Glow

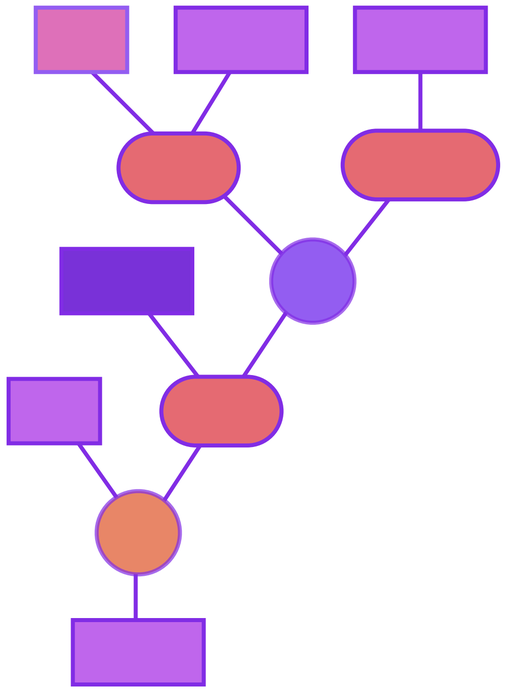
Graph Lowering Compiler for Hardware Accelerators

Nadav Rotem
FACEBOOK

Roman Levenstein
FACEBOOK

PyTorch

Neural Networks



A NEURAL NETWORK

″Where is my cat?″ ⟷ ″Где моя кошка?″

″Thumbs up!!!11 :)″ ⟶ fake account!

POPULAR APPLICATIONS

PyTorch

A machine learning framework that provides a seamless path from research to production.

PyTorchVision

PyTorchReasoning

PyTorchLanguage

PyTorchSpeech

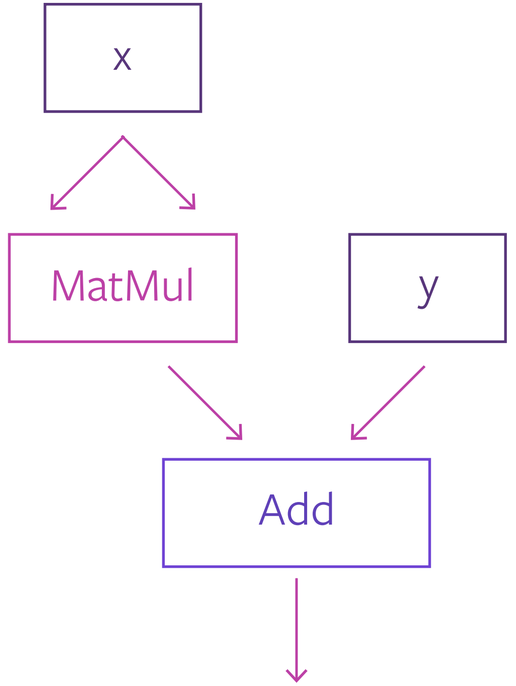
PyTorch Tools

PyTorch

DEFINE-BY-RUN

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t

x = torch.Tensor([[1,2],[3,5]])
y = torch.Tensor([[3,7],[1,2]])
foo(x, y)
```



ML at scale

- Facebook has billions of active users.
- Many services at Facebook use AI.
- NNs require lots of compute power.
- CPUs and GPUs are not efficient.

CPUs and GPUs are inefficient

- CPUs and GPUs work hard to extract parallelism.
- Matrix operations are very regular and expose lots of parallelism. Easy to accelerate.
- No need to waste power/area on useless features.

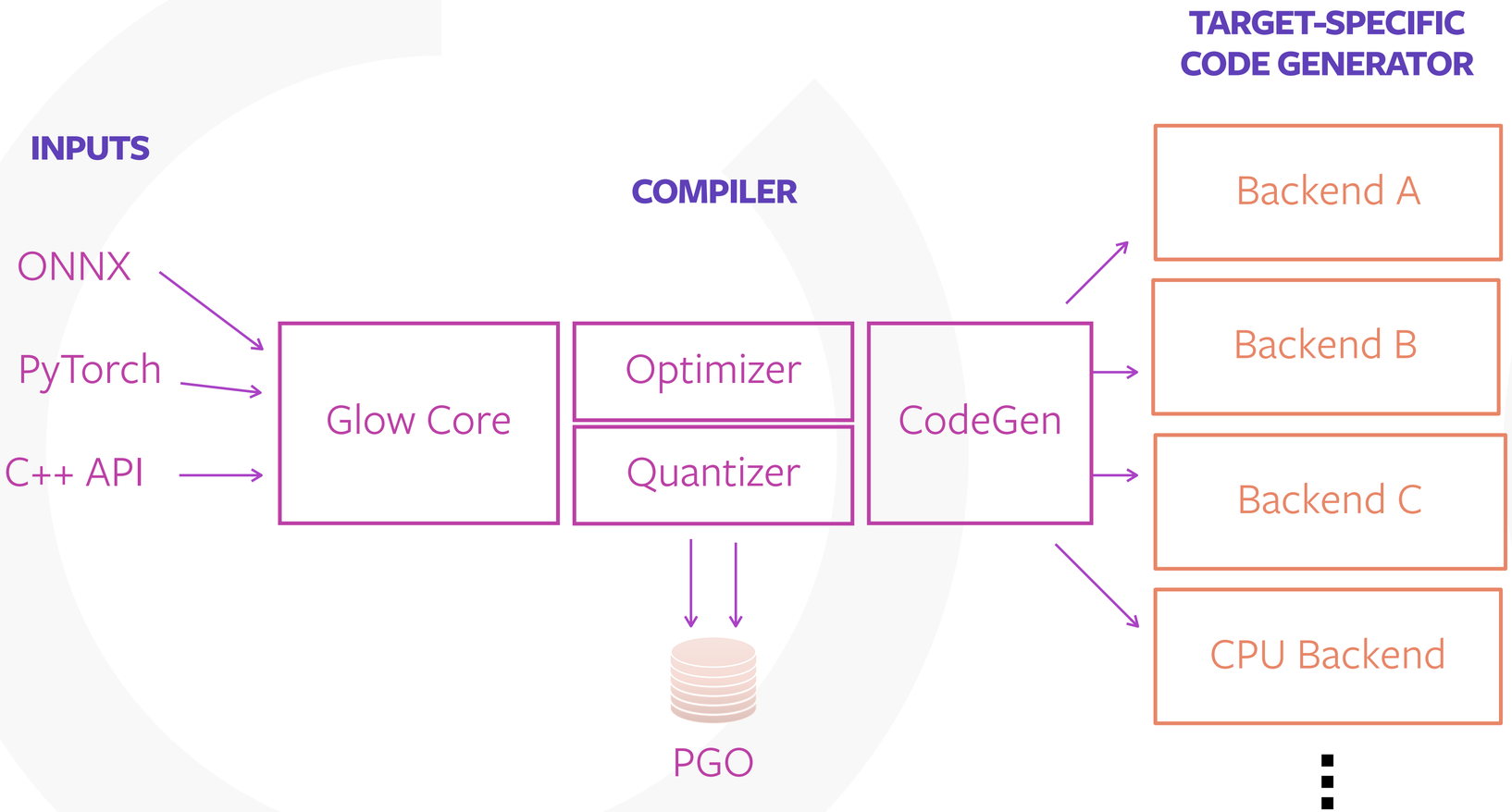
Accelerators are efficient because they are specialized

- Have many arithmetic execution units.
- Use dedicated local memories.
- Reduce the arithmetic bit-widths.
- Use a specialized programming model.

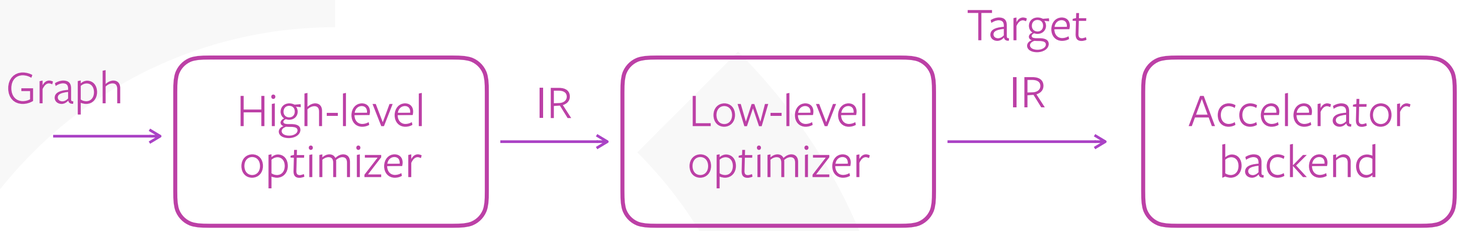
***Computer Architecture: A Quantitative Approach. [Hennessy, Patterson]. Chapter 7.**

Facebook is building ML hardware acceleration ecosystem with partners using the Glow compiler.

Glow Compiler Design



Compilation pipeline



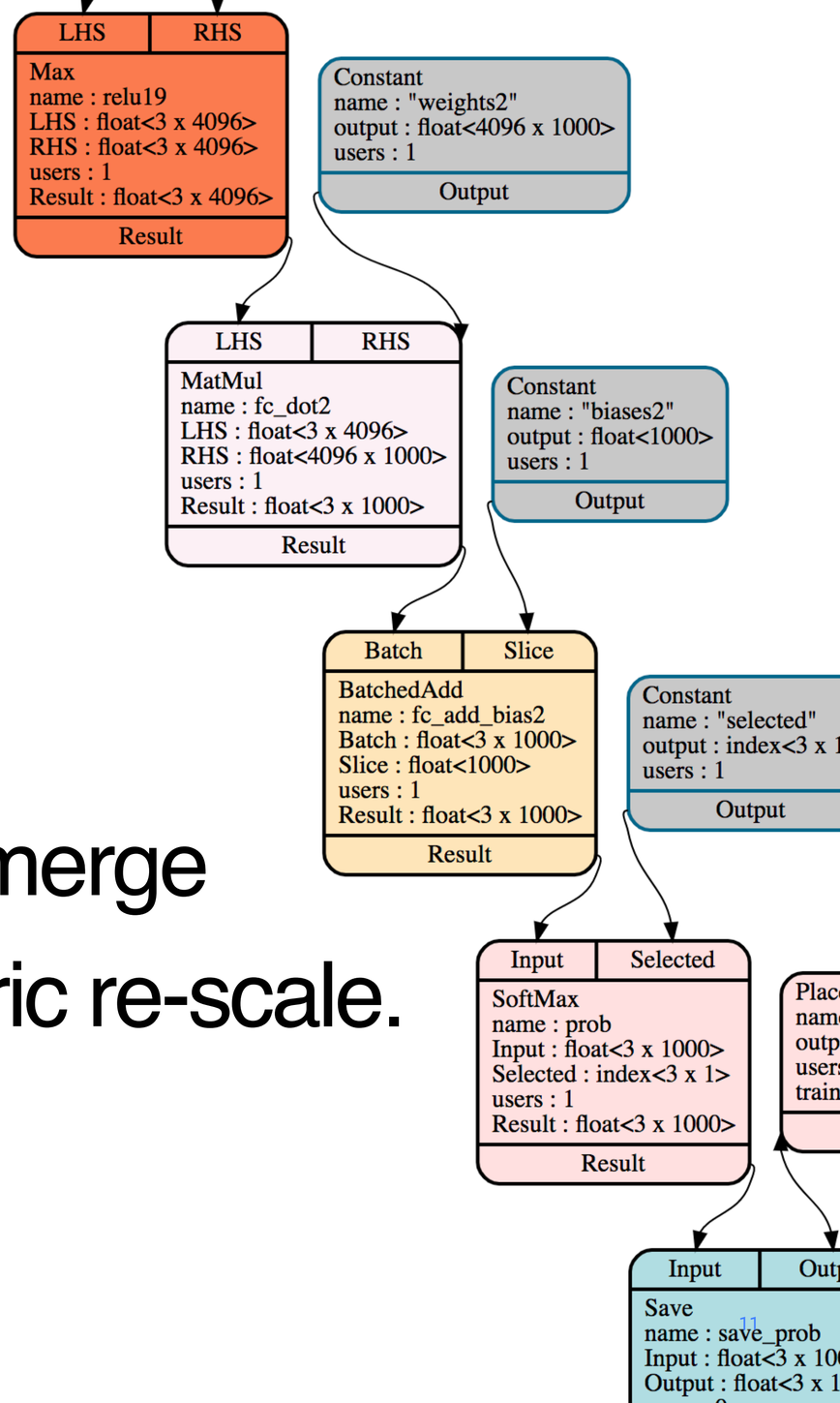
Performs high-level graph optimizations. Focus on linear-algebra kind of optimizations.

Performs low-level IR optimizations. Focus on buffer and memory reuse optimizations.

Performs target-specific lowering and optimizations for specific accelerator.

High-Level Graph

- Static-shaped data-flow graph.
- Enables high-level domain-specific optimizations.
- Example: Change the matrix layout, merge batchnorm into conv, eliminate numeric re-scale.



Low-Level Instructions

- Linear instruction-based address-only representation.
- Operands are typed pointers to buffers.
- Memory optimizations:

  Instruction scheduling,

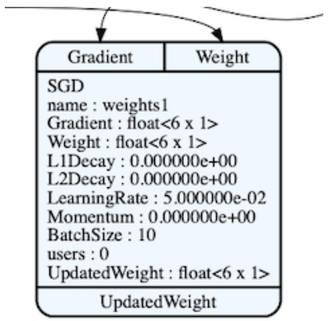
  Buffer sharing,

  shortening buffer lifetime.

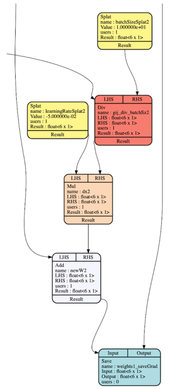
```
declare {
  %input = WeightVar float<10 x 5000> mutable
  %rnn_initial_state = WeightVar float<10 x 20> mutab
  %rnn_Whh = WeightVar float<20 x 20> mutable
  %result = WeightVar float<100 x 500> mutable
}
program {
  %X_0 = allocactivation  { Ty: float<10 x 500>}
  %X_01 = extracttensor @out %X_0, @in %input { Offse
  %mergeLHS11 = allocactivation  { Ty: float<100 x 50
  %mergeLHS111 = splat @out %mergeLHS11 { Value: 0.00
  %mergeLHS112 = inserttensor @inout %mergeLHS11, @in
  %bigMatMul1_res = allocactivation  { Ty: float<100
  %bigMatMul1 = matmul @out %bigMatMul1_res, @in %mer
  %dealloc10 = deallocactivation @out %mergeLHS11
  %mergedBA = batchedadd @out %bigMatMul1_res, @in %b
  %rnn_add_0_res = allocactivation  { Ty: float<10 x
  %fc_dot = matmul @out %rnn_add_0_res, @in %rnn_init
  %fc_add_bias = batchedadd @out %rnn_add_0_res, @in
```

Graph Lowering

- ML frameworks support hundreds of op kinds, implemented in C and CUDA.
- Writing hundreds of ops for accelerators isn't scalable.
- Glow lowers complex high-level nodes into primitive nodes.



BEFORE



AFTER

Automatic Node Generation

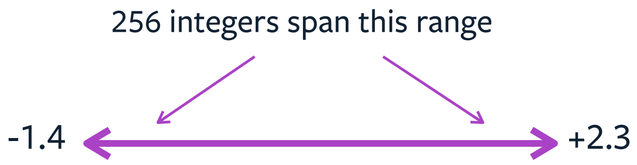
- Graph Nodes are classes with many methods: ctor, hash, set/get, compare, clone, print, etc.
- Instead of writing the methods in C++ we auto-generate them.

CONVOLUTION NODE DEFINITION

```
BB.newNode("Convolution")
    .addInput("Input")
    .addInput("Filter")
    .addInput("Bias")
    .addMember(MemberType::VectorUnsigned, "Kernels")
    .addMember(MemberType::VectorUnsigned, "Strides")
    .addMember(MemberType::VectorUnsigned, "Pads")
    .addMember(MemberType::Unsigned, "Group")
    .setDocstring("Performs Convolution using a given Input, Filter, and "
                  "Bias tensors, as well as provided Kernels, Strides, Pads, "
                  "and Group.");
```

Quantization

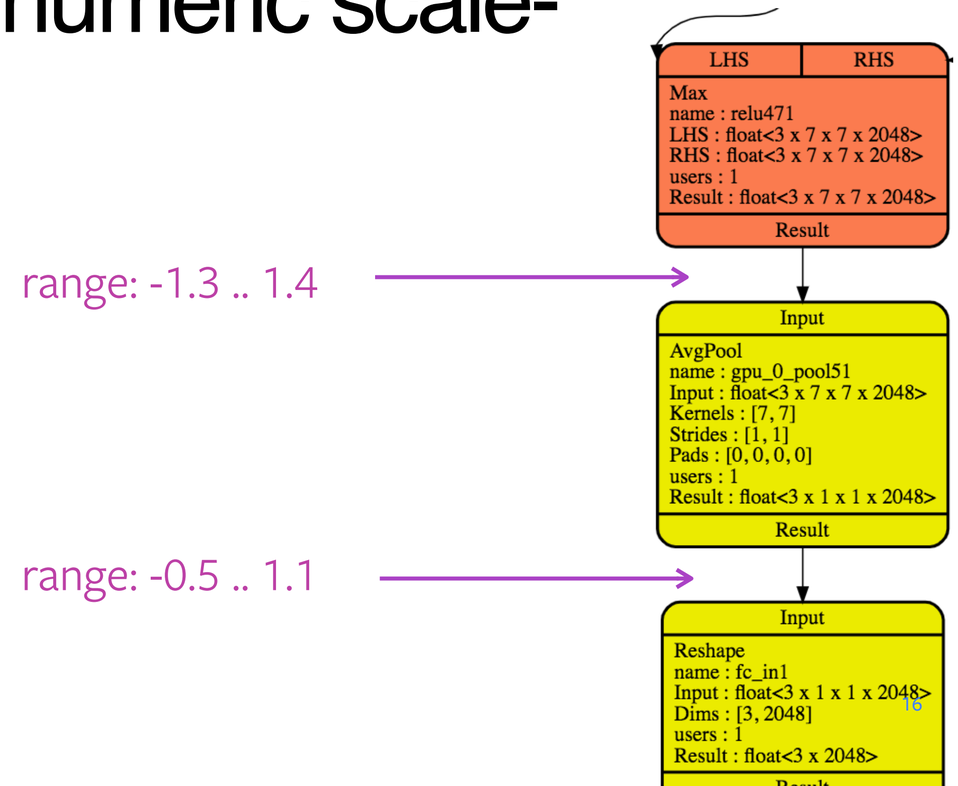
- Neural networks are resilient and can work with reduced bit-width (i8, fp16 instead of fp32).
- Quantization is the process of converting the network to integer arithmetic.
- Represent a range of real numbers using integers.



real = (integer - offset) * scale

Profile Guided Quantization

- Glow uses profile-guided quantization to estimate the range of each edge in the graph.
- Compiler optimizations eliminate numeric scale-conversion between nodes.



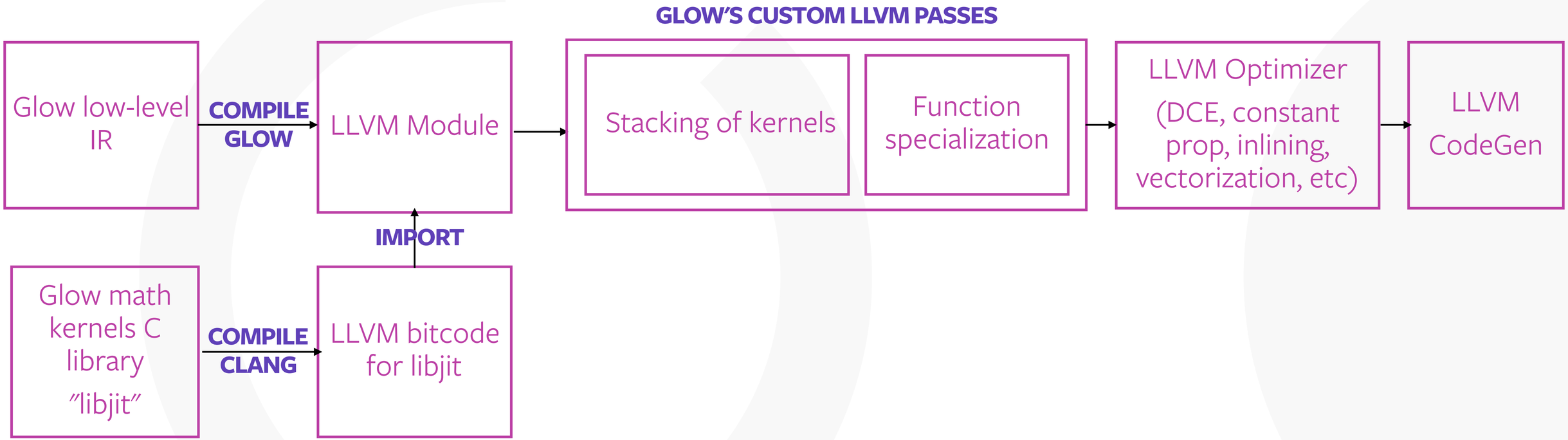
Glow CPU/JIT backend

- Significantly better performance vs an interpreter
  - Eliminates the dispatching overhead
  - Performs target-specific optimizations

- Just-in-time information allows for better compiler decisions & optimizations
  - Vendor libraries already JIT (CUDA, dnnMKL, libxssm), but one operator at a time
  - JITting the whole graph gives the compiler even more opportunities

- Ability to specialize based on the concrete NN model
  - Most shapes, types and sometimes addresses of memory buffers are constants at the time of JIT compilation
  - JIT can produce a tailor-made optimized code for the specific values/shapes of some/all parameters
  - Easy for a JIT, but not possible to achieve using any general-purpose libraries!

CPU/JIT design



Stacking of kernels

- Many tensor operations are element-wise, e.g. add, max, mul
  - There are often chains of such operations like sub(z, mul(x, y))
- Sequential execution of these operations traverses the whole tensor every time and trashes the cache
- Instead, Glow generates LLVM IR for a stacked kernel where all those operations are performed on each element of a tensor during a single tensor traversal

Function specialization

- Specializes library kernels for constant parameters
- Tensor shapes and many other parameters are compile-time constants
- Produces a cloned LLVM IR function where the values of constant parameters are substituted
- Specialization leads to much better performance

EXAMPLE OF AN OPERATION

```
int argmax(float *arr, int n) {
  float maxVal = arr[0];
  int maxPos = 1;
  for (int i = 1; i < n; ++i) {
    if (arr[i] < maxVal) {
      maxVal = arr[i];
      maxPos = i;
    }
  }
  return maxPos;
}
```

UNSPECIALIZED CODE

```
argmax(float*, int): # @argmax(float*, int)
  mov eax, 1
  cmp esi, 2
  jl .LBB0_6
  vmovss xmm0, dword ptr [rdi] # xmm0 = mem[0],zero,zero,zero
  mov r8d, esi
  lea rax, [r8 - 2]
  add esi, 7
  and esi, 7
  cmp rax, 7
  jae .LBB0_7
  mov eax, 1
  mov ecx, 1
  test esi, esi
  jne .LBB0_4
  jmp .LBB0_6
.LBB0_7:
  sub r8, rsi
  mov eax, 1
  mov ecx, 1
.LBB0_8: # =>This Inner Loop Header: Depth=1
  vmovss xmm1, dword ptr [rdi + 4*rcx] # xmm1 = mem[0],zero,zero,zero
  vmovss xmm2, dword ptr [rdi + 4*rcx + 4] # xmm2 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  vminss xmm0, xmm1, xmm0
  cmova eax, ecx
  lea edx, [rcx + 1]
  vucomiss xmm0, xmm2
  cmovbe edx, eax
  vminss xmm0, xmm2, xmm0
  lea eax, [rcx + 2]
  vmovss xmm1, dword ptr [rdi + 4*rcx + 8] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmovbe eax, edx
  vminss xmm0, xmm1, xmm0
  lea edx, [rcx + 3]
  vmovss xmm1, dword ptr [rdi + 4*rcx + 12] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmovbe edx, eax
  vminss xmm0, xmm1, xmm0
  lea eax, [rcx + 4]
  vmovss xmm1, dword ptr [rdi + 4*rcx + 16] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmovbe eax, edx
  vminss xmm0, xmm1, xmm0
  lea edx, [rcx + 5]
  vmovss xmm1, dword ptr [rdi + 4*rcx + 20] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmovbe edx, eax
  vminss xmm0, xmm1, xmm0
  lea r9d, [rcx + 6]
  vmovss xmm1, dword ptr [rdi + 4*rcx + 24] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmovbe r9d, edx
  vminss xmm0, xmm1, xmm0
  lea eax, [rcx + 7]
  vmovss xmm1, dword ptr [rdi + 4*rcx + 28] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmovbe eax, r9d
  vminss xmm0, xmm1, xmm0
  add rcx, 8
  cmp r8, rcx
  jne .LBB0_8
  test esi, esi
  je .LBB0_6
.LBB0_4:
  neg rsi
.LBB0_5: # =>This Inner Loop Header: Depth=1
  vmovss xmm1, dword ptr [rdi + 4*rcx] # xmm1 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  cmova eax, ecx
  vminss xmm0, xmm1, xmm0
  add rcx, 1
```

SPECIALIZE FOR N == 256

SPECIALIZED CODE

```
argmax(float*, int): # @argmax(float*, int)
  vmovss xmm0, dword ptr [rdi] # xmm0 = mem[0],zero,zero,zero
  mov eax, 1
  mov ecx, 1
.LBB0_1: # =>This Inner Loop Header: Depth=1
  vmovss xmm1, dword ptr [rdi + 4*rcx] # xmm1 = mem[0],zero,zero,zero
  vmovss xmm2, dword ptr [rdi + 4*rcx + 4] # xmm2 = mem[0],zero,zero,zero
  vucomiss xmm0, xmm1
  vminss xmm0, xmm1, xmm0
  cmova eax, ecx
  lea edx, [rcx + 1]
  vucomiss xmm0, xmm2
  cmovbe edx, eax
  vminss xmm0, xmm2, xmm0
  vmovss xmm1, dword ptr [rdi + 4*rcx + 8] # xmm1 = mem[0],zero,zero,zero
  lea eax, [rcx + 2]
  vucomiss xmm0, xmm1
  cmovbe eax, edx
  vminss xmm0, xmm1, xmm0
  vmovss xmm1, dword ptr [rdi + 4*rcx + 12] # xmm1 = mem[0],zero,zero,zero
  lea edx, [rcx + 3]
  vucomiss xmm0, xmm1
  cmovbe edx, eax
  vminss xmm0, xmm1, xmm0
  vmovss xmm1, dword ptr [rdi + 4*rcx + 16] # xmm1 = mem[0],zero,zero,zero
  lea eax, [rcx + 4]
  vucomiss xmm0, xmm1
  cmovbe eax, edx
  vminss xmm0, xmm1, xmm0
  add rcx, 5
  cmp rcx, 256
  jne .LBB0_1
  ret
```

- ✓ Runtime checks are eliminated
- ✓ Control flow is simplified
- ✓ Smaller code
- ✓ Faster execution

AOT and debugging support

- Ahead-of-time (AOT) compilation
  - Save the LLVM generated machine code for a NN model as a self-contained object file
  - Interesting e.g. for deployments on mobile and memory-constrained devices

- Debugging support
  - Glow emits LLVM debug information for NN models
  - Debugging is done in terms of Glow IR instead of machine code

AN EXAMPLE OF A DEBUGGING SESSION USING LLDB

```
Process 95176 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason
    frame #0: 0x000000010000015f8 resnet50`resnet50 [inline
   282    162 %gpu_0/res4_4_branch2a__1024 = cpuconvdkkc8
 Kernel: 1, Stride: 1, Pad: 0, Depth: 256}
   283    163 %relu__1089 = cpumaxsplat @out %gpu_0_res4_4
   284    164 %gpu_0_res4_4_branch2b__1026_res = allocacti
-> 285    165 %gpu_0/res4_4_branch2b__1026 = cpuconvdkkc8
ernel: 3, Stride: 1, Pad: 1, Depth: 256}
   286    166 %dealloc37 = deallocactivation @out %gpu_0_r
   287    167 %relu__1090 = cpumaxsplat @out %gpu_0_res4_4
   288    168 %gpu_0_res4_4_branch2c_bn__786_res = allocac
, @in 173, @out 171
Target 0: (resnet50) stopped.
(lldb) p &gpu_0_res4_4_branch2a__1024_res
(float (*)[1][14][14][256]) $0 = 0x0000000100010060
(lldb) p &conv_filter__1025
(float (*)[32][3][3][256][8]) $1 = 0x00000001020f4210
(lldb) p conv_filter__1025[1][2][2][100][5]
(float) $2 = 0.15751867
(lldb) p conv_filter__1025[1][2][2][100]
(float [8]) $3 = ([0] = 0.032105349, [1] = -0.0109192021,
(lldb)
```

Memory management for HW accelerators

- Accelerators have many processing elements (PEs)
- Usually no caches, no out-of-order execution
- Accelerators have multiple memory banks with different properties in terms of size and access speed: DRAM, SRAM, scratchpads, etc
  - Memory in all memory banks needs to be managed explicitly
  - Data transfers between some memory banks is possible only by means of explicit DMA commands
- Instructions may have requirements on the memory banks to be used for their operands and on the memory layout of their operands

Static memory allocation

- Memory can’t be ‘malloc’-ed on the HW accelerator
- Glow compiler has to manage and allocate the on-device memory statically
  - Allocation is performed for each memory bank
- Live buffers are allocated and freed.
- Scheduler and IR optimizer reduce memory pressure and shorten buffers lifetime.

Memory management strategy

- Ensure that buffer operands are loaded into the required memory banks, usually into fast scratchpads
- Try to keep data in fast memory banks as long as possible
- Evict data from fast memory banks only if it cannot be avoided
  - Often involves explicit DMA transfers
- Minimize the cost of evictions, i.e. slow data transfers between memory banks

Sounds familiar???

Yes, it is rather similar to **register allocation**!!!

- The analogy is: buffers == virtual registers, fast memory banks == physical registers, eviction from fast memory banks == register spilling
- But there are differences:
  - Buffers have different varying sizes
  - Evicting buffers from fast memory banks is expensive, often involves DMA data transfers
    - Cost of eviction is proportional to the amount of data to be transferred!

How to achieve the best performance?

- Keep the processing elements always busy
- Hide latency of memory accesses
  - Use pre-fetching
  - Intermix data-fetching and computation
- Partition data to fit into accelerator's memory banks and process it in parallel
  - e.g. scatter/gather approach
- Reduce the amount of data transfers
  - Between accelerator RAM and fast memory banks
  - Between the host and accelerator RAM
- Use a good scheduling algorithm

EXAMPLE: POSSIBLE CODE TO PERFORM CONVOLUTION

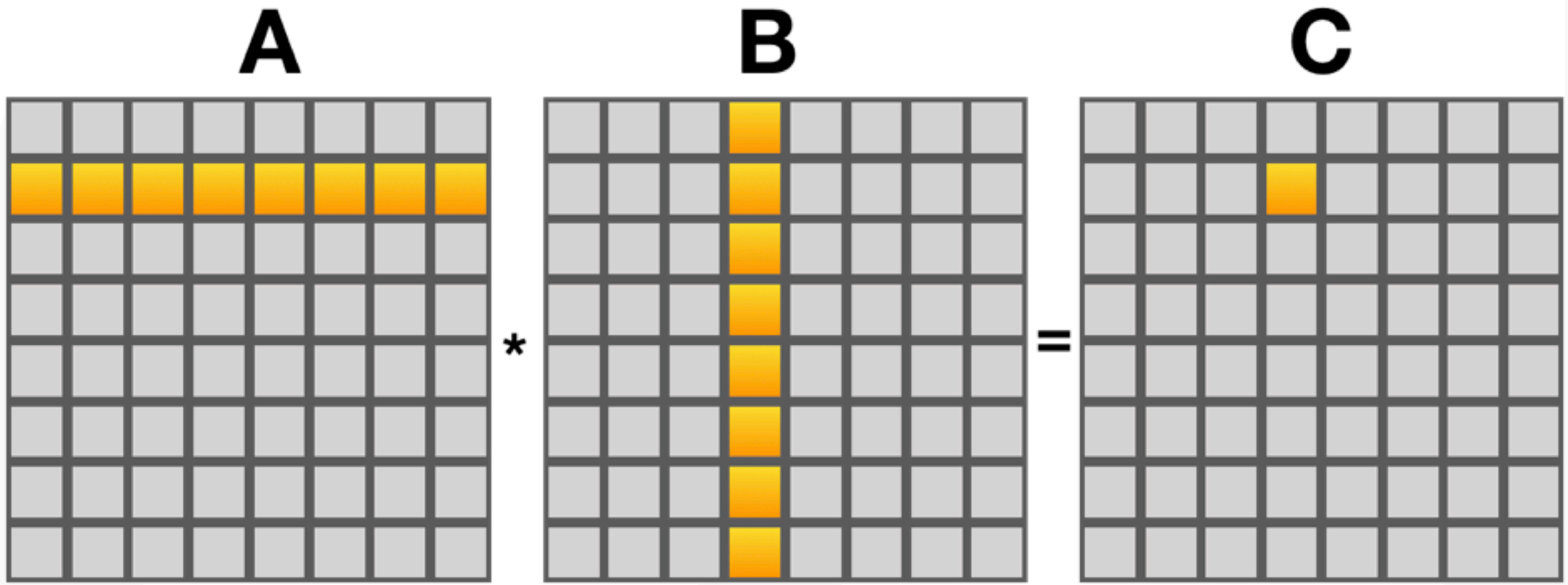
```
Set DMA mode to stride [0x0, 0x0, 0x400, 0x400, 0x0, 0x0]
Start DMA request for block #1 into SRAM address 0xA000
Start DMA request for block #2 into SRAM address 0x0B800
Start DMA request for block #3 into SRAM address 0xFF0000
Configure the state of activation unit to 'RELU'
Wait for #1 and #2
Start Matrix Multiplication on #1 and #2 to SRAM 0xD0000
Wait for #3
Start Matrix Multiplication on #1 and #3 to SRAM 0xD0400
Start DMA request for block #4 into address SRAM 0xFF0000
Wait for #4
Start Matrix Multiplication on #1 and #4 to SRAM 0xA0400
Configure the state of activation unit to lookup table from
address SRAM 0xB0400
```

Matrix Multiplication

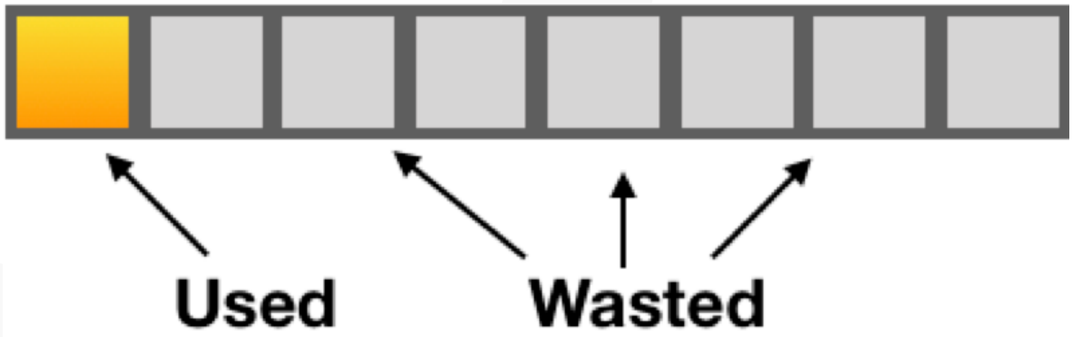
```
C(m, n) = A(m, k) * B(k, n)
```

```
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    for (int p = 0; p < k; p++) {
      C(i, j) += A(i, p) * B(p, j);
    }
  }
}
```

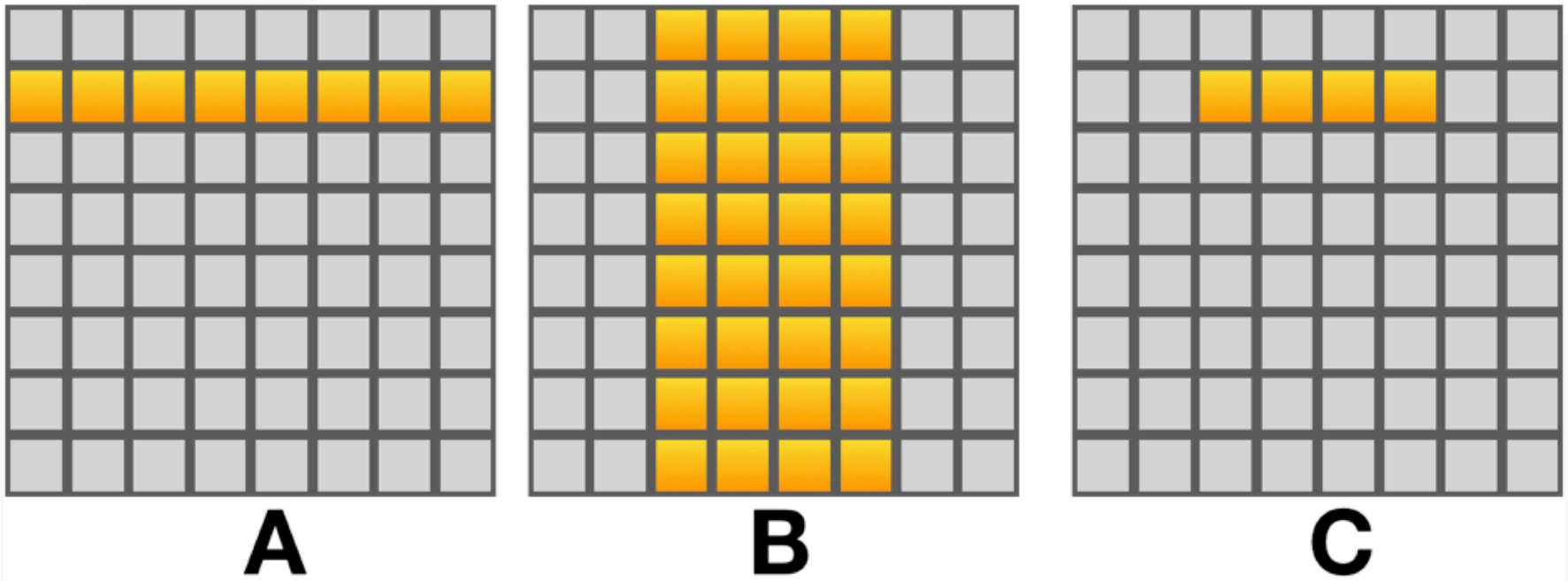
Matrix Multiplication



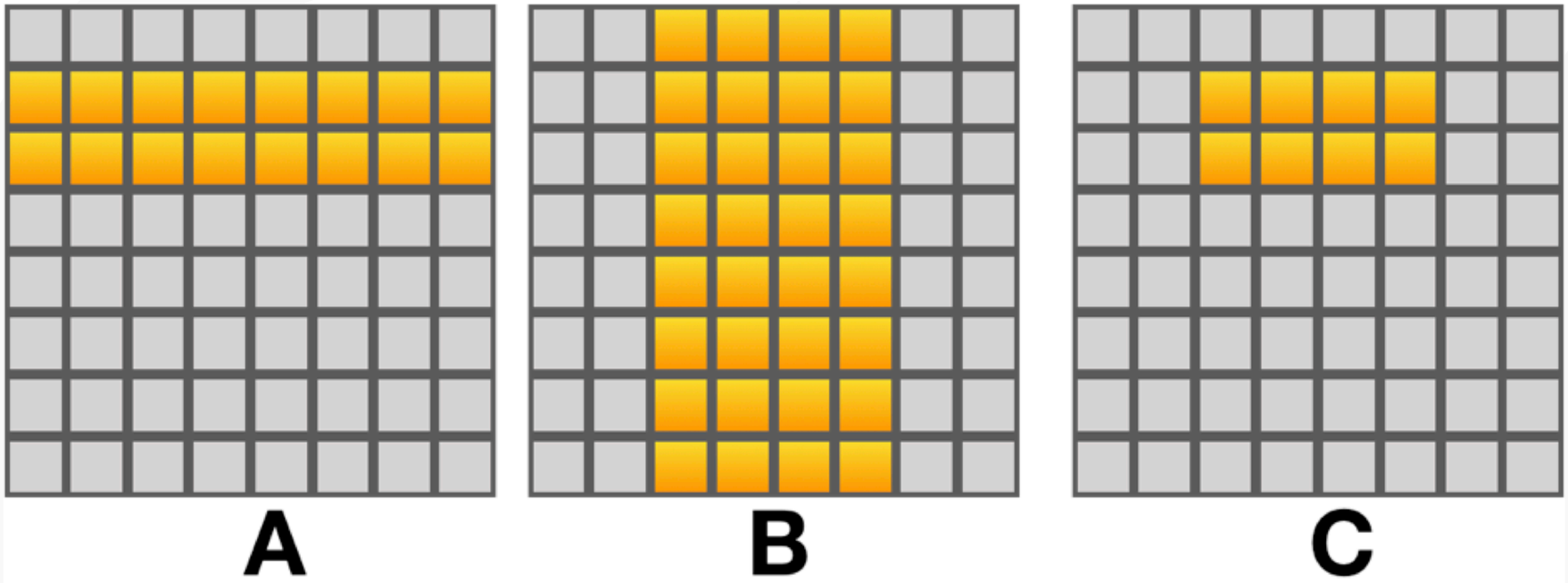
Parallelism is the key to performance



Vectorization



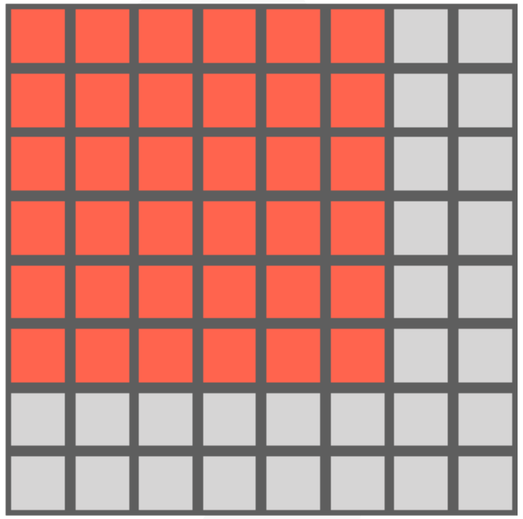
Register Blocking

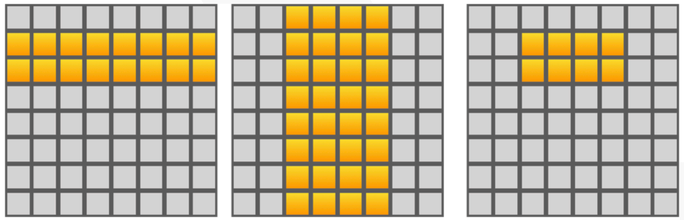


Register Blocking

Regs A	Regs B	Registers Used	Memory Ops	Arithmetic Ops
1	4	5	5	4
2	4	10	6	8
2	5	12	7	10
4	3	16	7	12
1	8	9	9	8
3	4	15	7	12

Tiling





Glow

- Glow is a machine learning compiler for accelerators.
- Facebook is working with hardware partners on accelerating AI.
- We rely on LLVM in many parts of the compiler.
- Lot's of hard problems to solve. Facebook is hiring. ;)

Participate on GitHub

Glow: Compiler for Neural Network Hardware Accelerators

https://github.com/pytorch/glow

arXiv publication

Glow: Graph Lowering Compiler Techniques for Neural Networks

https://arxiv.org/abs/1805.00907

@Scale 2018 Keynote

Glow: A community-driven approach to AI

https://atscaleconference.com/videos/scale-2018-keynote-glow-a-community-driven-approach-to-ai/

Thank you