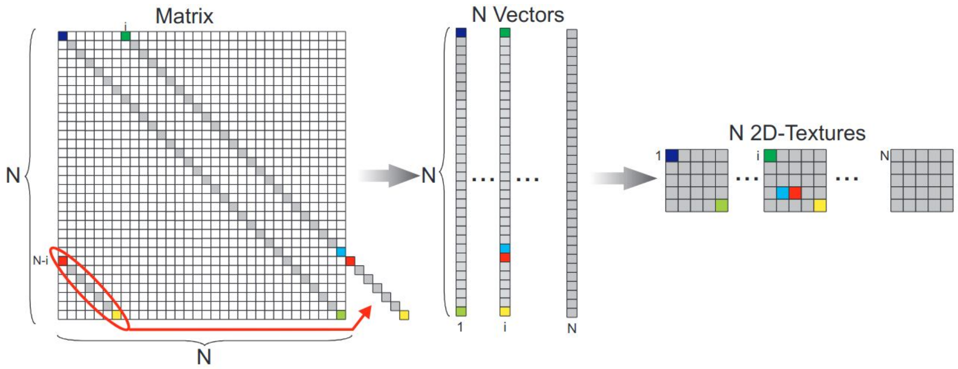
NVIDIA GeForce 256: First “Graphics Processing Unit”

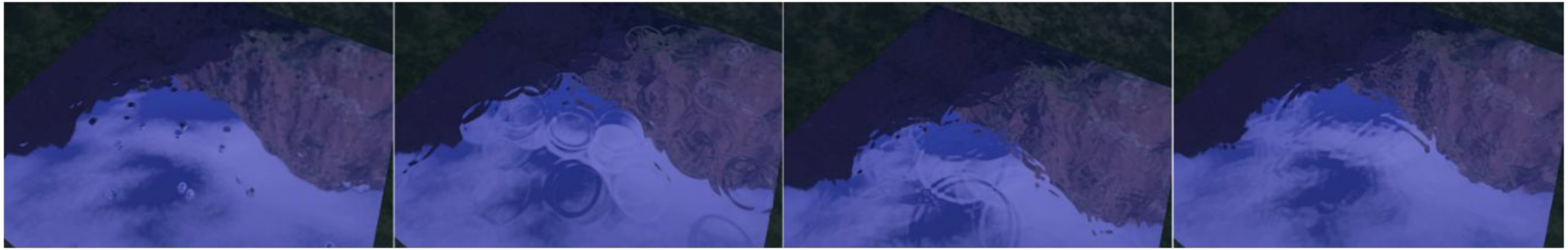
- Marketing term
- Shaders, lighting, and more on device



GPUs for Linear Algebra

- Vector arithmetic
- Matrix-vector product
- Sparse matrices
- General numerical computation
  - Conjugate gradient method
    - Solving sparse $Ax = b$
  - Gauss-Seidel solver
    - Solving $x^i = Lx^i + (D + U)x^{(i-1)}$





J. Kruger, R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms", ACM Transactions on Graphics, 2003.

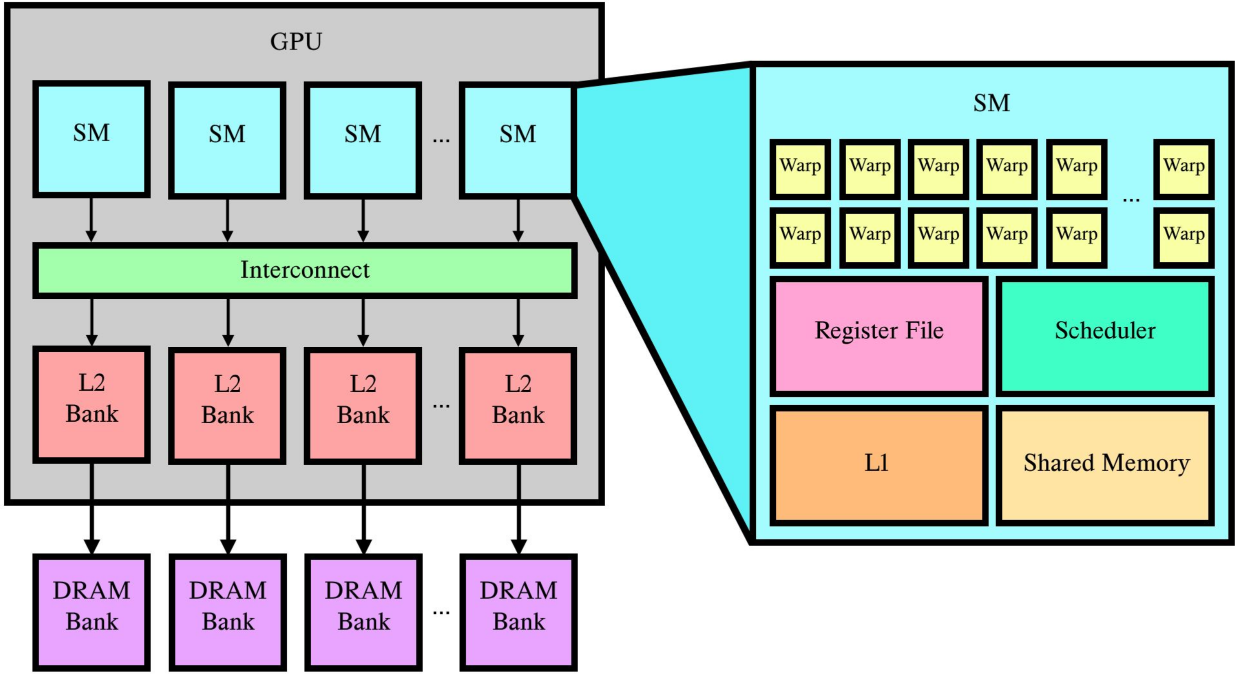
CUDA

- Host (CPU) code and device (GPU) code
- Host data and device data
    - cudaMemcpy, send data across wire
- vector_add executed 10,000 times
- CUDA → PTX → SASS

```
__global__ void vector_add(int* result, int* op1, int* op2, int length) {
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if (idx < length)
    result[idx] = op1[idx] + op2[idx];
}

void main() {
  int length = 10000;
  int *h_vec1 = create_and_fill_array(length);
  int *h_vec2 = create_and_fill_array(length);

  int *d_vec1;
  cudaMalloc(&d_vec1, sizeof(int) * length);
  cudaMemcpy(d_vec1, h_vec1, sizeof(int) * length, cudaMemcpyHostToDevice);

  int *d_vec2;
  cudaMalloc(&d_vec2, sizeof(int) * length);
  cudaMemcpy(d_vec2, h_vec1, sizeof(int) * length, cudaMemcpyHostToDevice);

  int *d_result;
  cudaMalloc(&d_result, sizeof(int) * length);

  vector_add<<<length / 1024, 1024>>>(d_result, d_vec1, d_vec2, length);
  cudaDeviceSynchronize();

  int *h_result = (int*)malloc(sizeof(int) * length);
  cudaMemcpy(h_result, d_result, sizeof(int) * length, cudaMemcpyDeviceToHost);
}
```

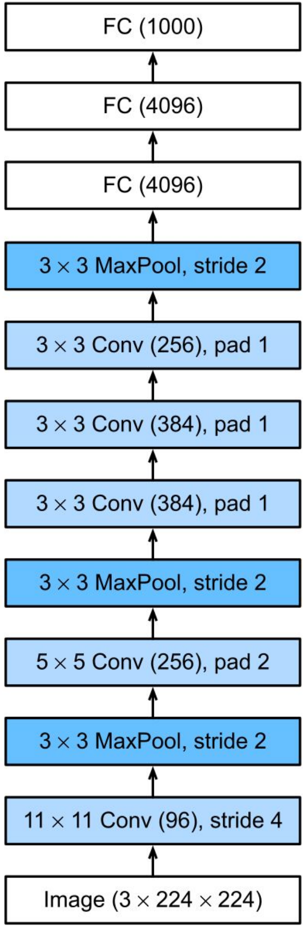
Graphics Processing Units

- Warps
  - Multiple lanes
- Latency hiding
  - Scheduler
  - Banked memory
  - MSHRs
- Tensor cores

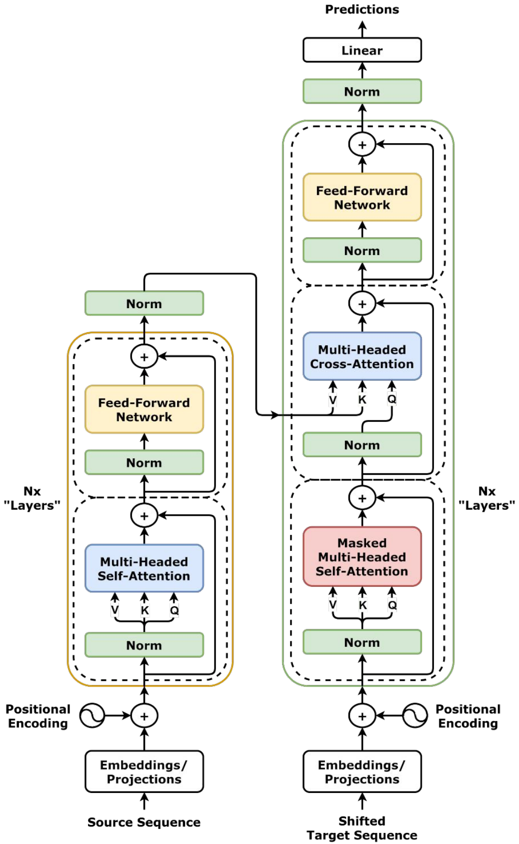


Deep Learning

- Deep neural networks
  - 1990: LeNet
  - 2012: AlexNet
  - 2014: VGG
  - 2015: ResNet
  - 2017: Transformer
- Compute-heavy
- Lots of composite functions



AlexNet



Transformer

Deep Learning Compilers

- Find optimized software representation
- Nonlinear problem
- Direct and indirect optimizations

```
for (n=0; n<N; n++)
 for (c_o=0; c_o<C_o; c_o++)
  for (c_i=0; c_i<C_i; c_i++)
   for (h_o=0; h_o<H_o; h_o++)
    for (w_o=0; w_o<W_o; w_o++)
     for (h_k=0; h_k<H_k; h_k++)
      for (w_k=0; w_k<W_k; w_k++)
       Ofm[n][c_o][h_o][w_o] +=
       Ifm[n][c_i][h_o+h_k][w_o+w_k] * W[c_o][c_i][h_k][w_k]
```

Off-chip memory access pattern

```
for (n=0; n<N; n++)
 for (c_oe=0; c_oe<C_o; c_oe+=TC_o)
  for (c_ie=0; c_ie<C_i; c_ie+=TC_i)
   for (h_oe=0; h_oe<H_o; h_oe+=TH_o)
    for (w_oe=0; w_oe<W_o; w_oe+=TW_o)
```

On-chip computation

```
     for (c_o=c_oe; c_o<c_oe+TC_o; c_o++)
      for (c_i=c_ie; c_i<c_ie+TC_i; c_i++)
       for (h_o=h_oe; h_o<h_oe+TH_o; h_o++)
        for (w_o=w_oe; w_o<w_oe+TW_o; w_o++)
         for (h_k=0; h_k<H_k; h_k++)
          for (w_k=0; w_k<W_k; w_k++)
           Ofm[n][c_o][h_o][w_o] +=
           Ifm[n][c_i][h_o+h_k][w_o+w_k] * W[c_o][c_i][h_k][w_k]
```

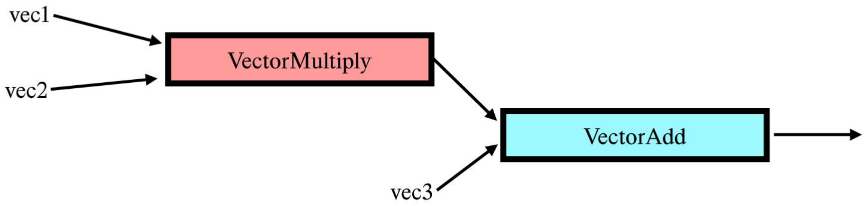
M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, M. Shafique, “Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges and the road ahead,” in IEEE Access, 2020.

Direct Optimizations

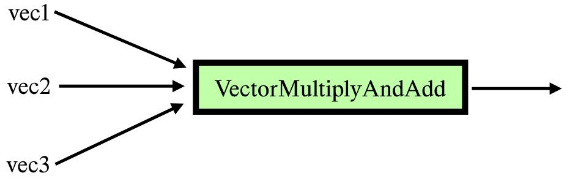
- Optimizations applied without profiling
- “Best guess”
    - Value propagation
    - Operator inlining
    - Strength reduction
    - Loop restructuring

```
1 TransformedIR:
2 /* define loop extents as variables for code brevity */:
3 extent_ko, extent_ki = (C_k / TB_tile_k), (TB_tile_k / Warp_tile_k)
4  /* Declare buffer size. */
5  alloc A_shared[3][...]
6  alloc A_reg[2][...]
7  /* Prologue for A_shared and A_reg */
8  for ko in 0 .. 2:
9      /* load into shared memory buffer (same as Line 15-17) */
10 for ki in 0 .. 1:
11     /* load into reg. buffer (same as Line 24-27) */
12 for ko in 0 .. extent_ko:
13     /* load into shared memory buffer  */
14     /* guard data copy with producer primitives at Line 15 and Line 17 */
15     A_shared.producer_acquire()
16     async_memcpy(A_shared [ (ko + 2) % 3 ][...], A[..., (ko + 2) % extent_ko ])
17     A_shared.producer_commit()
18    /* compute with data in shared memory buffer */
19    /* guard data usage with consumer primitives at Line 22 and Line 30 */
20    for ki in 0 .. extent_ki:
21         if (( ki  + 1 ) % extent_ki  )== 0:
22             A_shared.consumer_wait()
23             /* load into register buffer */
24         async_memcpy(
25             A_reg [ (ki + 1) % 2 ][...],
26             A_shared [ (ko + ((ki+1) / extent_ki) ) % 3 ][..., (ki + 1) % extent_ki ]
27         )
28        /* tensor-core compute with data in register buffer */
29        wmma(A_reg [ (ki % 2) ] [...], ...)
30         A_shared.consumer_release()
```

Value Propagation and Operator Inlining

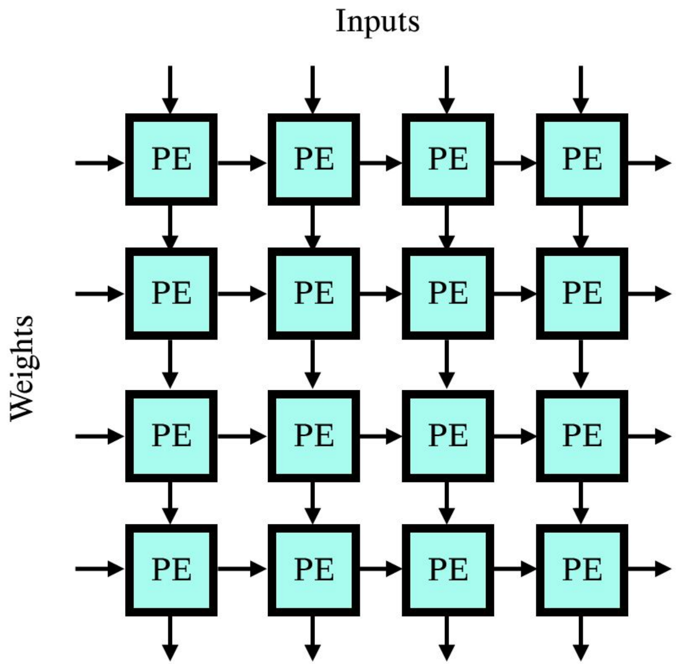



```
__global__ void VectorMultiply(int* result, int length, int* vec1, int* vec2) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < length)
        result[idx] = vec1[idx] * vec2[idx];
}

__global__ void VectorAdd(int* result, int length, int* vec1, int* vec2) {
    int idx = threadIdx + blockDim.x * blockIdx.x;
    if (idx < length)
        result[idx] = vec1[idx] + vec2[idx];
}
```




```
__global__ void VectorMultiplyAndAdd(int* result, int* vec1, int* vec2, int* vec3, int length) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < length)
        result[idx] = vec1[idx] * vec2[idx] + vec3[idx];
}
```

Strength Reduction with Tensor Cores

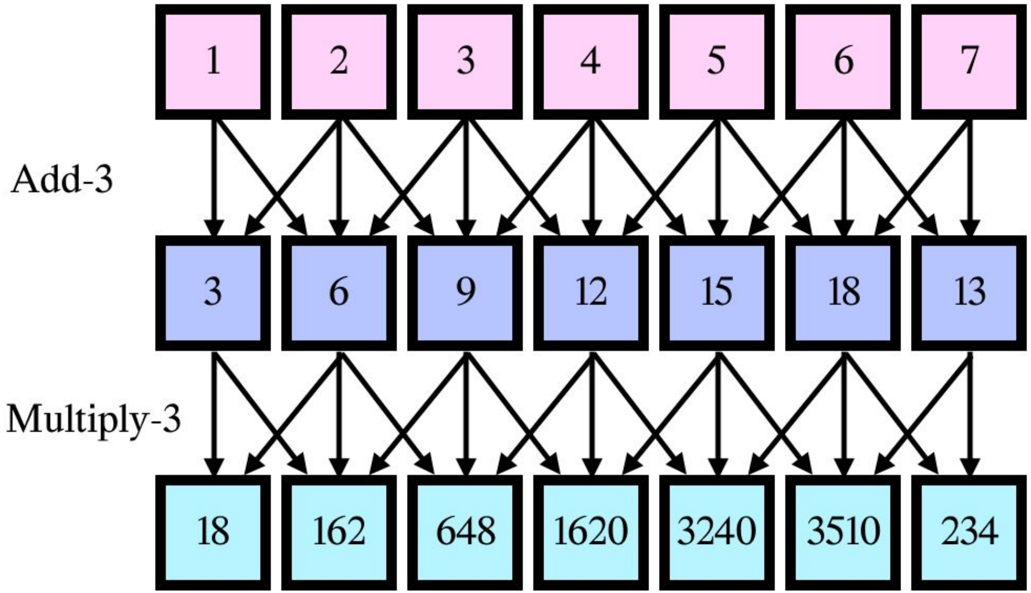



```
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        for (int r = 0; r < k; r++)
            C[i,j] += A[i,r] * B[r,j];
```

```
Buffer<fp16, 16, 16> tensorA;
Buffer<fp16, 16, 16> tensorB;
Buffer<fp32, 16, 16> tensorC;
for (int i = 0; i < n; i += 16) {
    for (int j = 0; j < m; j += 16) {
        for (int r = 0; r < k; r += 16) {
            tensorA = Load(A[i:16, r:16]);
            tensorB = Load(B[r:16, j:16]);
            tensorC += TensorCore(tensorA, tensorB);
        }
        Store(C[i:16, j:16], tensorC);
    }
}
```

Loop Restructuring

- Loop unrolling
- Vectorization
- Loop tiling



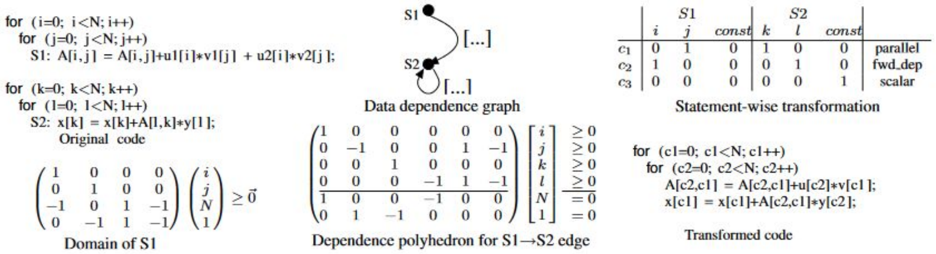
Polyhedral Model

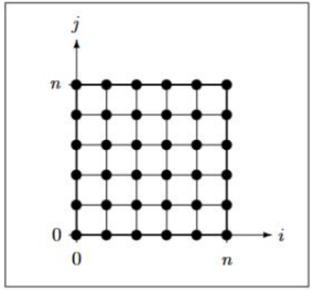
- Describe loop with affine inequalities
  - Iteration domain forms lattice
  - Dependency graph links iterations
- Represent with mathematical expression
  - "Scattering" function describes statement instances
  - Create new scattering by solving equations
    - Different possible solutions
  - Scattering creates new code

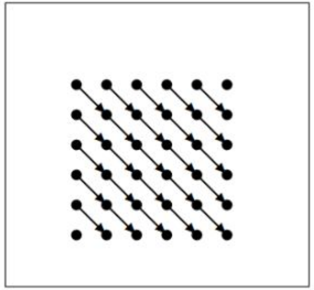
Algorithm 1 SCoP pseudocode

Require: $n, a[n], b[n], c[n]$

for $i \leftarrow 0, n$ **do**
    for $j \leftarrow n, 0$ **do**
        if $i == 0 \,\|\, j == n$ **then**
            $c[i+j] \leftarrow a[i] \times b[j]$ ▷ S1
        else
            $c[i+j] \leftarrow c[i+j] + a[i] \times b[j]$ ▷ S2
        end if
    end for
end for



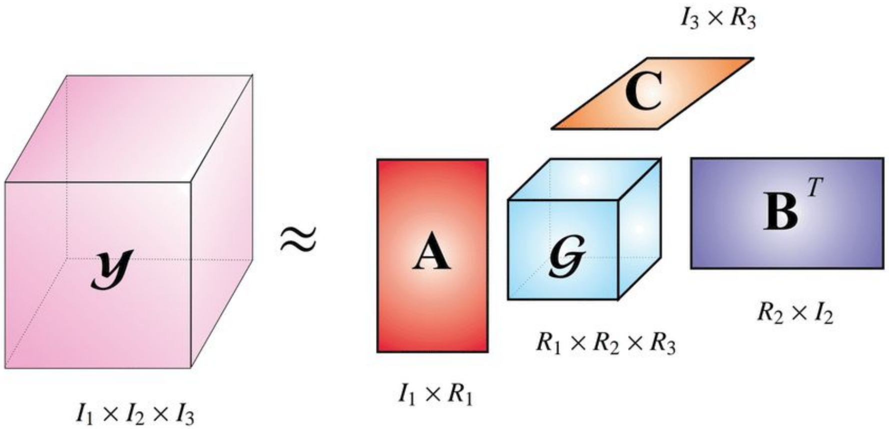




- U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, *"A practical automatic polyhedral parallelizer and locality optimizer," in ACM SIGPLAN Notices, 2008.*
- *C. Lengauer, "Loop parallelization in the polytope model," in 4th International Conference on Concurrency Theory (CONCUR'93), pages 398–416, 1993.*

Tensor Operations

- Overparameterized networks
  - Tucker decomposition
    - "Core" tensor and matrices
  - Canonical polyadic decomposition
    - List of vectors
  - Dimensionality reduction
  - Clustering
  - Noise reduction
- Some operators are naturally tensors
  - Convolution
  - Multi-headed attention
    - Remove "flatten" operator



Indirect Optimizations

- An optimization may cause adverse effects
- Optimization space very large for many deep-learning programs
  - Local (loop-wide, function-wide) vs. global (program-wide)

```
for (int i = 0; i < 256; i++) {        // Loop "L1"
    a[i] = 0;
    for (int j = 0; j < 256; j++)      // Loop "L2"
        a[i] += b[j + i] + c[j + i];
}
```

Program to optimize: L1 (outer), L2 (inner)

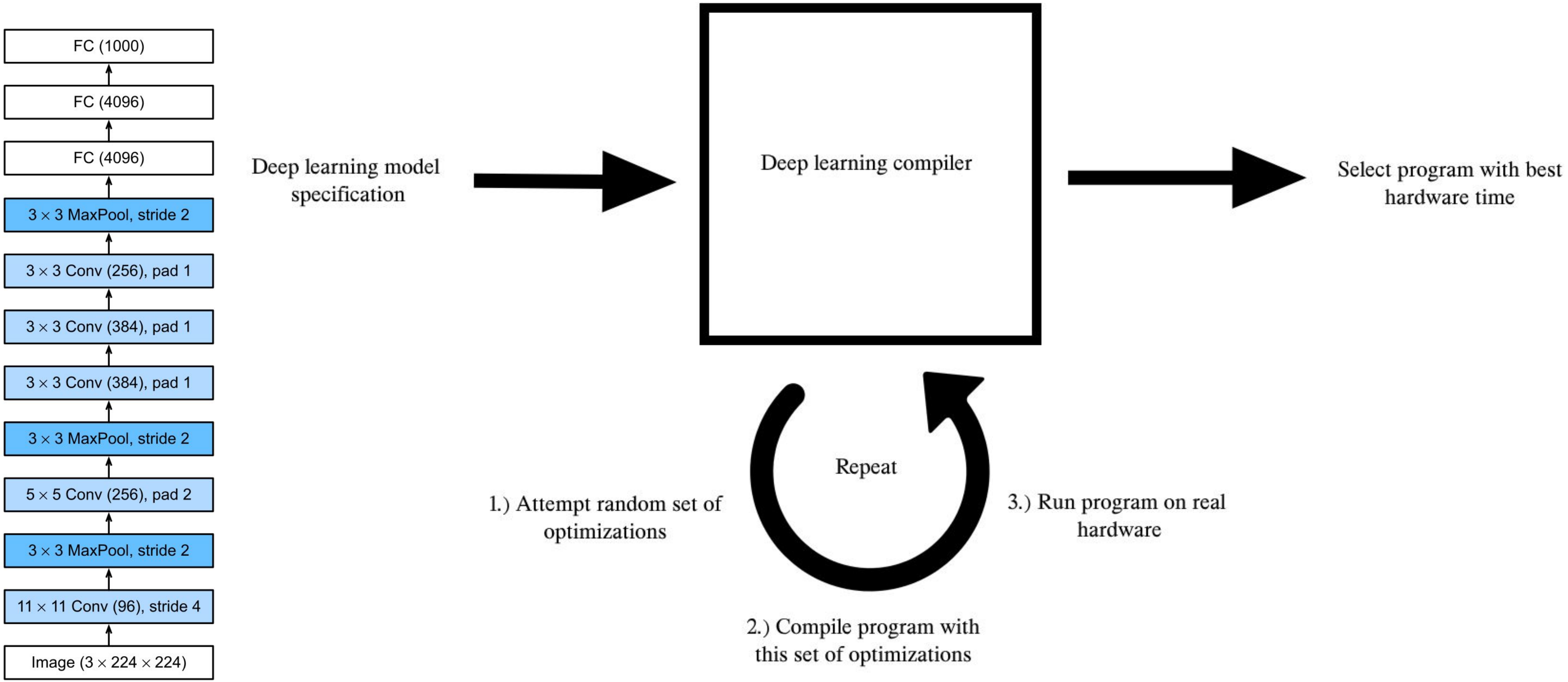
```
for (int i = 0; i < 256; i += 4) {
    a[i  ] = 0;
    a[i+1] = 0;
    a[i+2] = 0;
    a[i+3] = 0;
    for (int j = 0; j < 256; j++)
        a[i  ] += b[j + i    ] + c[j + i    ];
    for (int j = 0; j < 256; j++)
        a[i+1] += b[j + i + 1] + c[j + i + 1];
    for (int j = 0; j < 256; j++)
        a[i+2] += b[j + i + 2] + c[j + i + 2];
    for (int j = 0; j < 256; j++)
        a[i+3] += b[j + i + 3] + c[j + i + 3];
}
```

Unroll factor: L1 = 4, L2 = 1

```
for (int i = 0; i < 256; i++) {
    a[i] = 0;
    for (int j = 0; j < 256; j += 8) {
        a[i] += b[j     + i] + c[j     + i] +
                b[j + 1 + i] + c[j + 1 + i] +
                b[j + 2 + i] + c[j + 2 + i] +
                b[j + 3 + i] + c[j + 3 + i] +
                b[j + 4 + i] + c[j + 4 + i] +
                b[j + 5 + i] + c[j + 5 + i] +
                b[j + 6 + i] + c[j + 6 + i] +
                b[j + 7 + i] + c[j + 7 + i];
    }
}
```

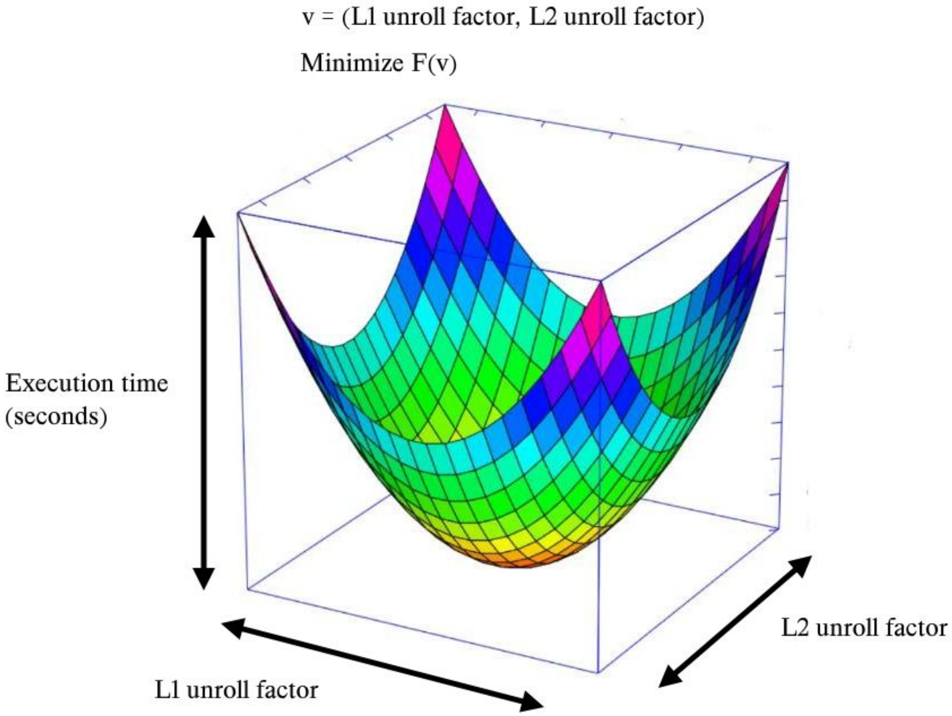
Unroll factor: L1 = 1, L2 = 8

Indirect Optimizations

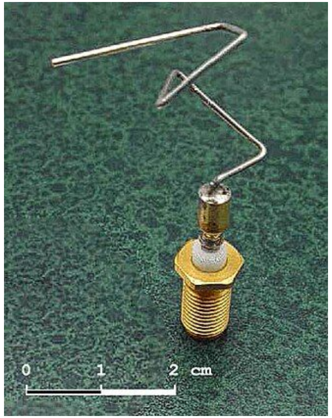


Optimization Pool

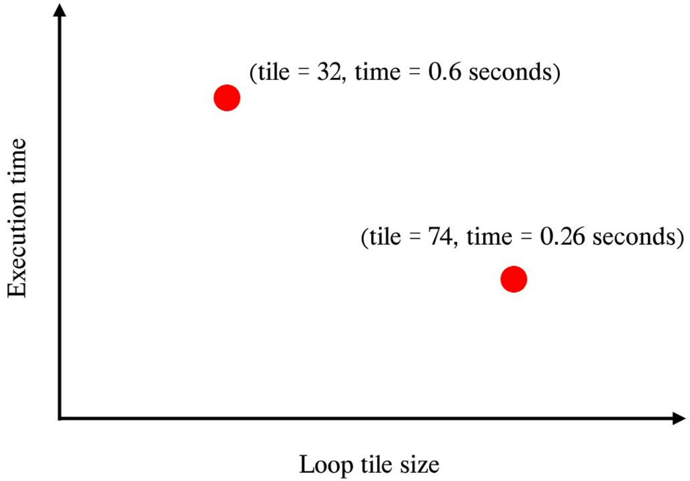
- Create pool of possible optimizations
- Form into vector
  - | v1 - v2 | < ε, optimizations are similar
- On-device measurement
  - Time in seconds = F(v)
- Approximate minimum of function
  - Random search
  - Grid search
  - Genetic algorithms
  - Simulated annealing
  - Deep learning



Genetic Algorithms and Simulated Annealing

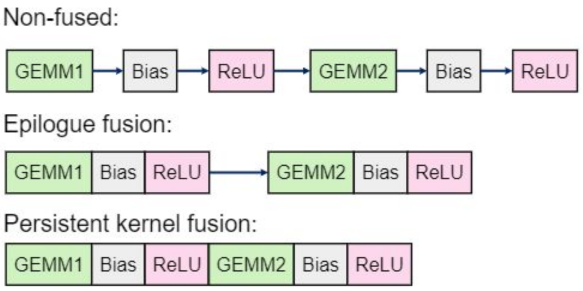


- Requires constraint that nearby points have nearby costs
- Device measurements are expensive
- Simulated annealing
  - Single-variable genetic
- Genetic algorithms
  - Selection
  - Crossover
  - Mutation

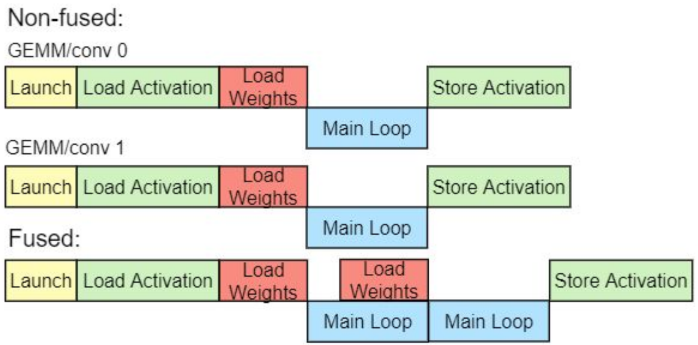


BOLT: Auto-tuner for Template Library Parameters

- Use template libraries (CUTLASS)
  - Instead of generating low-level CUDA code
- Design parameters:
  - Thread/block counts
  - Operator fusion
  - Data alignment
  - Synchronization



(a) The graph view of persistent kernel fusion



(b) The kernel view of persistent kernel fusion

J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, "Bolt: Bridging the Gap Between Auto-Tuners and Hardware-Native Performance", in Proceedings of Machine Learning and Systems (MLSys), 2022, pages 204-216.

Halide: Graph Optimizer and Auto-Tuner for Image Pipelines

- Memory-limited stencils interconnected in a graph
    - Sharpen, blur, etc.
- Performance limited by interleaving allocation, execution, and communication
    - Separate algorithm (high-level operator) from schedule (low-level implementation) in domain-specific language
    - Automatically search

```
blurx(x,y) =  in(x-1,y) +     in(x,y) +     in(x+1,y)
out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

(Baseline) A 3x3 two-stage blur algorithm.

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x]=blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

(1) Compute and store every point in `blurx` before doing anything in the output. High parallelism, but little producer-consumer locality.

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i]= in[y-1+i][x-1]+in[y-1+i][x]+in[y-1+i][x+1]
    out[y][x] = blurx[0] + blurx[1] + blurx[2]
```

(2) Could compute each point in `blurx` immediately before the point that uses it. High parallelism and maximum locality, but lots of redundant work.

```
alloc out[2046][3072]
alloc blurx[3][3072]
for each y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x]=in[y+1][x-1]+in[y+1][x]+in[y+1][x+1]
    if y < 1: continue
    out[y][x] = blurx[(y-1)%3][x]
              + blurx[ y % 3 ][x]
              + blurx[(y+1)%3][x]
```

(3) Could interleave the two stages over a sliding window, wasting no work but no opportunities for parallelization.

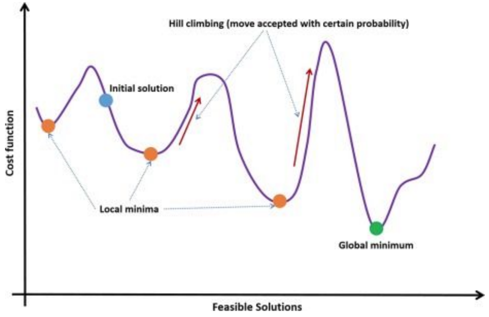
```
alloc out[2046][3072]
for each ty in 0..2048/8:
  alloc blurx[-1..1][3072]
  for y in -2..8:
    for x in 0..3072:
        blurx[(y+1)%3][x] = in[ty*8+y+1][tx*32+x-1]
                          + in[ty*8+y+1][tx*32+x]
                          + in[ty*8+y+1][tx*32+x+1]
    if y < 0: continue
    for x in 0..3072:
      out[ty*8+y][x] = blurx[(y-1)%3][x]
                     + blurx[ y % 3][x]
                     + blurx[(y+1)%3][x]
```

(4) Each version 1-3 had a pitfall. The optimal solution is somewhere in the middle, like sliding a window over strips of independent scanlines. Hardware independent, so this is searched for automatically.

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines", in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13), 2013, pages 519-530.

Apache TVM

- Relay
    - High-level IR
    - Computation graph
    - Optimizations allow graph rewriting
- Tensor Expression
    - Low-level IR
    - “Template” defined from placeholders, knobs, tuners, parallelism, vectorization
    - Search space + cost model
        - AutoTVM
            - Simulated annealing



e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

x_0 default code

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

s_1 loop tiling

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

$x_1 = g(e, s_1)$

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
            A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

s_2 tiling, map to micro kernel intrinsics

```
yo,xo,ko,yi,xi,ki = s[C].title(y,x,k,8,8,8)
s[C].tensorize(yi, intrin.gemm8x8)
```

$x_2 = g(e, s_2)$

```
for yo in range(128):
  for xo in range(128):
    intrin.fill_zero(C[yo*8:yo*8+8][xo*8:xo*8+8])
    for ko in range(128):
      intrin.fused_gemm8x8_add(
        C[yo*8:yo*8+8][xo*8:xo*8+8],
        A[ko*8:ko*8+8][yo*8:yo*8+8],
        B[ko*8:ko*8+8][xo*8:xo*8+8])
```

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L. and Guestrin, C. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. Proceedings of the 13th USENIX Symposium on Operating System Design and Implementation (OSDI’18).

Ansor: Hierarchical Optimizer

- Rule-based kernel expansion
  - Input: mathematical notation
  - Evolutionary search
  - Rules generate sketches (high-level)
    - Don’t specify tile dimensions, parallelization, or unrolling
  - Randomly label loops as parallel or vectorized; randomly assign tile dimensions; genetic algorithm iterates

Example Input 1:

* The mathmetical expression:

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

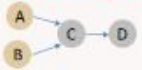
$$D[i,j] = \max(C[i,j], 0.0)$$

where $0 \leq i, j, k < 512$

* The corresponding naive program:

```
for i in range(512):
  for j in range(512):
    for k in range(512):
      C[i, j] += A[i, k] * B[k, j]
for i in range(512):
  for j in range(512):
    D[i, j] = max(C[i, j], 0.0)
```

* The corresponding DAG:



No	Rule Name
1	Skip
2	Always Inline
3	Multi-level Tiling
4	Multi-level Tiling with Fusion
5	Add Cache Stage
6	Reduction Factorization
...	User Defined Rule

Generated sketch 1

```
for i.0 in range(TILE_I0):
 for j.0 in range(TILE_J0):
  for i.1 in range(TILE_I1):
   for j.1 in range(TILE_J1):
    for k.0 in range(TILE_K0):
     for i.2 in range(TILE_I2):
      for j.2 in range(TILE_J2):
       for k.1 in range(TILE_I1):
        for i.3 in range(TILE_I3):
         for j.3 in range(TILE_J3):
          C[...] += A[...] * B[...]
    for i.4 in range(TILE_I2 * TILE_I3):
     for j.4 in range(TILE_J2 * TILE_J3):
      D[...] = max(C[...], 0.0)
```

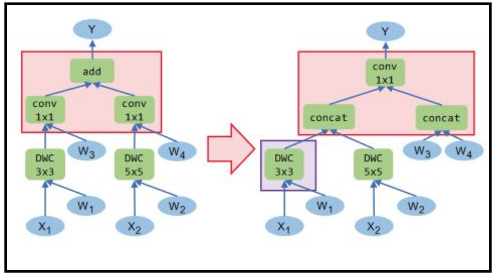
Sampled program 1

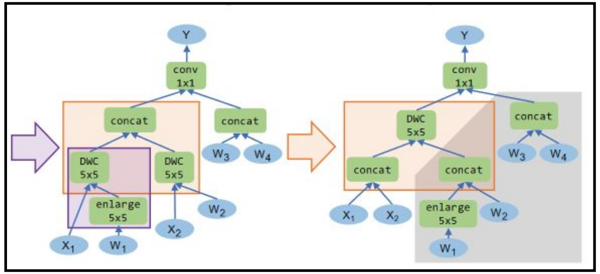
```
parallel i.0@j.0@i.1@j.1 in range(256):
  for k.0 in range(32):
    for i.2 in range(16):
      unroll k.1 in range(16):
        unroll i.3 in range(4):
          vectorize j.3 in range(16):
            C[...] += A[...] * B[...]
  for i.4 in range(64):
    vectorize j.4 in range(16):
      D[...] = max(C[...], 0.0)
```


Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C.H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K. and Gonzalez, J.E., 2020, November. Ansor: Generating high-performance tensor programs for deep learning. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI’20).

Layout Structure Organization

- TASO: substitute subgraphs using mathematical properties of operators
  - Theorem prover
- DNNFusion: based on rules and categories
  - Different fusion/codegen rules depending on label (one-to-one, etc.)
  - Check associativity/distributivity/commutativity of each operator
- Apollo: rule-based under polyhedral model





$$\forall s, p, x, y, z.\ \mathrm{conv}(s, p, \mathrm{A_{none}}, \mathrm{ewadd}(x, y), z) = \mathrm{ewadd}(\mathrm{conv}(s, p, \mathrm{A_{none}}, x, z), \mathrm{conv}(s, p, \mathrm{A_{none}}, y, z))$$

Example mathematical property, asserts convolution without activation is linear in first argument.

Table 3. Mapping type analysis. The first column and the first row (both without color) show the mapping types of first and second operators, respectively, before fusion, and the colored cells show the mapping type of the operator after fusion. Green implies that these fusion combinations can be fused directly (i.e., they are profitable). Red implies that these fusions are unprofitable. Yellow implies that further profiling is required to determine profitability.

Second op / First op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-One	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-Many	One-to-Many	One-to-Many	✕	One-to-Many	One-to-Many
Many-to-Many	Many-to-Many	Many-to-Many	✕	Many-to-Many	Many-to-Many
Reorganize	Reorganize	One-to-Many	Many-to-Many	Reorganize	Reorganize
Shuffle	Shuffle	One-to-Many	Many-to-Many	Reorganize	Shuffle

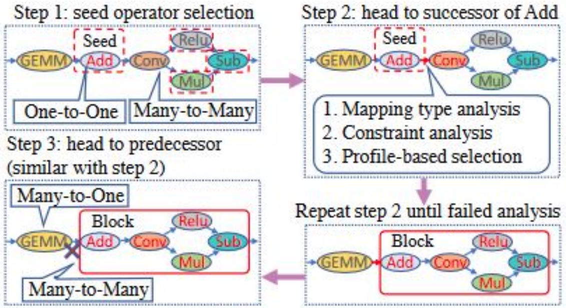
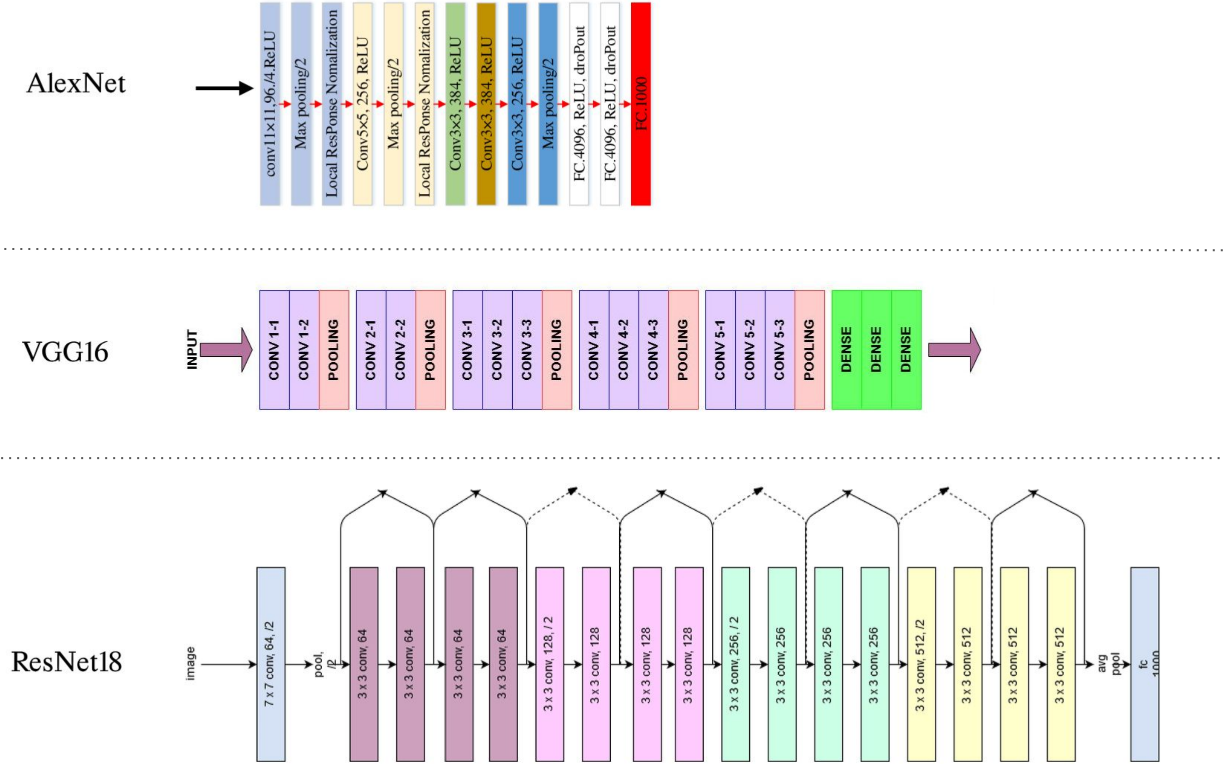


Figure 3. An example of fusion plan exploration. Assume Add, Conv, Relu, Mul, and Sub have identical output shape and IRS size.

- Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, “TASO: Optimizing deep learning computation with automatic generation of graph substitutions,” in Proceedings of the 27th ACM Symposium on Operating System Principles (SOSP), 2019, pages 47-62.
- J. Zhao, X. Gao, R. Xia, Z. Zhang, D. Chen, L. Chen, R. Zhang, Z. Geng, B. Cheng, and X. Jin, “Apollo: Automatic partition-based operator fusion through layer by layer optimization,” in Proceedings of the 5th MLSys Conference (MLSys), 2022.
- W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, “DNNFusion: Accelerating deep neural networks execution with advanced operator fusion,” in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2021.

Image Recognition Networks

- VGG-16 spends 93.1% of execution time running convolution



2D Convolution

Step 1

Input Image

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

*

Filter

5	3	1
6	2	8
7	3	9

1 * 5 + 2 * 3 + 3 * 1 +
6 * 6 + 7 * 2 + 8 * 8 +
11 * 7 + 12 * 3 + 9 * 13 = 358

Step 2

Input Image

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

*

Filter

5	3	1
6	2	8
7	3	9

2 * 5 + 3 * 3 + 4 * 1 +
7 * 6 + 8 * 2 + 9 * 8 +
12 * 7 + 13 * 3 + 14 * 9 = 402

Lowering

Input Image

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

*

Filter

5	3	1
6	2	8
7	3	9

1	2	3	6	7	8	11	12	13	Input Image (first iteration)

*

5	3	1	6	2	8	7	3	9	Filter (first channel)

Lowering and Duplication

1	2	3	6	7	8	11	12	13
2	3	4	7	8	9	12	13	14
3	4	5	8	9	10	13	14	15
6	7	8	11	12	13	16	17	18
7	8	9	12	13	14	17	18	19
8	9	10	13	14	15	18	19	20
11	12	13	16	17	18	21	22	23
12	13	14	17	18	19	22	23	24
13	14	15	18	19	20	23	24	25

*

5
3
1
6
2
8
7
3
9

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

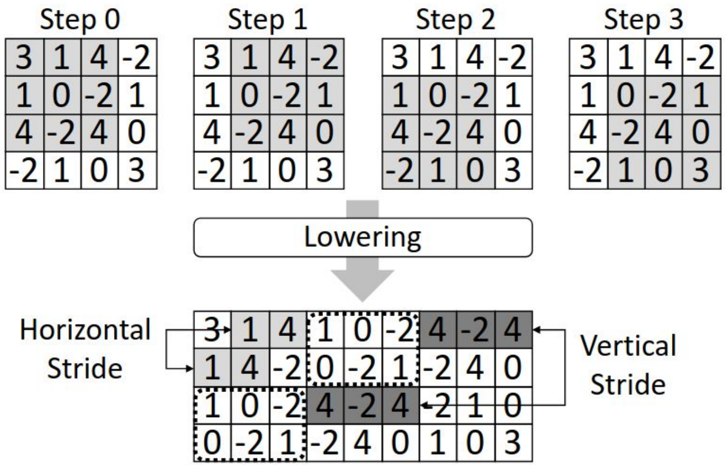
*

5	3	1
6	2	8
7	3	9

- ResNet18 is 14.5x faster, but has 8.4x more memory transactions

Duplo: Reducing Redundant Memory Accesses

- Repetition occurs in predictable locations, if we know convolution parameters
- Register renaming for WMMA registers
  - Load history buffer
  - Proceed to L2 if not exists in L1, else rename



$$\text{dprod}_{\text{total}} = \text{dprod}_{\text{row}} \times \text{dprod}_{\text{col}} : \text{input}_{\text{channels}}$$

$$\text{dprod}_{\text{row}} = 1 + \frac{\text{input}_{\text{rows}} - \text{filter}_{\text{rows}}}{\text{stride}_{\text{rows}}}$$

$$\text{dprod}_{\text{col}} = 1 + \frac{\text{input}_{\text{col}} - \text{filter}_{\text{cols}}}{\text{stride}_{\text{cols}}}$$

$$\text{dprod}_{\text{total}} = \text{dprod}_{\text{row}} \times \text{dprod}_{\text{col}}$$

$$\text{workspace}_{\text{index}} = \frac{\text{address} - \text{input}_{\text{start address}}}{\text{size}_{\text{data type}}}$$

$$\text{workspace}_{\text{row}} = \text{workspace}_{\text{index}} / \text{dprod}_{\text{size}}$$

$$\text{workspace}_{\text{col}} = \text{workspace}_{\text{index}} \bmod \text{dprod}_{\text{size}}$$

$$\text{filter}_{\text{row}} = (\text{workspace}_{\text{row}} / \text{dprod}_{\text{col}}) \times \text{stride}_{\text{rows}}$$

$$\text{filter}_{\text{col}} = (\text{workspace}_{\text{row}} \bmod \text{dprod}_{\text{col}}) \times \text{stride}_{\text{cols}}$$

$$\text{relative}_{\text{row}} = (\text{workspace}_{\text{col}} / \text{filter}_{\text{cols}}) \bmod \text{filter}_{\text{rows}}$$

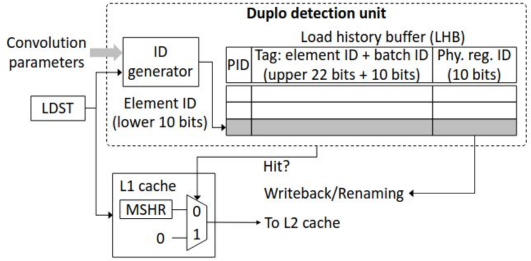
$$\text{relative}_{\text{col}} = \text{workspace}_{\text{col}} \bmod \text{filter}_{\text{cols}}$$

$$\text{channel} = \text{workspace}_{\text{col}} / (\text{filter}_{\text{rows}} \times \text{filter}_{\text{cols}})$$

$$\text{row} = \text{filter}_{\text{row}} + \text{relative}_{\text{row}}$$

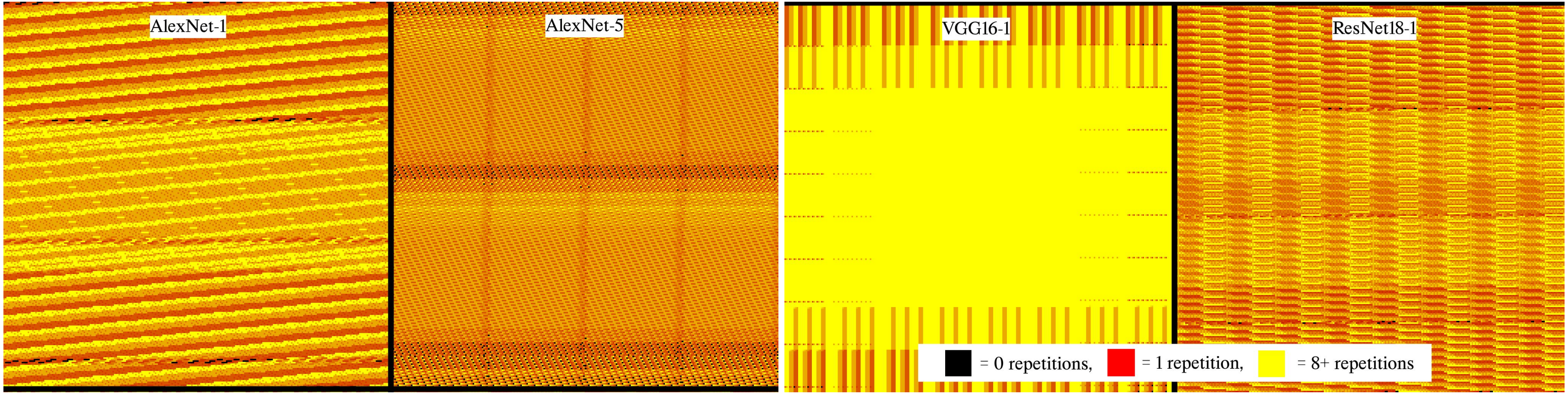
$$\text{col} = \text{filter}_{\text{col}} + \text{relative}_{\text{col}}$$

$$\text{id} = \text{channel} \times (\text{input}_{\text{rows}} \times \text{input}_{\text{cols}}) + (\text{row} \times \text{input}_{\text{cols}}) + \text{col}$$



element_id = F(address)

Effect on the L2 Cache



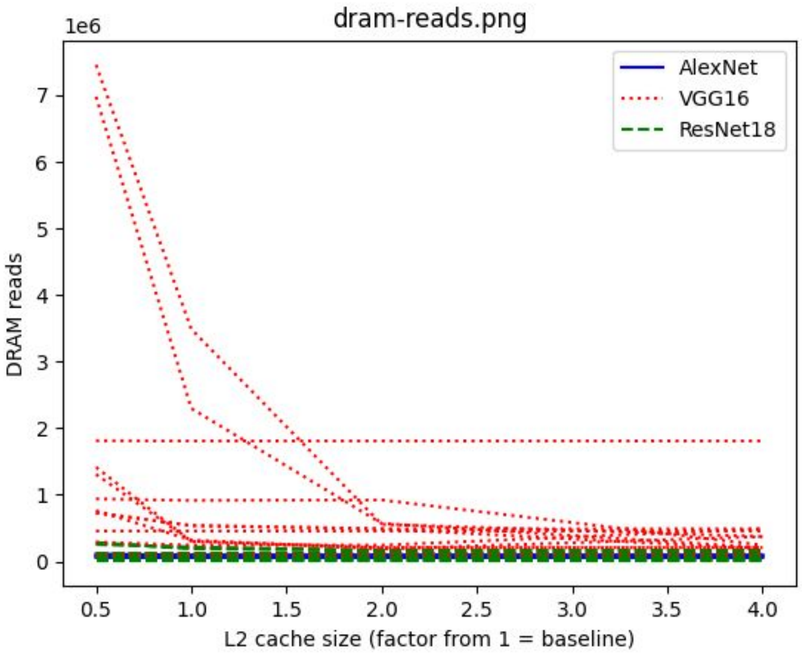
Channels: 3
Filter size: 11x11
Stride: 4

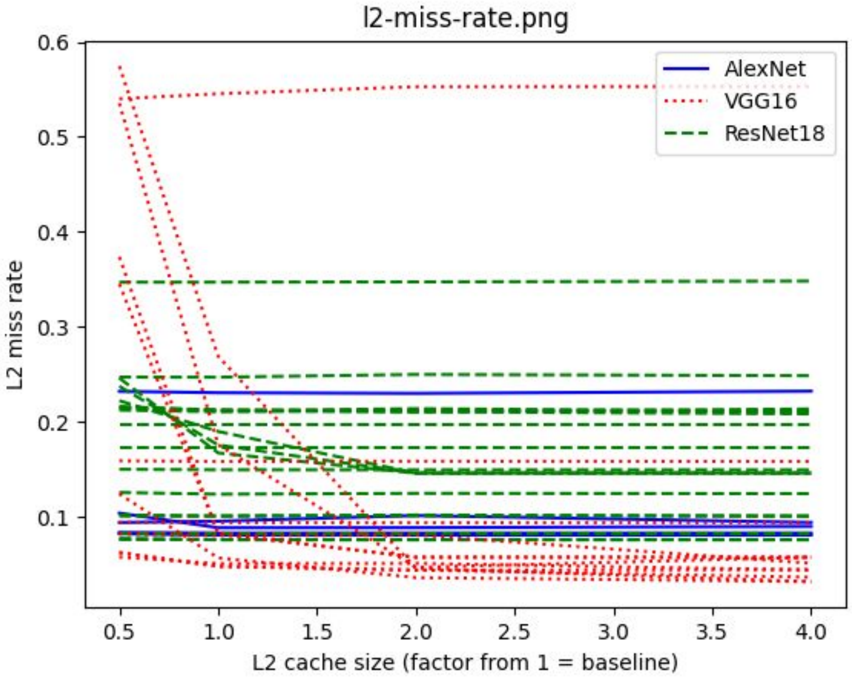
Channels: 256
Filter size: 3x3
Stride: 1

Channels: 3
Filter size: 3x3
Stride: 1

Channels: 3
Filter size: 7x7
Stride: 2

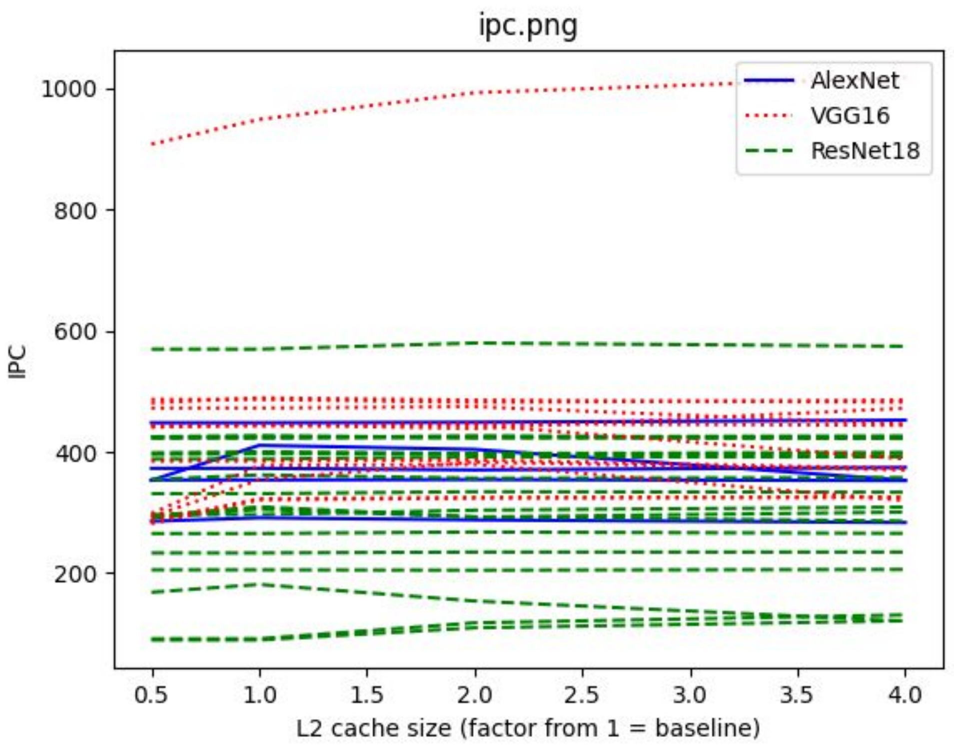
Effect on the L2 Cache



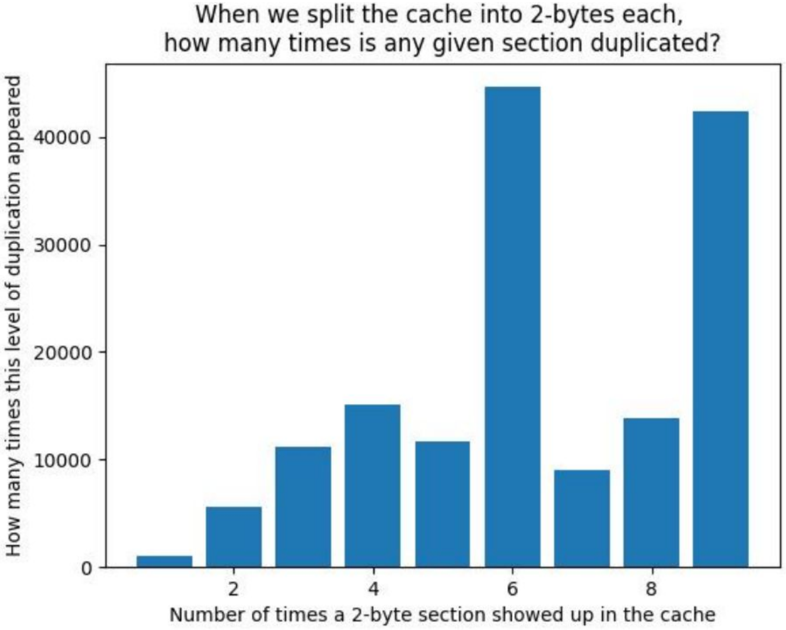


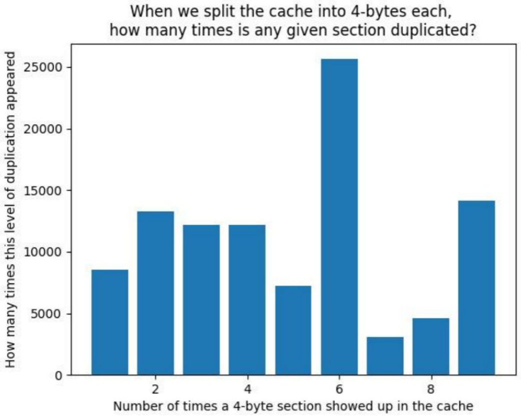
Effect on System Performance

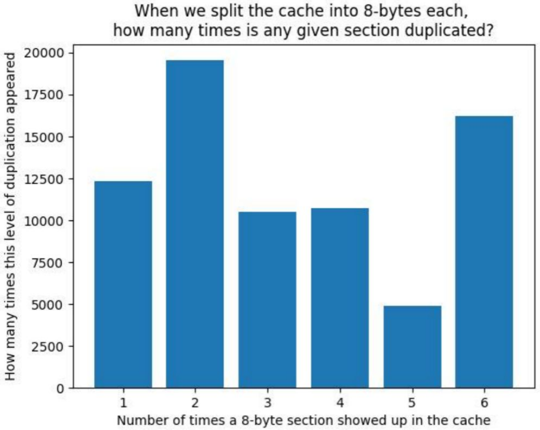
- Increasing cache size will not improve performance
- Instead, halve the cache size
    - Prevent a decrease in IPC

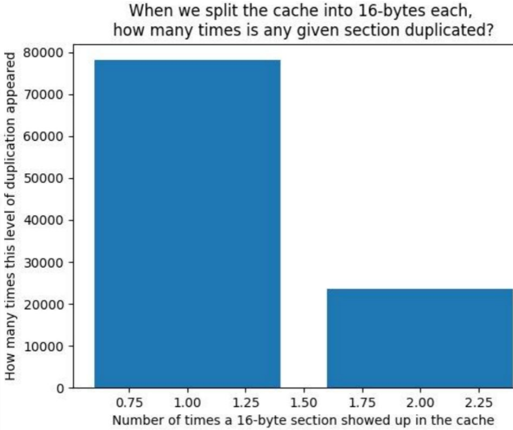


Subsequent Duplication





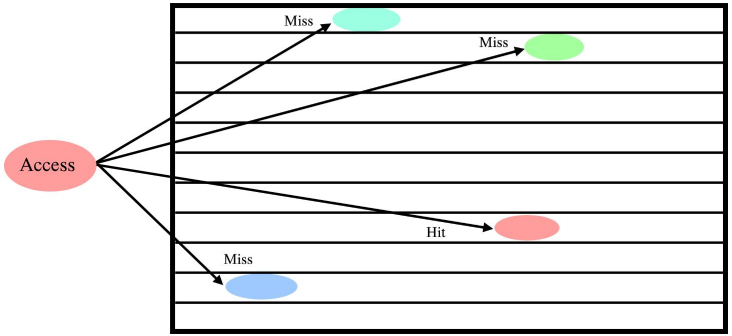




Duplication Patterns

- Invert element-ID function
    - Normal: element_id = F(address)
    - Inverse: address[] = F(element_id)

L2 Cache



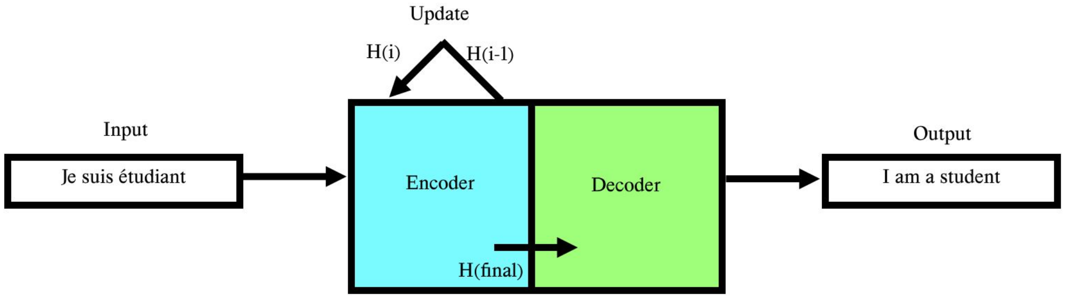

```c
 * Performs the inverse of the "g_workspace_indices" function. Given an
 * element_id pointing somewhere in the lowered matrix, find an address from
 * the lowered matrix (between g_B_start_ptr and g_B_end_ptr) that would map
 * to the same element_id if it were put through "g_workspace_indices" again.
 */
bool g_inverse_workspace(int element_id, new_addr_type** addrs, int* length) {
  // -1 is an indicator of a value in the lowered matrix that points to some
  // padded area, so it's not something we can return from this function.
  if (element_id == -1)
    return 0;

  // We can consider the 0 padding as a part of the matrix in this step. This
  // should be considered "padding" in the same way the 16-element-padding
  // is considered "padding"; thus, it should ideally have an element_id equal
  // to -1.
  int conv_input_rows = g_conv_input_rows + g_conv_padding_rows * 2;
  int conv_input_cols = g_conv_input_cols + g_conv_padding_cols * 2;

  // Before we do stuff specifically related to this function, obtain the
  // general info for workspace calculations. This part is the same as the
  // main "g_workspace_indices" function.
  // How many elements are in one dot product.
  int dprod_size = g_conv_filter_rows * g_conv_filter_cols *
    g_conv_input_channels;

  // How many dot products are in a row/column/filter.
  int dprods_per_row = (conv_input_rows - g_conv_filter_rows) /
    g_conv_stride_rows + 1;
  int dprods_per_col = (conv_input_cols - g_conv_filter_cols) /
    g_conv_stride_cols + 1;
  int num_dprods = dprods_per_row * dprods_per_col;

  // The array is padded by a multiple of 16, while the actual lowered matrix
  // is not. While we don't need to know if "element_id" points to somewhere
  // outside of the workspace (since we already know the element_id is not -1),
  // we still need to know the padding for address calculations.
  int pad_amount_row = (16 - dprod_size % 16) % 16;
  int row_size = dprod_size + pad_amount_row;
  int pad_amount_col = (16 - num_dprods % 16) % 16;
  int col_size = num_dprods + pad_amount_col;

  // Obtain the original components of the 3d vector that make up the position.
  // These three values denote a single location in the convolution tensor.
  int chaab  = element_id / (conv_input_rows * conv_input_cols);
  element_id = element_id % (conv_input_rows * conv_input_cols);
  int rowab  = element_id / conv_input_cols;
  element_id = element_id % conv_input_cols;
  int colab  = element_id;

  // There are multiple workspace rows/columns that will contain this position,
  // so locate the first one (the dot-product operation that would have
  // occurred first, or the one with the smallest row index). Each counter here
  // represents how many dot-products in the same row/column would need to
  // occur before we have the first dot-product that hits the element. For
  // example, if row_counter = 5 and col_counter = 7, then the dot product
  // happening on iteration (col_counter * dprods_per_row + row_counter)
  // would contain this element.
  int col_counter = max(
    colab - g_conv_filter_cols + g_conv_stride_cols /
    g_conv_stride_cols, 0);
  int row_counter = max(
    rowab - g_conv_filter_rows + g_conv_stride_rows /
    g_conv_stride_rows, 0);
  int dprod_counter = row_counter * dprods_per_row + col_counter;

  // We need to identify the position within the filter the first occurrence
  // of the element_id is within the dot products.
  int crel = colab - col_counter * g_conv_stride_cols;
  int rrel = rowab - row_counter * g_conv_stride_rows;

  // Now we can construct workspace rows/columns from these values. The first
  // workspace row is the dprod_counter, and the workspace_col is the flattened
  // relative position.
  int workspace_row_0 = dprod_counter;
  int workspace_col_0 = rrel * g_conv_filter_cols + crel;
```


```c
  // Now we can construct workspace rows/columns from these values. The first
  // workspace row is the dprod_counter, and the workspace_col is the flattened
  // relative position.
  int workspace_row_0 = dprod_counter;
  int workspace_col_0 = rrel * g_conv_filter_cols + crel;
```

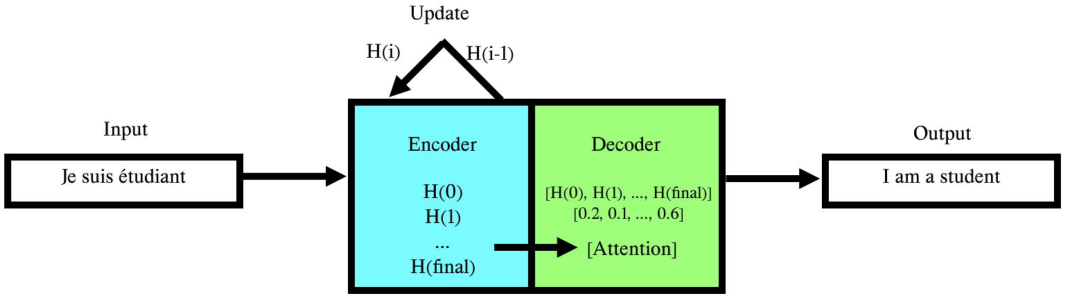

```c

  // Each dot-product is put into the lowered B matrix in order, so
  // dprod_counter denotes the index of the first dot-product occurring that
  // contains this element. We can calculate how many times dot-products will
  // contain this element in the same row, and again for the same column.
  int repeat_col = ceil((float)(crel + 1) / g_conv_stride_cols);
  int repeat_row = ceil((float)(rrel + 1) / g_conv_stride_rows);

  // Finally, the list of workspace rows/columns that contain this element is
  // equal to (repeat_row * repeat_col). We can iterate across each of them
  // and convert their workspace positions into real byte positions easily.
  *length = repeat_row * repeat_col;
  *addrs = (new_addr_type*)malloc(sizeof(int) * *length);
  int workspace_row_current = workspace_row_0;
  int workspace_col_current = workspace_col_0;
  for (int r = 0; r < row_counter; r++) {
    for (int c = 0; c < col_counter; c++) {
      // The variables "workspace_{row,col}_current" map to a position in the
      // workspace that contains the same element_id as the parameter to this
      // function. First, add back in the 16-element padding that got added
      // to each dprod to make it a multiple of 16. Then, turn the idx (points
      // to elements) into a byte offset (each element is a half, so two bytes)
      // and add it to the base address (start of the B matrix). The result is
      // the final address.
      int widx = workspace_row_current * dprod_size + workspace_col_current;
      int idx = widx + workspace_row_current * pad_amount_row;
      new_addr_type byte_offset = idx * 2;
      new_addr_type addr = byte_offset + g_B_start_ptr;
      (*addrs)[r * col_counter + c] = addr;

      // Within a row (or, across columns), move the filter to the right by the
      // stride. This makes the workspace_col decrease by the stride. The
      // next dot-product will contain this element_id still, so increase the
      // workspace_row by 1.
      workspace_col_current -= g_conv_stride_cols;
      workspace_row_current++;
    }

    // Once we've reached the end of the dot-products in this row that contain
    // the element, we move on to the next row. The number of dot-products we
    // pass between these points is equal to (dprods_per_row - repeat_row).
    // The new position within the filter is equal to the original (first)
    // filter position moved upwards by the stride amount.
    workspace_row_current += dprods_per_row - repeat_row;
    workspace_col_current = workspace_col_0 -
      g_conv_stride_rows * g_conv_filter_cols * (1 + row_counter);
  }

  return 0;
}
```

Large Language Models

- Previous models lost long-term information



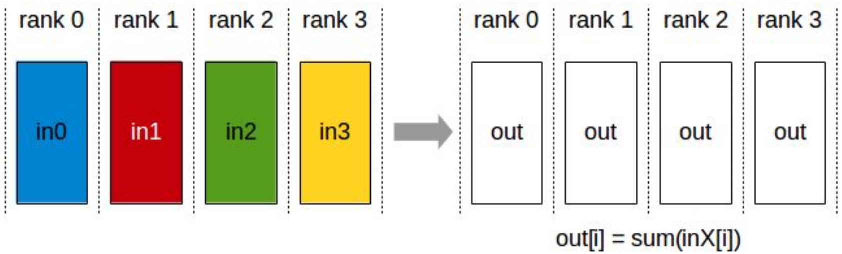
- New “attention” mechanism keeps all intermediate states, less likely to lose long-term information



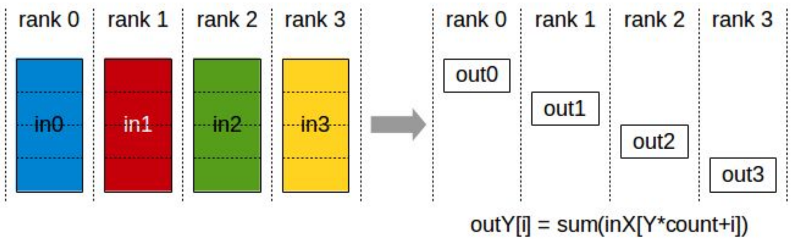
Interconnects

- NVLink: 1.8TB/s bidirectional, direct GPU-to-GPU interconnect
- Allows implementation of communication collectives

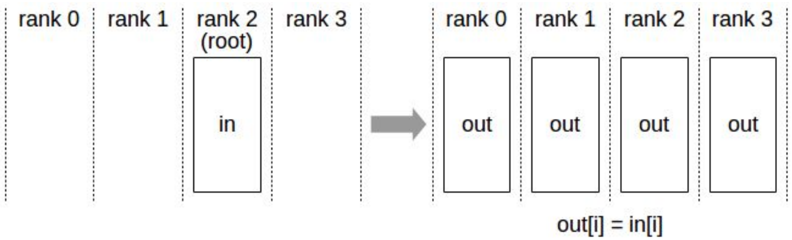
AllReduce



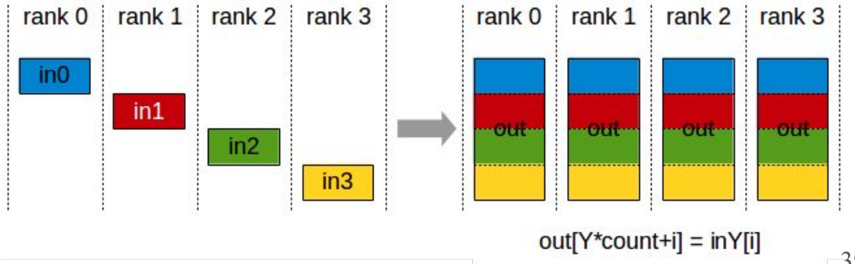
ReduceScatter



Broadcast

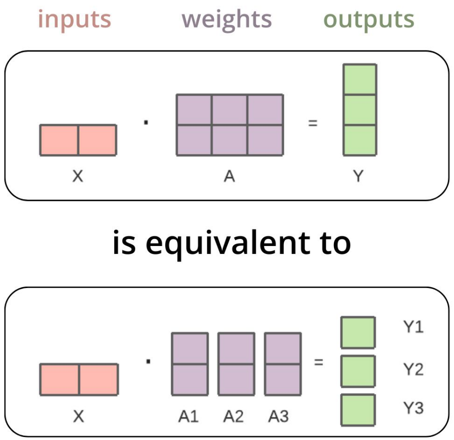


AllGather



Large Language Models

- Megatron-LM
  - 462 billion parameters
  - Trained on 6144 H100 GPUs
- Parallelism modes
  - Data parallel: split batch
  - Tensor parallel: split weights
  - Pipeline parallel: split layers



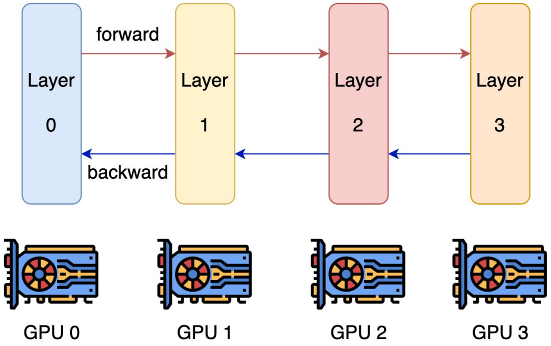
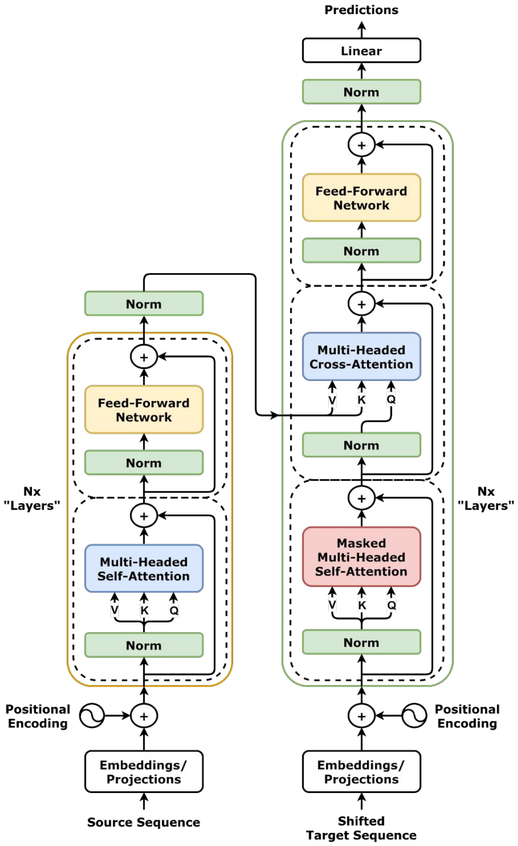
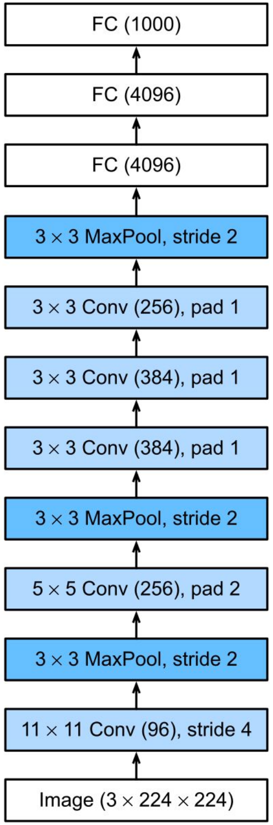


Image Net vs. LLM Comparison

- Significantly more kernels than image networks
  - AlexNet
    - 62 million parameters
    - 11 layers: 3 fully-connected, 5 conv2d
  - BERT
    - 110 million parameters
      - 12 encoders
      - 12 attention heads
      - 64 training examples / 2 epochs
    - 274,825 GEMM kernels (41.6 seconds)
    - 638,408 non-GEMM kernels (31.0 seconds)
      - E.g., data movement, elementwise
      - Time includes profiler





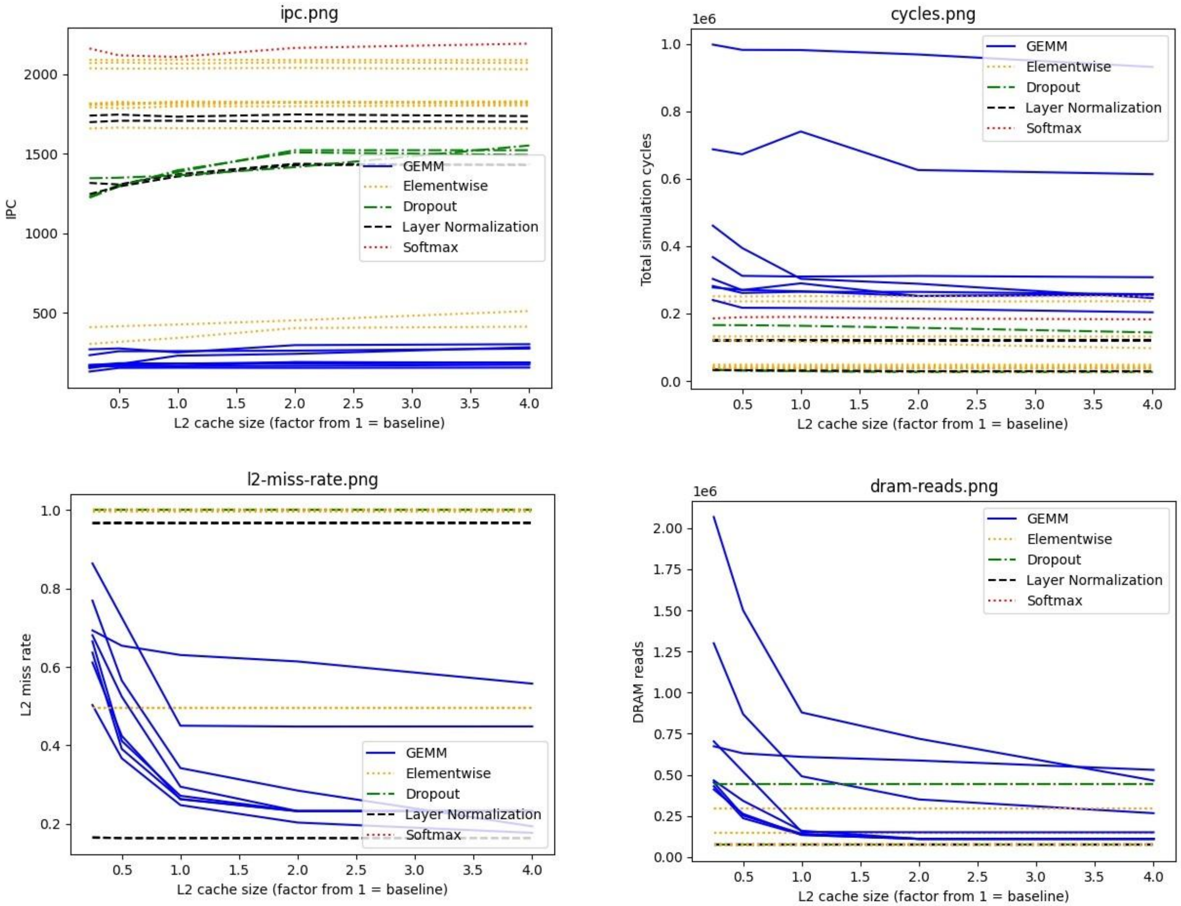
BERT Characteristics

1. GEMM
2. Elementwise
3. Dropout
4. Elementwise
5. Layer Normalization
6. Layer Normalization
7. GEMM
8. Elementwise
9. Elementwise
10. GEMM
11. Elementwise
12. Dropout
13. Elementwise
14. Layer Normalization
15. Layer Normalization
16. GEMM
17. Elementwise
18. GEMM
19. Elementwise
20. GEMM
21. Elementwise
22. GEMM
23. Elementwise
24. Elementwise
25. Softmax
26. Dropout
27. GEMM
28. Elementwise

- Repeats 12 times
- Memory access overlap:
  - GEMM/Elementwise: 39%
  - Elementwise/Dropout: 40%
  - Dropout/Elementwise: 57%
- Average repeat accesses:
  - GEMM: 6.2
  - Elementwise: 1.3
  - Dropout: 1.0
- Average total access count:
  - GEMM: 48,000
  - Elementwise: 6,100
  - Dropout: 7600
- Average unique access count:
  - GEMM: 7600
  - Elementwise: 3000
  - Dropout: 7600

L2 Size

- Does not improve with increasing L2 size
  - Similar to image nets



Conclusion

- Deep learning requires fast hardware and software
    - The unique properties of deep learning allows iterative compilation
    - Compute-heavy operations are good for GPUs
    - Communication libraries make distributed computing across GPUs easy