Manual versus Automatic Generation of Computational Kernels for Deep Learning Inference

Guillermo Alaejos

Adrián Castelló

Pedro Alonso

Enrique S. Quintana-Ortí

Francisco D. Igual

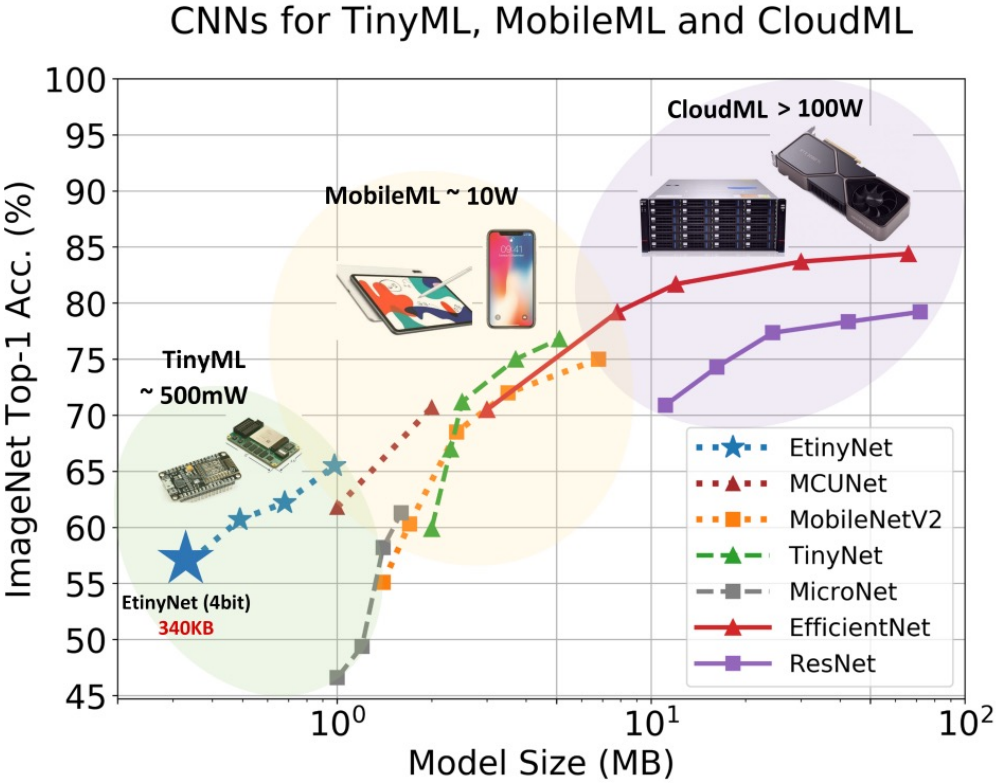
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSIDAD COMPLUTENSE MADRID

Motivation

Deep learning (DL) is key for edge/IoT applications:

- Security & privacy
- Latency
- Energy consumption and/or power constraints



Motivation

Convolutional neural networks (CNNs) are key for computer vision & signal processing

Convolution operator take up to 90-95% of the execution time

Four major approaches for the convolution:

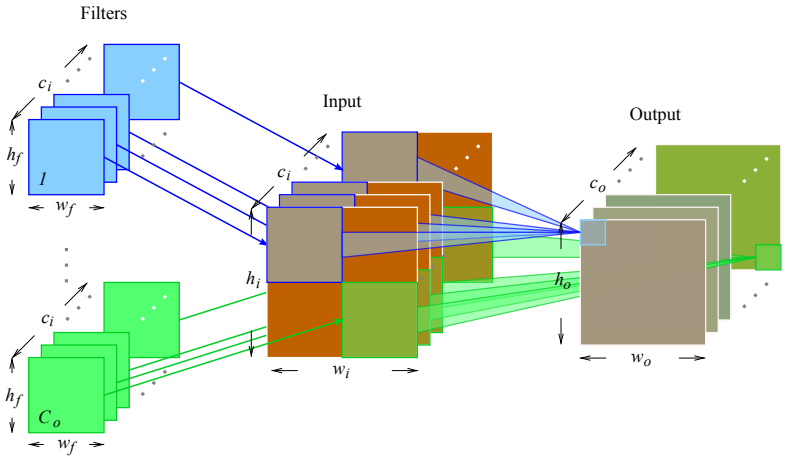
- Lowering: im2col/im2row + large GEMM (matrix multiplication)
- Direct convolution: blocked as a collection of small GEMMs
- Winograd: 1/3 of operations correspond to GEMM
- FFT

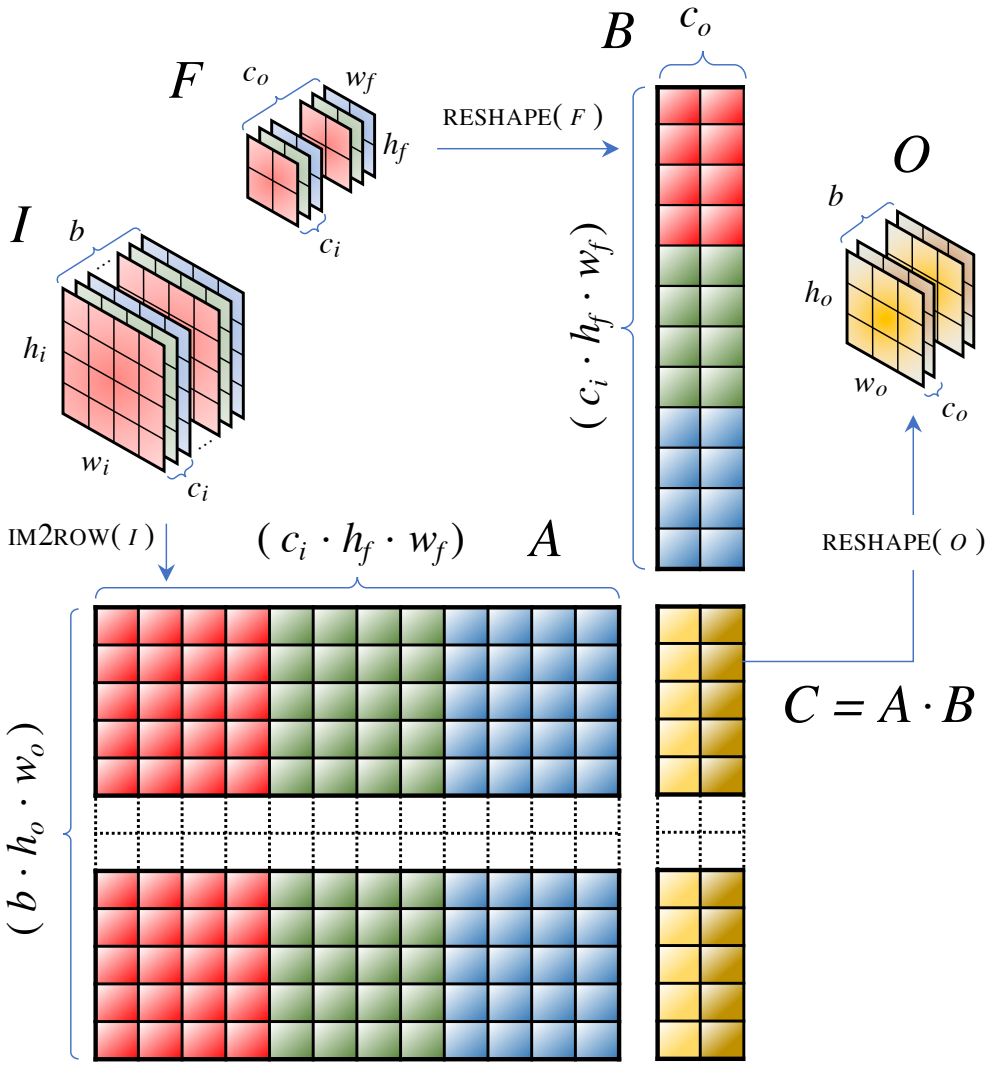
Motivation

Lowering is by far the most flexible technique

Also highly parallel (thread-level parallelism)

$$O = \text{CONV}(F, I)$$





Motivation

IM2ROW transform = Data copies and replication

Libraries for GEMM → Convolution?

- GotoBLAS2, OpenBLAS, BLIS, oneAPI, AMD AOCL, Intel MKL, ARMPL,…

- Large memory footprint (for IoT)
- Missing functionality (INT8, INT16, FP16, BF16, TF32)
- Suboptimal performance (later)
- No support for operator fusion (deep learning)

Motivation

IM2ROW transform = Data copies and replication

Libraries for GEMM → Convolution?

Hardware-specific solution in a heterogenous world!

- Intel/AMD AVX; Intel AVX-512
- ARM NEON, SVE
- RISC-V “V”
- Microcontroller units (MCUs)...

Motivation

IM2ROW transform = Data copies and replication

Libraries for GEMM → Convolution?

Hardware-specific solution in a heterogenous world!

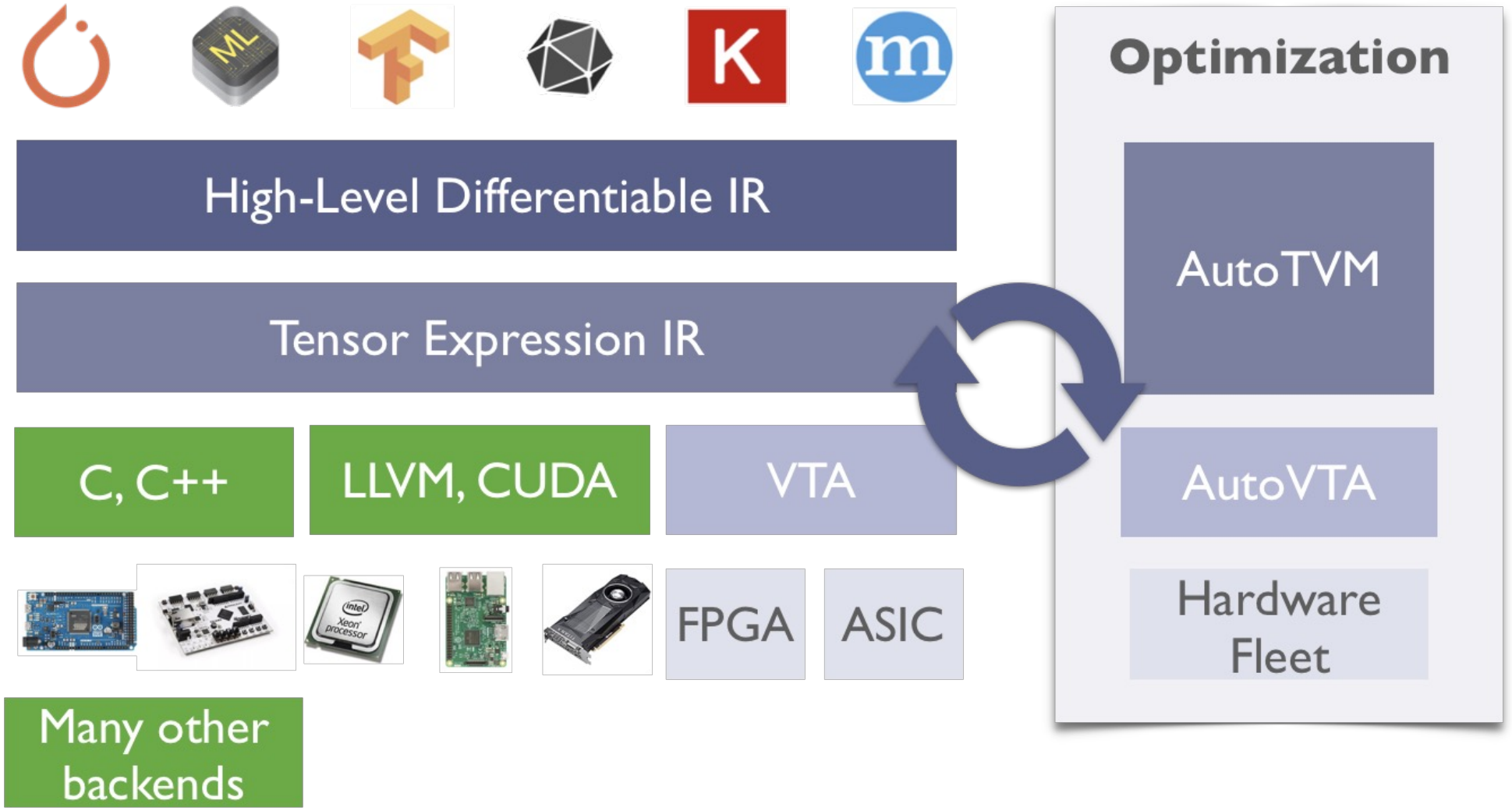
- Intel/AMD AVX; Intel AVX-512
- ARM NEON, SVE
- RISC-V “V”
- Microcontroller units (MCUs)...

→ Automatic generation… but guided by experience!

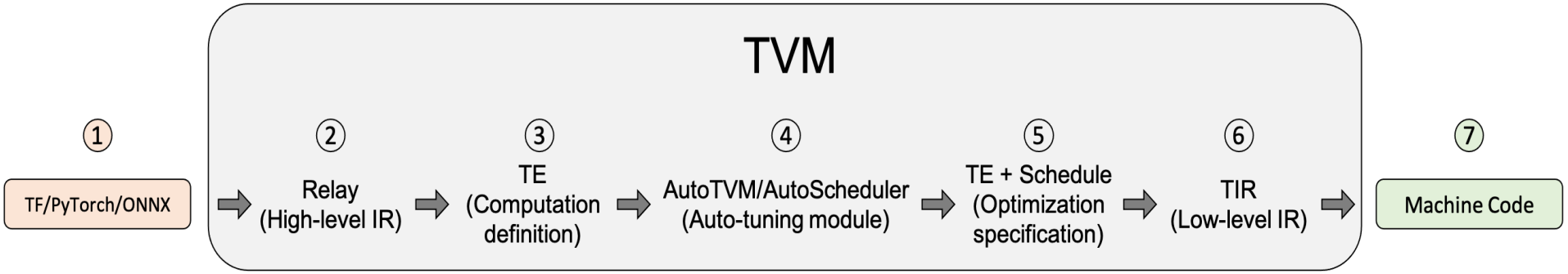
Outline

- Basic GEMM with Apache TVM
- Guided high performance GEMM with Apache TVM
- Experiments on portability
- Conclusions

Apache TVM


ML
K
m
High-Level Differentiable IR
Tensor Expression IR
C, C++
LLVM, CUDA
VTA
FPGA
ASIC
Many other backends
Optimization
AutoTVM
AutoVTA
Hardware Fleet

Apache TVM




```
tvmc compile --target "llvm" --output
resnet50-tvm.tar resnet50.onnx →
```

- mod.so: model as a C++ library to be be loaded by the TVM runtime
- mod.json: representation of the TVM Relay computation graph
- mod.params: file containing the parameters for the pre-trained model

```
tvmc run --inputs imagenet.npz --output
predictions.npz resnet50-tvm.tar
```

Basic GEMM with TVM

$$C = A \cdot B$$

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

Placeholders:

$A \rightarrow m\ x\ k$

$B \rightarrow k\ x\ n$

```
1  def basic_GEMM(m, n, k):
2    # B1) Define operation
3    A = te.placeholder((m, k), name="A")
4    B = te.placeholder((k, n), name="B")
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Basic GEMM with TVM

$$C = A \cdot B$$

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

Define result:

$C \rightarrow m\ x\ n$

Reduce over axis p

```
1  def basic_GEMM(m, n, k):
2    # B1) Define operation
3    A = te.placeholder((m, k), name="A")
4    B = te.placeholder((k, n), name="B")
5    p = te.reduce_axis((0, k), "p")
6    C = te.compute((m, n), lambda i, j:
7                   te.sum(A[i, p] * B[p, j],
8                   axis=p), name="C")
9
10
11
12
13
14
15
16
17
18
19
20
```

Basic GEMM with TVM

$$C = A \cdot B$$

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

Schedule (loop order):

$$i \rightarrow j \rightarrow k$$

```
1  def basic_GEMM(m, n, k):
2    # B1) Define operation
3    A = te.placeholder((m, k), name="A")
4    B = te.placeholder((k, n), name="B")
5    p = te.reduce_axis((0, k), "p")
6    C = te.compute((m, n), lambda i, j:
7                   te.sum(A[i, p] * B[p, j],
8                   axis=p), name="C")
9
10   # B2) Prepare schedule
11   sched  = te.create_schedule(C.op)
12   (i, j) = C.op.axis
13   (p,)   = C.op.reduce_axis
14
15   # B3) Loop schedule i->j->p
16   sched[C].reorder(i, j, p)
17
18
19
20
```

- Other variants (jik, ikj,…) easy to obtain

Basic GEMM with TVM

$$C = A \cdot B$$

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

Generate code for `llvm`

```
1  def basic_GEMM(m, n, k):
2    # B1) Define operation
3    A = te.placeholder((m, k), name="A")
4    B = te.placeholder((k, n), name="B")
5    p = te.reduce_axis((0, k), "p")
6    C = te.compute((m, n), lambda i, j:
7                   te.sum(A[i, p] * B[p, j],
8                   axis=p), name="C")
9
10   # B2) Prepare schedule
11   sched  = te.create_schedule(C.op)
12   (i, j) = C.op.axis
13   (p,)   = C.op.reduce_axis
14
15   # B3) Loop schedule i->j->p
16   sched[C].reorder(i, j, p)
17
18   # B4) Generate code with LLVM backend
19   return tvm.build(sched, [A, B, C],
20                    target="llvm")
```

- Independent of data type

Basic GEMM with TVM

$$C = A \cdot B$$

Even with AutoTVM, the search space for optimization is huge!

- Tiling (blocking) for cache hierarchy (3 problem dimensions), packing, parallelization (multi-threading), vectorization
- AutoTVM applied to `basic_GEMM` with m,n,k = 2000

→ 2,5 hours to optimize

→ 26,5 GFLOPS on ARM Carmel

BLIS achieves > 28 GFLOPS!

```
1  def basic_GEMM(m, n, k):
2    # B1) Define operation
3    A = te.placeholder((m, k), name="A")
4    B = te.placeholder((k, n), name="B")
5    p = te.reduce_axis((0, k), "p")
6    C = te.compute((m, n), lambda i, j:
7                   te.sum(A[i, p] * B[p, j],
8                   axis=p), name="C")
9
10   # B2) Prepare schedule
11   sched  = te.create_schedule(C.op)
12   (i, j) = C.op.axis
13   (p,)   = C.op.reduce_axis
14
15   # B3) Loop schedule i->j->p
16   sched[C].reorder(i, j, p)
17
18   # B4) Generate code with LLVM backend
19   return tvm.build(sched, [A, B, C],
20                    target="llvm")
```

High Performance (HP) GEMM

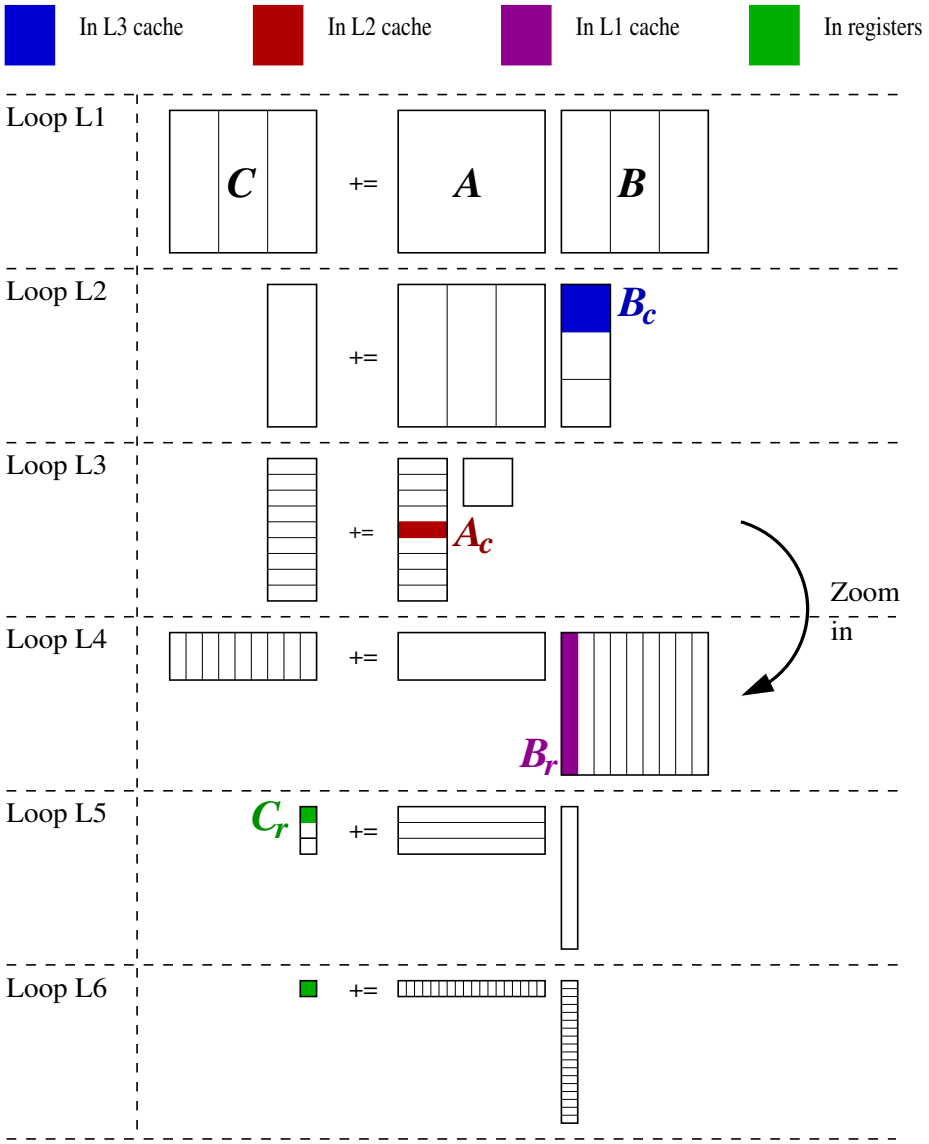
GotoBLAS

K. Goto and R. A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw.

- BLIS
- OpenBLAS
- ARMPL
- AMD AOCL
- Intel MKL, oneAPI?

Automatic generation (TVM) + **Experience**

High Performance (HP) GEMM




```
for ( j_c = 0; j_c < N; j_c += N_c )                    // L1
  for ( p_c = 0; p_c < K; p_c += K_c ) {                // L2
    B_c := B[p_c : p_c + K_c - 1][j_c : j_c + N_c - 1];
    for ( i_c = 0; i_c < M; i_c += M_c ) {              // L3
      A_c := A[i_c : i_c + M_c - 1][p_c : p_c + K_c - 1];
      for ( j_r = 0; j_r < N_c; j_r += N_r )            // L4
        for ( i_r = 0; i_r < M_c; i_r += M_r )          // L5
          // Micro-kernel
          for ( p_r = 0; p_r < K_c; p_r++ )             // L6
            C_c[i_r : i_r + M_r - 1][j_r : j_r + N_r - 1]
                  += A_c[i_r : i_r + M_r - 1][p_r]
                  .    B_c[p_r][j_r : j_r + N_r - 1];
  } }
```

- 5 + 1 loops
- 2 packing routines
- A micro-kernel

- Reduce cache misses
- Access with unit stride from micro-kernel
- Accommodate (SIMD) vectorization & expose loop parallelism

Guided HP GEMM with TVM

1) Tiling:

Why?

- Reduce cache misses

How?

- Partition $m, n, k \rightarrow mc, nc, kc$
- Loop order: jc, pc, ic, jr, ir, pr

```
for ( jc = 0; jc < N; jc += Nc )                          // L1
  for ( pc = 0; pc < K; pc += Kc ) {                      // L2
    Bc := B[pc : pc + Kc - 1][jc : jc + Nc - 1];
    for ( ic = 0; ic < M; ic += Mc ) {                    // L3
      Ac := A[ic : ic + Mc - 1][pc : pc + Kc - 1];
      for ( jr = 0; jr < Nc; jr += Nr )                   // L4
        for ( ir = 0; ir < Mc; ir += Mr )                 // L5
          // Micro-kernel
          for ( pr = 0; pr < Kc; pr++ )                   // L6
            Cc[ir : ir + Mr - 1][jr : jr + Nr - 1]
                  += Ac[ir : ir + Mr - 1][pr]
                  .   Bc[pr][jr : jr + Nr - 1];
  } }
```

Guided HP GEMM with TVM

2) Packing:

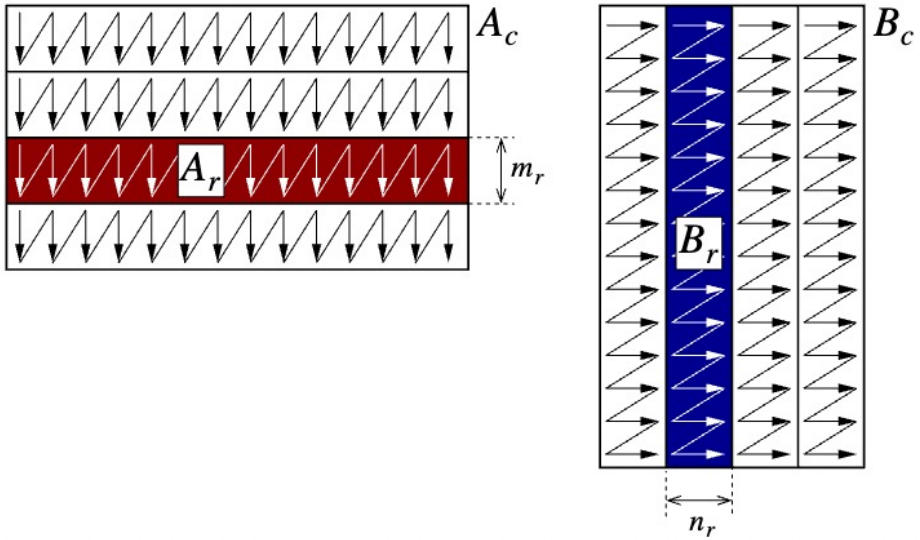
Why?

- Reduce cache misses
- Access with unit stride

How?

- Buffer $Ac \rightarrow mc\ x\ kc$
- Buffer $Bc \rightarrow kc\ x\ nc$

```
for ( jc = 0; jc < N; jc += Nc )                          // L1
  for ( pc = 0; pc < K; pc += Kc ) {                      // L2
    Bc := B[pc : pc + Kc − 1][jc : jc + Nc − 1];
    for ( ic = 0; ic < M; ic += Mc ) {                    // L3
      Ac := A[ic : ic + Mc − 1][pc : pc + Kc − 1];
      for ( jr = 0; jr < Nc; jr += Nr )                   // L4
        for ( ir = 0; ir < Mc; ir += Mr )                 // L5
          // Micro-kernel
          for ( pr = 0; pr < Kc; pr++ )                   // L6
            Cc[ir : ir + Mr − 1][jr : jr + Nr − 1]
                  += Ac[ir : ir + Mr − 1][pr]
                  ·   Bc[pr][jr : jr + Nr − 1];
  } }
```



Guided HP GEMM with TVM

1) Tiling:

```
for ( jc=0; jc<n; jc+=nc )
 for ( pc=0; pc<k; pc+=kc )
  for ( ic=0; ic<m; ic+=mc )
   for ( jr=0; jr<nc; jr++ )
    for ( ir=0; ir<mc; ir++ )
     for ( pr=0; pr<kc; pr++ )
      C[ic+ir][jc+jr]
        += A[ic+ir][pc+pr]
        * B[pc+pc][jc+jr];
```

```
1  def block_GEMM(m, n, k, mc, nc, kc):
2    # B1) as in basic_GEMM
3    # Omitted for brevity
4
5    # B2) Prepare schedule
6    sched = te.create_schedule(C.op)
7    ic, jc, \
8    ir, jr = sched[C].tile(C.op.axis[0],
9                           C.op.axis[1], mc, nc)
10   pc, pr = sched[C].split(p, factor=kc)
11
12
13
14
15
16
17
```

Partition:

$$m, n, k \rightarrow mc, nc, kc$$

Guided HP GEMM with TVM

1) Tiling:

```
for ( jc=0; jc<n; jc+=nc )
 for ( pc=0; pc<k; pc+=kc )
  for ( ic=0; ic<m; ic+=mc )
   for ( jr=0; jr<nc; jr++ )
    for ( ir=0; ir<mc; ir++ )
     for ( pr=0; pr<kc; pr++ )
      C[ic+ir][jc+jr]
        += A[ic+ir][pc+pr]
        * B[pc+pc][jc+jr];
```

Schedule:

$$jc, pc, ic, jr, ir, pr$$

```
1  def block_GEMM(m, n, k, mc, nc, kc):
2    # B1) as in basic_GEMM
3    # Omitted for brevity
4
5    # B2) Prepare schedule
6    sched = te.create_schedule(C.op)
7    ic, jc, \
8    ir, jr = sched[C].tile(C.op.axis[0],
9                           C.op.axis[1], mc, nc)
10   pc, pr = sched[C].split(p, factor=kc)
11
12   # B3) Loop schedule as in B3A2C0
13   sched[C].reorder(jc, pc, ic, jr, ir, pr)
14
15   # B4) Generate code with LLVM backend
16   return tvm.build(sched, [A, B, C],
17                    target="llvm")
```

Guided HP GEMM with TVM

2) Packing:

- Buffer $Ac \rightarrow mc\ x\ kc$
- Buffer $Bc \rightarrow kc\ x\ nc$

```
def packed_GEMM(m, n, k, mc, nc, kc, mr, nr):
  # P1) Define operation
  A = te.placeholder((m, k), name="A")
  B = te.placeholder((k, n), name="B")

  # 4D view into A and B to induce creation
  # of buffer by TVM
  Ac = te.compute((math.ceil(k/kc),
                   math.ceil(m/mr), kc, mr),
                   lambda i, j, q, r:
                   A[j * mr + r, i * kc + q],
                   name="Ac")
  Bc = te.compute((math.ceil(k/kc),
                   math.ceil(n/nr), kc, nr),
                   lambda i, j, q, r:
                   B[ i * kc + q, j * nr + r ],
                   name="Bc")
```

```

  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda i, j:
         te.sum(Ac[p//kc, i//mr,
                   tvm.tir.indexmod(p, kc),
                   tvm.tir.indexmod(i, mr)] *
                Bc[p//kc, j//nr,
                   tvm.tir.indexmod(p, kc),
                   tvm.tir.indexmod(j, nr)],
                axis=p), name="C")

  # P2) Prepare schedule
  sched = te.create_schedule(C.op)
  ic, jc, \
  ir, jr  = sched[C].tile(C.op.axis[0],
                          C.op.axis[1],
                          mc, nc)
  pc, pr = sched[C].split(p, factor=kc)

  # P3) Place Ac, Bc in the desired loops
  sched[Bc].compute_at(sched[C], pc)
  sched[Ac].compute_at(sched[C], ic)

  # P4) Loop schedule as in B3A2C0
  sched[C].reorder(jc, pc, ic, jr, ir, pr)

  # P5) Generate code with LLVM backend
  return tvm.build(sched, [A, B, C],
                   target="llvm")
```

Guided HP GEMM with TVM

3) Micro-kernel:

Why?

- SIMD vectorization

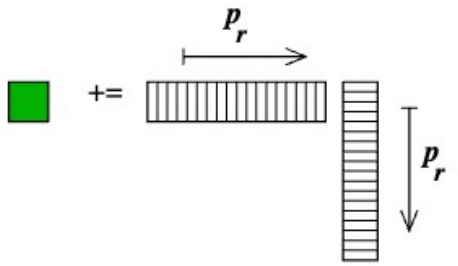
How?

- $C \rightarrow mr\ x\ nr$ in registers
- Ac, Bc streamed from caches

```
for ( jc = 0; jc < N; jc += Nc )                    // L1
  for ( pc = 0; pc < K; pc += Kc ) {                // L2
    Bc := B[pc : pc + Kc - 1][jc : jc + Nc - 1];
    for ( ic = 0; ic < M; ic += Mc ) {              // L3
      Ac := A[ic : ic + Mc - 1][pc : pc + Kc - 1];
      for ( jr = 0; jr < Nc; jr += Nr )             // L4
        for ( ir = 0; ir < Mc; ir += Mr )           // L5
          // Micro-kernel
          for ( pr = 0; pr < Kc; pr++ )             // L6
            C[ir : ir + Mr - 1][jr : jr + Nr - 1]
                  += Ac[ir : ir + Mr - 1][pr]
                  *  Bc[pr][jr : jr + Nr - 1];
  } }
```

```
for ( pr=0; pr<kc; pr++ )
  C(ic+ir:ic+ir+mr-1,jc+jr:jc+jr+nr-1)
    += Ac(ir:ir+mr-1,pr)
    *  Bc(pr,jr:jr+nr-1);
```

L6



Guided HP GEMM with TVM

3) Micro-kernel:

- $C \rightarrow mr\ x\ nr$ in registers
- Ac, Bc streamed from caches

Architecture-specific:

- ARM NEON/SVE
- Intel AVX/AVX-512
- AMD
- RISC-V
- MCUs

```
1  def opt_GEMM(m, n, k, mc, nc, kc, mr, nr):
2    # P1), P2), P3) as in pack_GEMM
3    # Omitted for brevity
4
5    # P4) Expose loops inside micro-kernel
6    ir, it = sched[C].split(ir, factor=mr)
7    jr, jt = sched[C].split(jr, factor=nr)
8
9    # P5) Loop schedule as in B3A2C0
10   sched[C].reorder(jc, pc, ic,
11                    jr, ir, pr, it, jt)
12
13   # P6) Unroll+vectorize micro-kernel loops
14   sched[C].unroll(it)
15   sched[C].vectorize(jt)
16   i, j = Bc.op.axis
17   sched[Bc].vectorize(j)
18   i, j, q, r = Ac.op.axis
19   sched[Ac].vectorize(r)
20
21   # P7) Generate code with LLVM backend
22   return tvm.build(sched, [A, B, C],
23                    target="llvm")
```

Guided HP GEMM with TVM

Recap:

- We can mimic all optimization techniques in modern instances of GEMM: tiling, packing, vectorization, parallelization

→ Why didn't we use BLIS (or any other HP library) GEMM then?

- Large memory footprint
- Missing functionality: INT8/INT16/INT32, BF16/TF32?
- Suboptimal performance
- Hardware-specific solution in a heterogenous world

Experiments on Portability

Pros:

+ Automatic generation of a variety of micro-kernels
+ Automatic generation for multiple datatypes (including INT)
+ Easy activation/deactivation of packing and associated micro-kernels
+ Automatic parallelization/vectorization options
+ Automatic generation of micro-kernels for ARM, Intel, AMD,…
+ Family of algorithms for GEMM and micro-kernels

→ Maintenance and exploration of different optimization options

Conns (TVM):

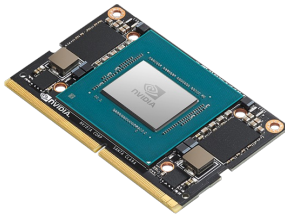
- No explicit control of vector instructions generated by compiler
- No parallelization of multiple loops
- Mixed precision

Experiments on Portability

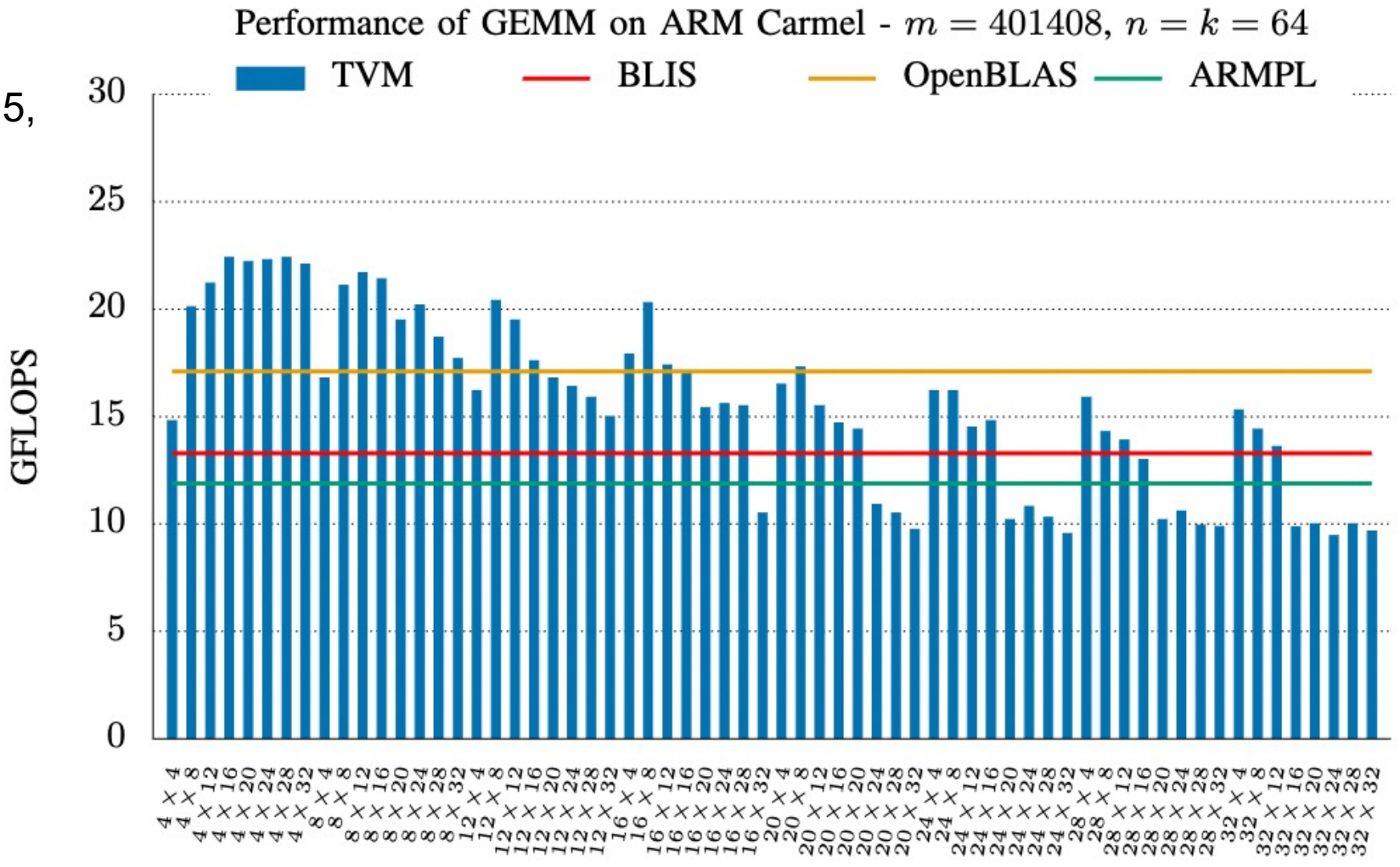
Dimensions of micro-kernels

```
def opt_GEMM(m, n, k, mc, nc, kc, mr, nr)
```

ARM Carmel



ResNet-50 v1.5, layer #6

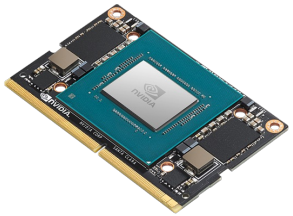


Experiments on Portability

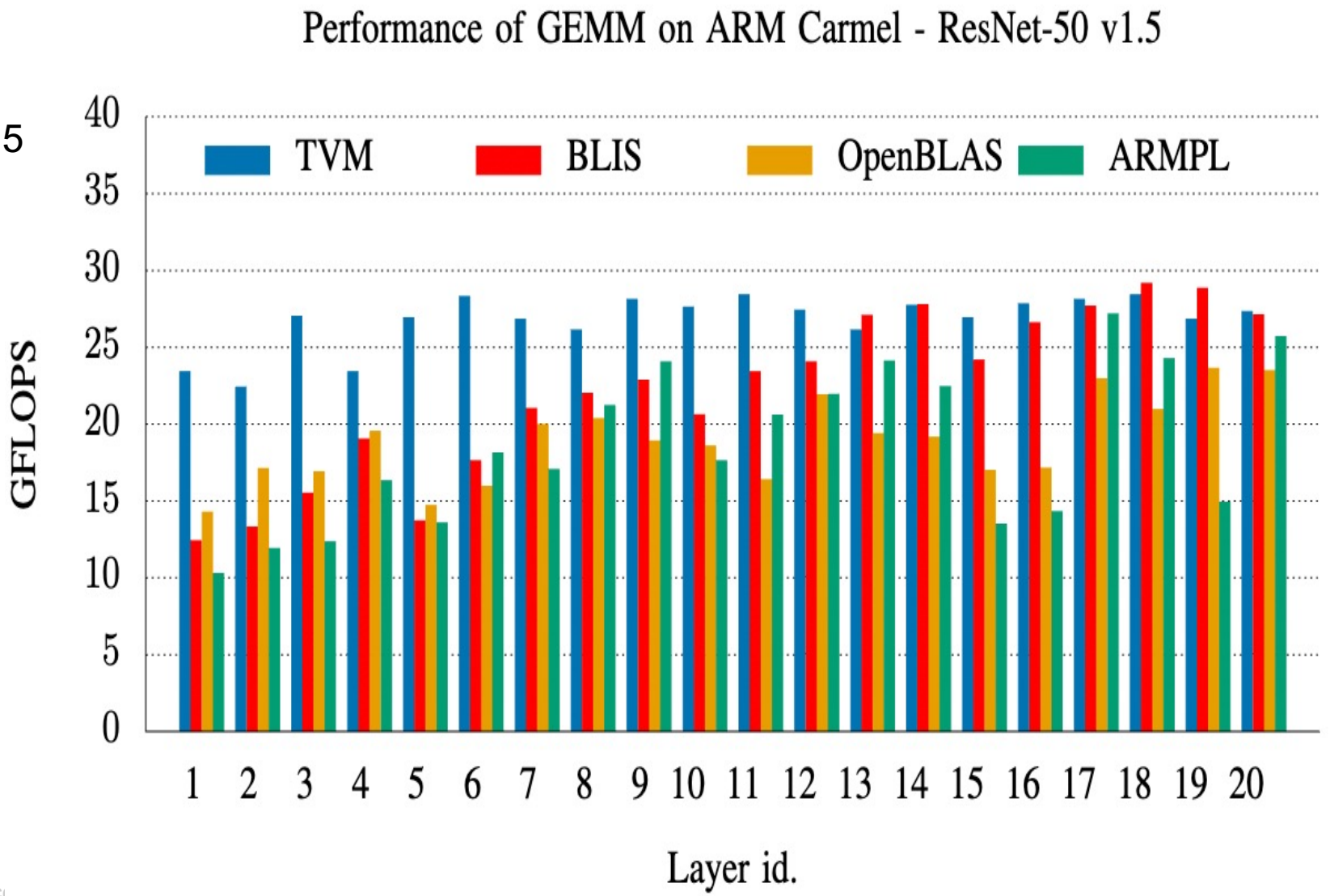
Dimensions of micro-kernels

```
def opt_GEMM(m, n, k, mc, nc, kc, mr, nr)
```

ARM Carmel



ResNet-50 v1.5

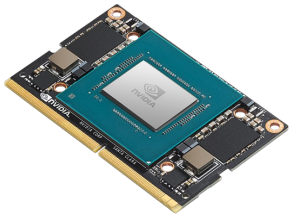


Experiments on Portability

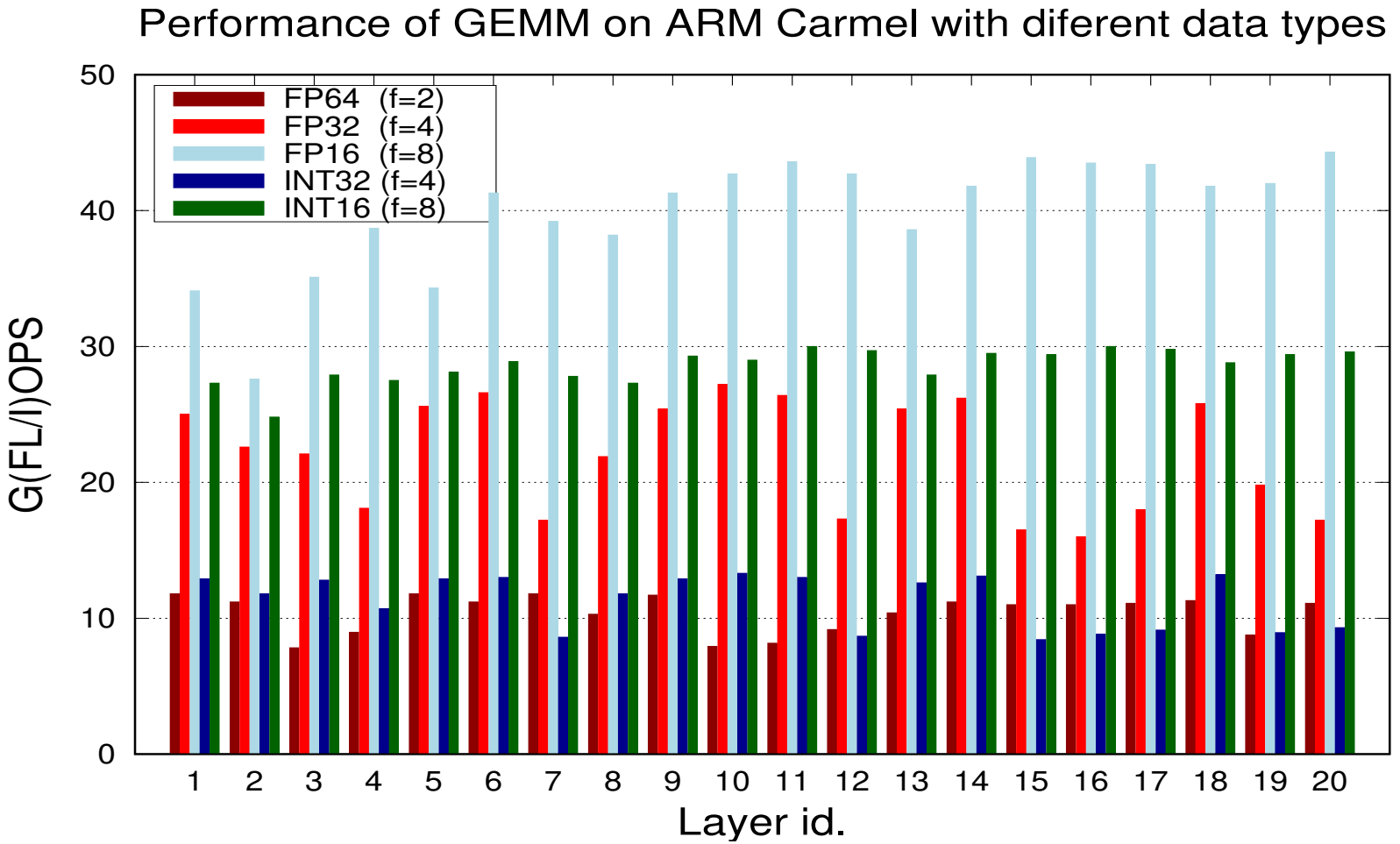
Data types

Just modify `dtype` for the operands: FP64, FP32, FP16…

ARM Carmel



ResNet-50 v1.5



Experiments on Portability

Packing/no packing:

Basically, eliminate references to Ac and/or Bc

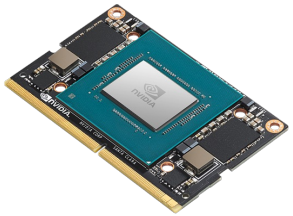
```
1  def packed_GEMM(m, n, k, mc, nc, kc, mr, nr):
2    # P1) Define operation
3    A = te.placeholder((m, k), name="A")
4    B = te.placeholder((k, n), name="B")
5
6    # 4D view into A and B to induce creation
7    # of buffer by TVM
8    Ac = te.compute((math.ceil(k/kc),
9                     math.ceil(m/mr), kc, mr),
10                    lambda i, j, q, r:
11                    A[j * mr + r, i * kc + q],
12                    name="Ac")
13   Bc = te.compute((math.ceil(k/kc),
14                    math.ceil(n/nr), kc, nr),
15                    lambda i, j, q, r:
16                    B[ i * kc + q, j * nr + r ],
17                    name="Bc")
```

Experiments on Portability

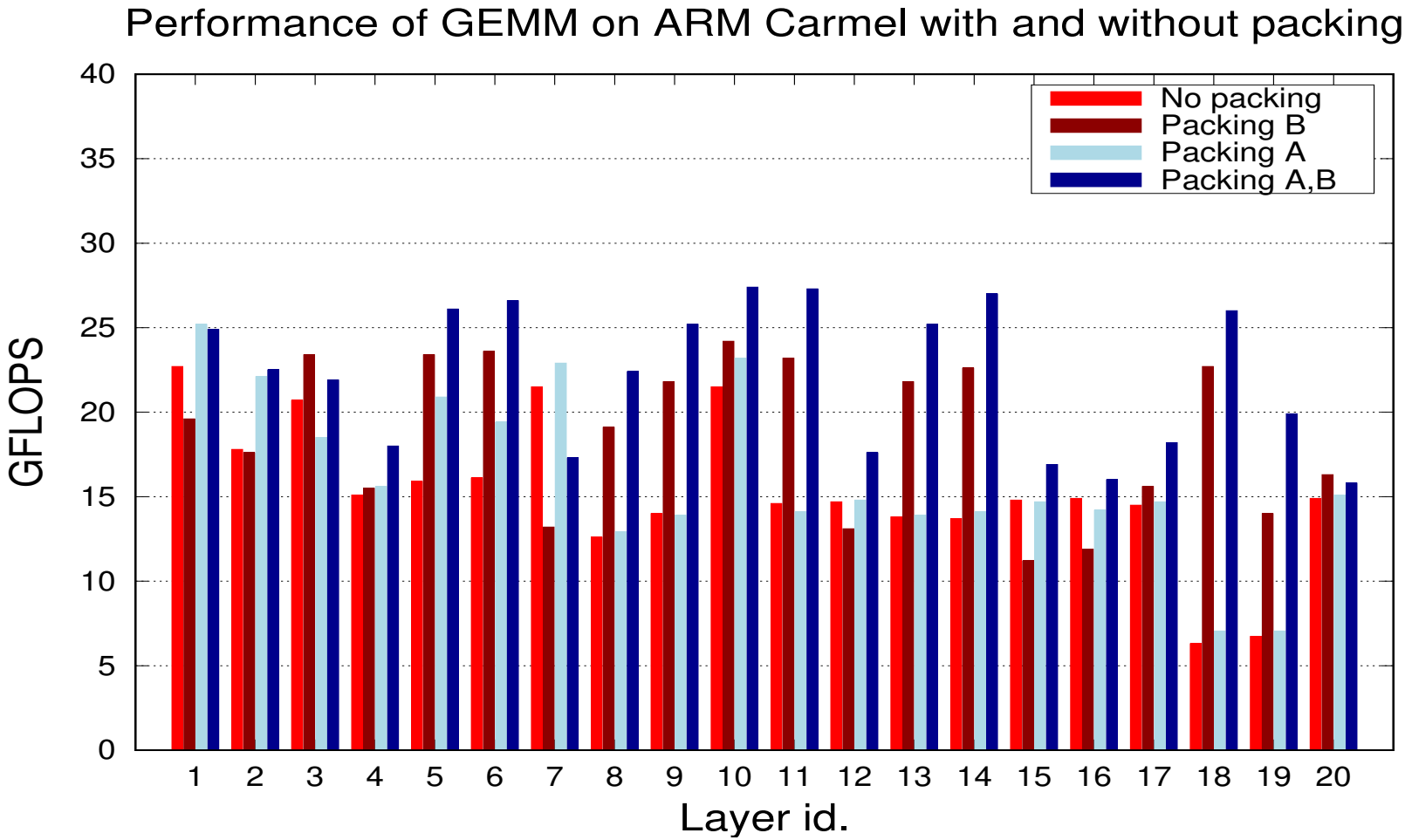
Packing/no packing:

Basically, eliminate references to Ac and/or Bc

ARM Carmel



ResNet-50 v1.5

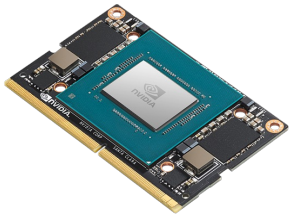


Experiments on Portability

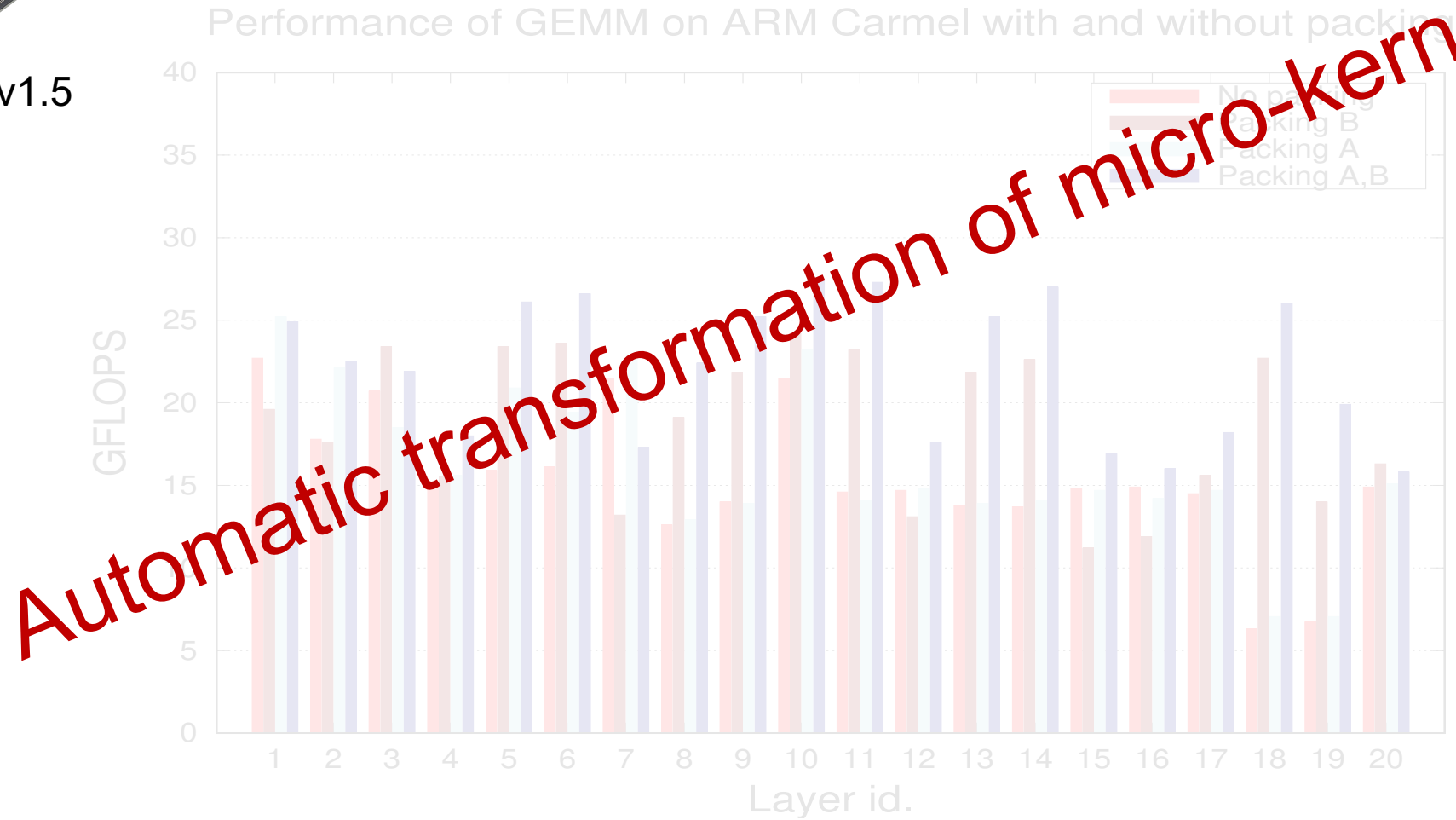
Packing/no packing:

Basically, eliminate references to Ac and/or Bc

ARM Carmel



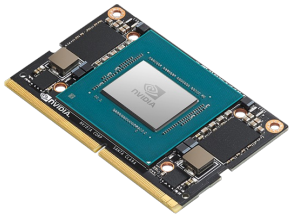
ResNet-50 v1.5



Automatic transformation of micro-kernels

Experiments on Portability

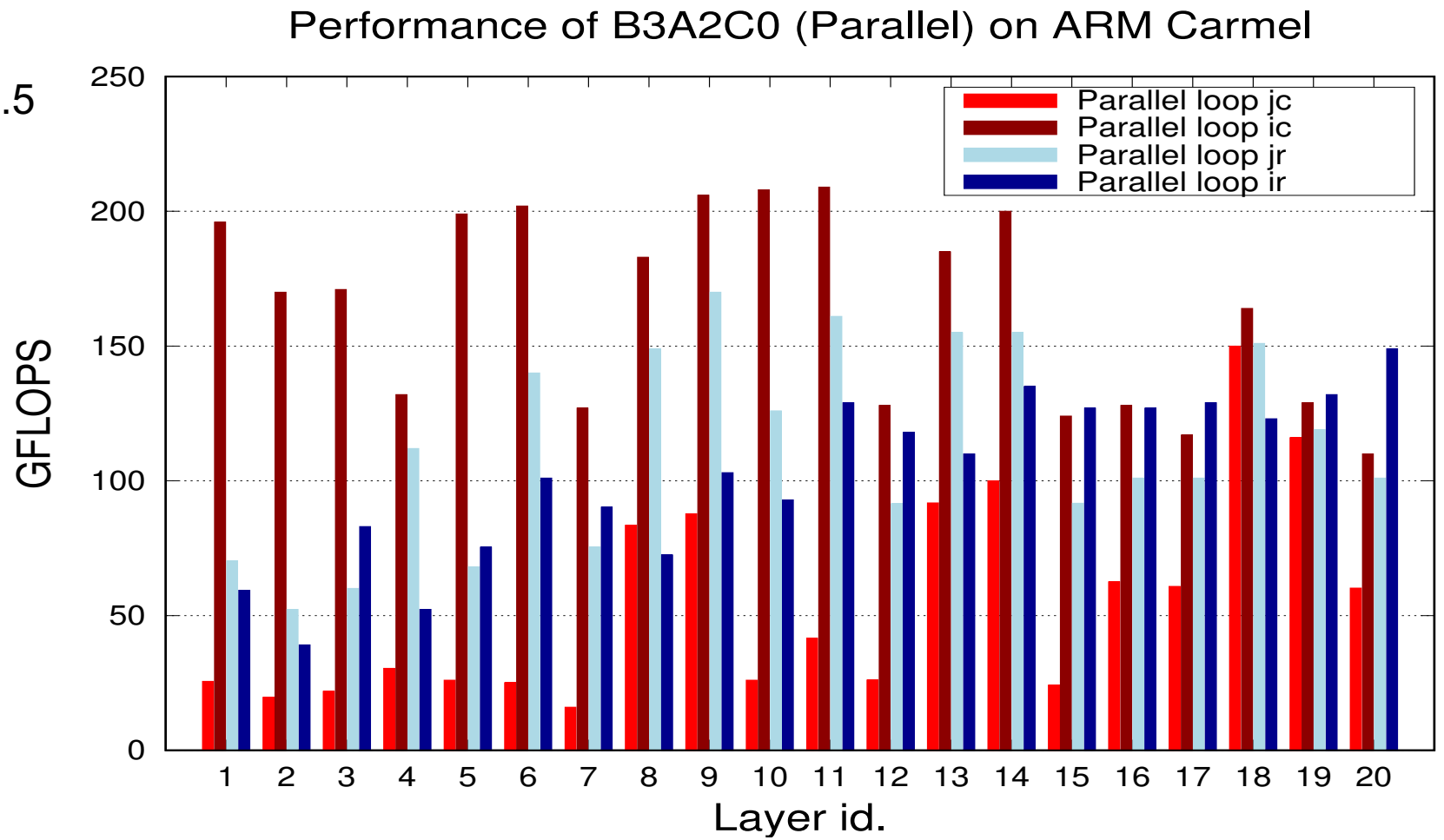
ARM Carmel



ResNet-50 v1.5

Parallelization:

```
sched[C].reorder(jc, pc, ic, jr, ir, pr, it, jt)
sched[C].parallel(ic)
```



Experiments on Portability

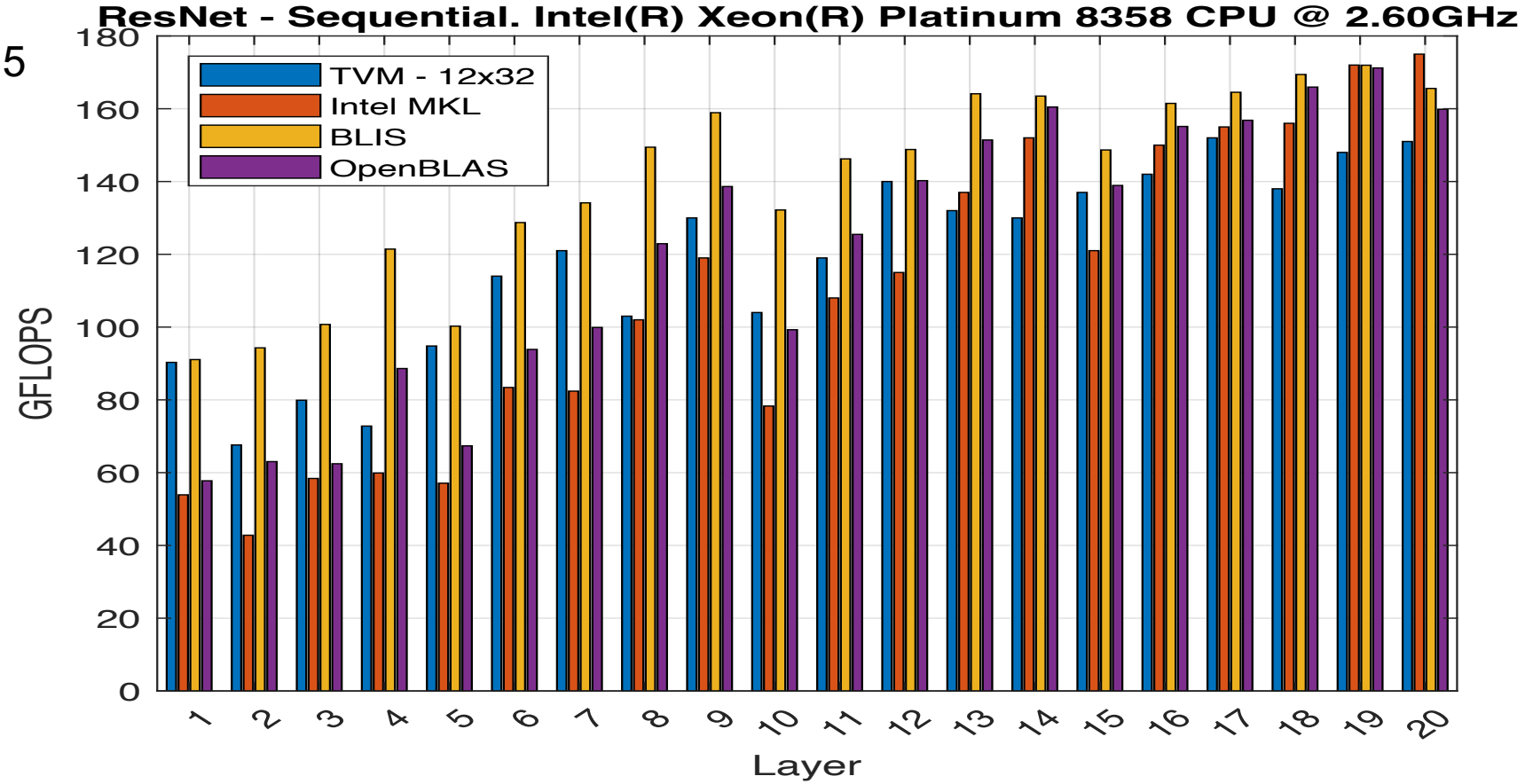
Target architectures:

Intel Ice Lake

ResNet-50 v1.5

ARM: `target = "llvm -device=arm_cpu -mattr=+v8.2a,+fp-armv8,+neon`

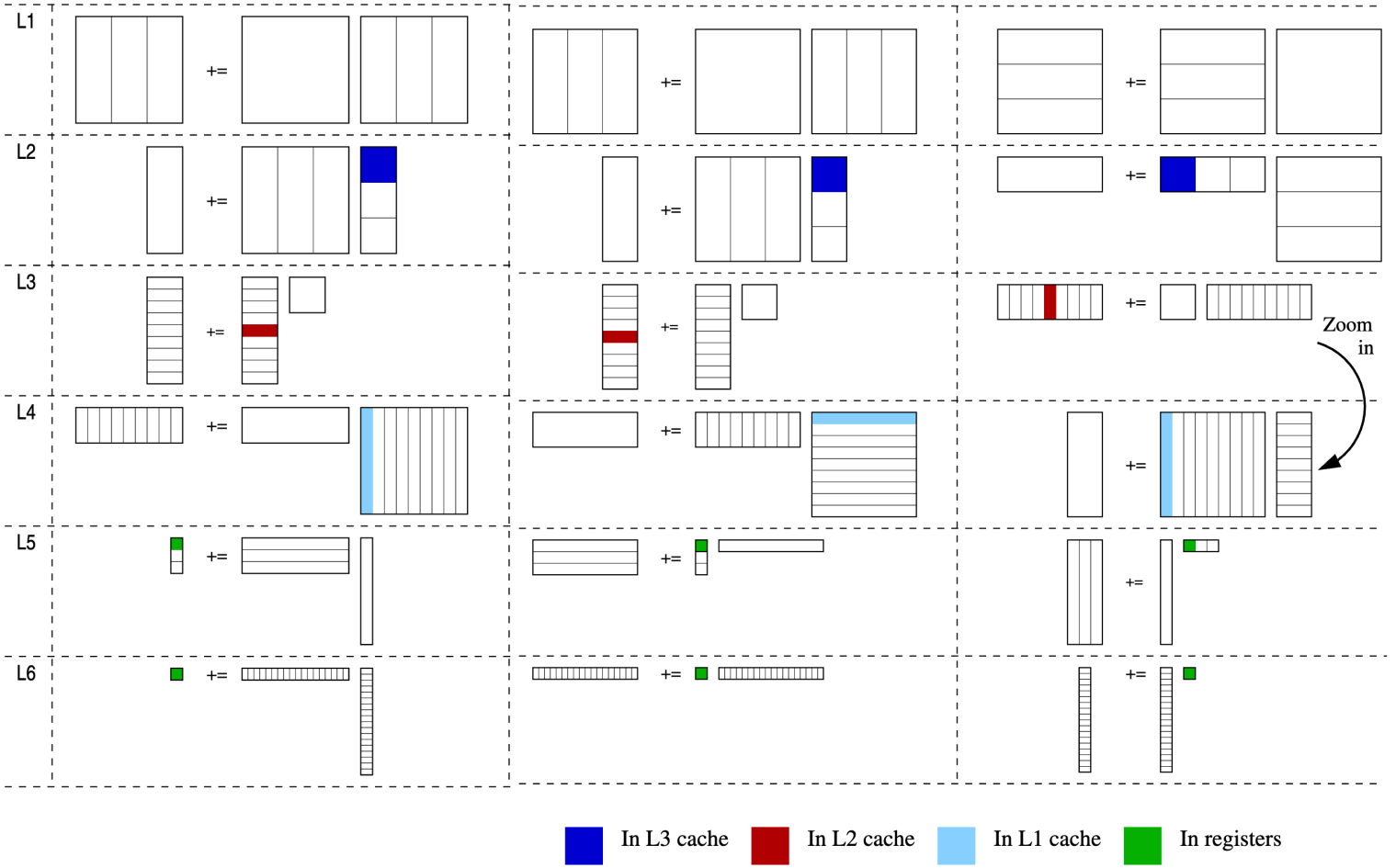
Intel: `target = "llvm -mcpu=icelake-avx512"`



Experiments on Portability

Algorithms and micro-kernels:

B3A2C0, A3B2C0, C3B2A0, B3C2A0, C3A2B0, A3C2B0

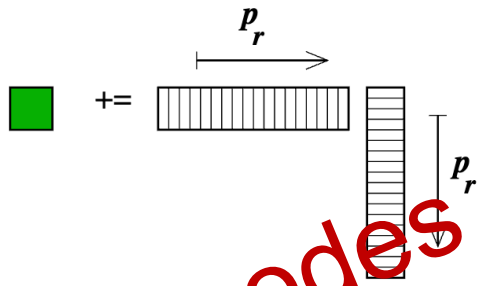


Experiments on Portability

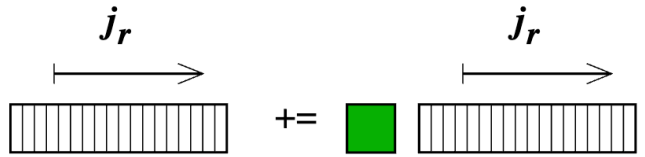
Algorithms and micro-kernels:

Micro-tile of C/A/B resident in processor registers

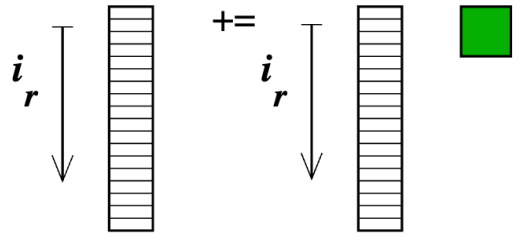
```
1 for (pr=0; pr<kc; pr++) // Loop L6
2   C(ic+ir:ic+ir+mr-1,
3     jc+jr:jc+jr+nr-1)
4     += Ac(ir:ir+mr-1,pr)
5     *  Bc(pr,jr:jr+nr-1);
```




```
1 for (jr=0; jr<nc; jr++) // Loop L6
2   Cc(ir:ir+mr-1,jr)
3     += A(ic+ir:ic+ir+mr-1,
4          pc+pr:pc+pr+kr-1)
5     *  Bc(pr:pr+kr-1,jr);
```




```
1 for (ir=0; ir<mr; ir++) // Loop L6
2   Cc(ir,jr:jr+nr-1)
3     += Ac(ir,pr:pr+kr-1)
4     *  B(pc+pr:pc+pr+kr-1,
5          jc+jr:jc+jr+nr-1);
```

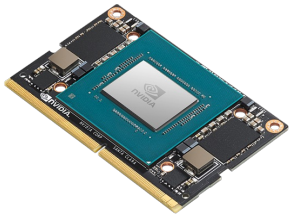


Some changes to TVM codes

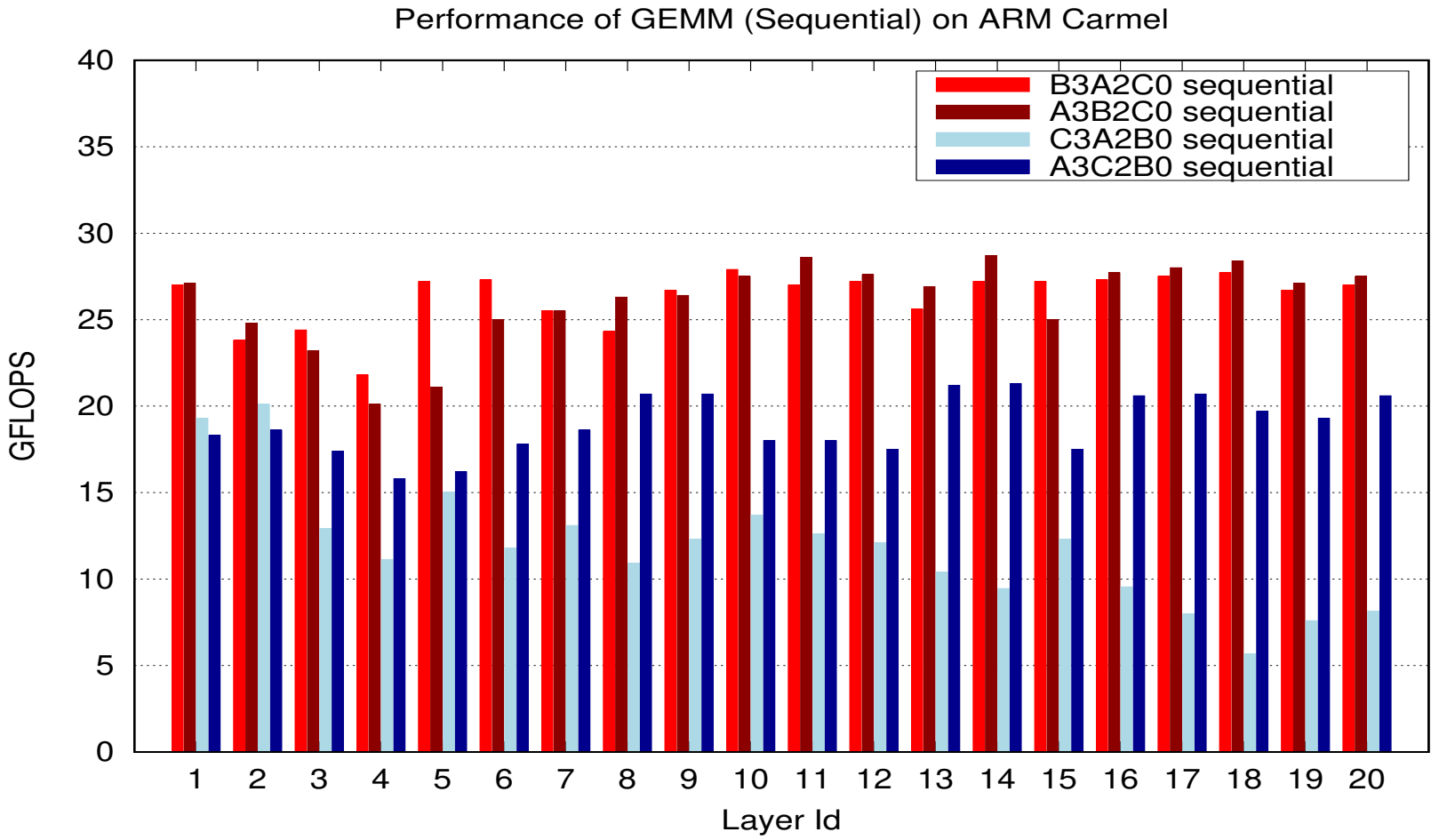
Experiments on Portability

Algorithms and micro-kernels:

ARM Carmel



ResNet-50 v1.5



Conclusions

Libraries provide a solution that is too rigid in a heterogenous world!

but

Automatic generation yields an explosion on the optimization search space

For convolution & GEMM, automatic generation guided by experience

- Search space limited to mr, nr. Only integer multiples of SIMD length
- Cache configuration parameters: mc, nc, kc, chosen using analytical model

T. M. Low, F. D. Igual, T. M. Smith, E. S. Quintana-Ortí. 2017. Analytical modeling is enough for high performance BLIS. ACM Trans. Math. Softw.

Thanks!

www.eFlows4HPC.eu

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

Additional Material

Autotune GEMM with TVM

$$C = A \cdot B$$

Add a function wrapper and return the operation tensors

```
@auto_scheduler.register_workload
def basic_GEMM(m, n, k):
  # B1) Define operation
  A = te.placeholder((m, k), name="A")
  B = te.placeholder((k, n), name="B")
  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda i, j:
                 te.sum(A[i, p] * B[p, j],
                 axis=p), name="C")

  # B2) Return tensors
  return [A, B, C]


def autotune_gemm(M,N,K,target):
  # A1) Create search task
  task = tvm.auto_scheduler.SearchTask(
         func=basic_gemm, args=(M,N,K),
         target=target)

  # A2) Configure search task
  tune = auto_scheduler.TuningOptions(
         num_measure_trials=1000,
         measure_callbacks=
         [RecordToFile("gemm.json")])

  # A3) Initiate the search
  task.tune(tune)

  # A4) Get the best approach
  sch, args = task.apply_best("gemm.json")

  # A5) Generate code with target backend
  return tvm.build(sch, args, target)
```

Autotune GEMM with TVM

$$C = A \cdot B$$

Add a function wrapper and return the operation tensors

Create a search task and configure it

```
1  @auto_scheduler.register_workload
2  def basic_GEMM(m, n, k):
3    # B1) Define operation
4    A = te.placeholder((m, k), name="A")
5    B = te.placeholder((k, n), name="B")
6    p = te.reduce_axis((0, k), "p")
7    C = te.compute((m, n), lambda i, j:
8                   te.sum(A[i, p] * B[p, j],
9                   axis=p), name="C")
10
11   # B2) Return tensors
12   return [A, B, C]
13
14
15 def autotune_gemm(M,N,K,target):
16   # A1) Create search task
17   task = tvm.auto_scheduler.SearchTask(
18          func=basic_gemm, args=(M,N,K),
19          target=target)
20
21   # A2) Configure search task
22   tune = auto_scheduler.TuningOptions(
23          num_measure_trials=1000,
24          measure_callbacks=
25          [RecordToFile("gemm.json")])
26
27   # A3) Initiate the search
28   task.tune(tune)
29
30   # A4) Get the best approach
31   sch, args = task.apply_best("gemm.json")
32
33   # A5) Generate code with target backend
34   return tvm.build(sch, args, target)
```

Autotune GEMM with TVM

$$C = A \cdot B$$

Add a function wrapper and return the operation tensors

Create a search task and configure it

Launch the search and apply the best

```python
@auto_scheduler.register_workload
def basic_GEMM(m, n, k):
  # B1) Define operation
  A = te.placeholder((m, k), name="A")
  B = te.placeholder((k, n), name="B")
  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda i, j:
                 te.sum(A[i, p] * B[p, j],
                 axis=p), name="C")

  # B2) Return tensors
  return [A, B, C]


def autotune_gemm(M,N,K,target):
  # A1) Create search task
  task = tvm.auto_scheduler.SearchTask(
         func=basic_gemm, args=(M,N,K),
         target=target)

  # A2) Configure search task
  tune = auto_scheduler.TuningOptions(
         num_measure_trials=1000,
         measure_callbacks=
         [RecordToFile("gemm.json")])

  # A3) Initiate the search
  task.tune(tune)

  # A4) Get the best approach
  sch, args = task.apply_best("gemm.json")

  # A5) Generate code with target backend
  return tvm.build(sch, args, target)
```

Autotune GEMM with TVM

$$C = A \cdot B$$

Add a function wrapper and return the operation tensors

Create a search task and configure it

Launch the search and apply the best

Generate code for target

```
@auto_scheduler.register_workload
def basic_GEMM(m, n, k):
  # B1) Define operation
  A = te.placeholder((m, k), name="A")
  B = te.placeholder((k, n), name="B")
  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda i, j:
                 te.sum(A[i, p] * B[p, j],
                 axis=p), name="C")

  # B2) Return tensors
  return [A, B, C]


def autotune_gemm(M,N,K,target):
  # A1) Create search task
  task = tvm.auto_scheduler.SearchTask(
         func=basic_gemm, args=(M,N,K),
         target=target)

  # A2) Configure search task
  tune = auto_scheduler.TuningOptions(
         num_measure_trials=1000,
         measure_callbacks=
         [RecordToFile("gemm.json")])

  # A3) Initiate the search
  task.tune(tune)

  # A4) Get the best approach
  sch, args = task.apply_best("gemm.json")

  # A5) Generate code with target backend
  return tvm.build(sch, args, target)
```