Using MLIR to Optimize Basic Linear Algebraic Subprograms

Steven Varoumas

Huawei Technologies Research & Development (UK)

Cambridge Research Centre – Compiler Lab

HUAWEI

Motivations

Writing libraries is a time consuming task:

> Many man-hours spent fine-tuning code to achieve best performance.

> Has to be adapted and optimized for any new hardware.

→ Can we give compilers the task of optimizing libraries that can compete with hand-written ones?

In this work, we intend to generate an optimized math library using compiler technologies.

> Aim to support the **B**asic **L**inear **A**lgebra **S**ubprograms (**BLAS**) specification.

> Reduce time taken optimizing/fine-tuning math functions.

> Automatize creation of hardware-specific code.

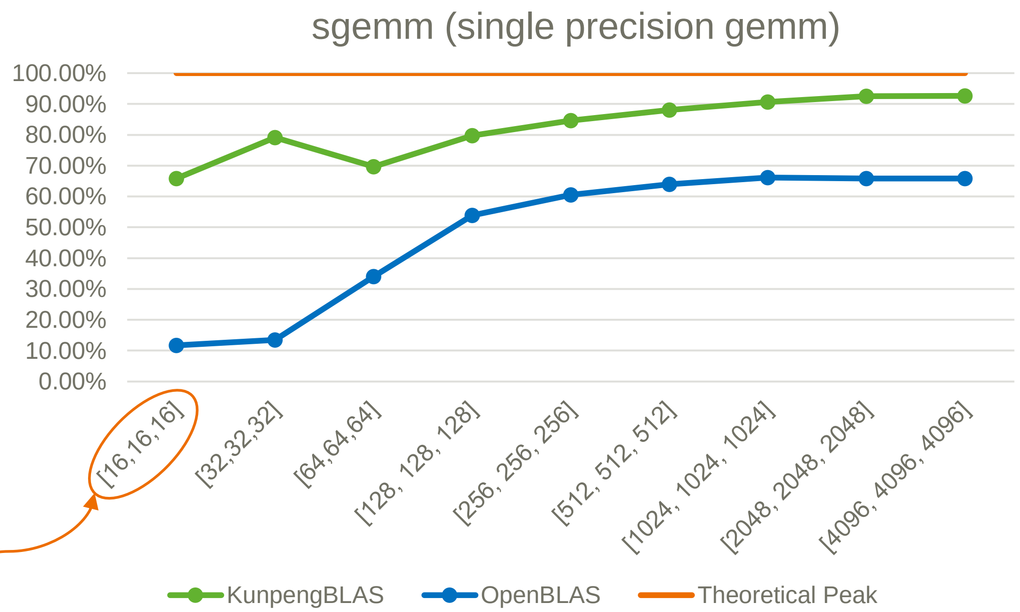
> Leverage the functionalities and extensibility of the MLIR framework.

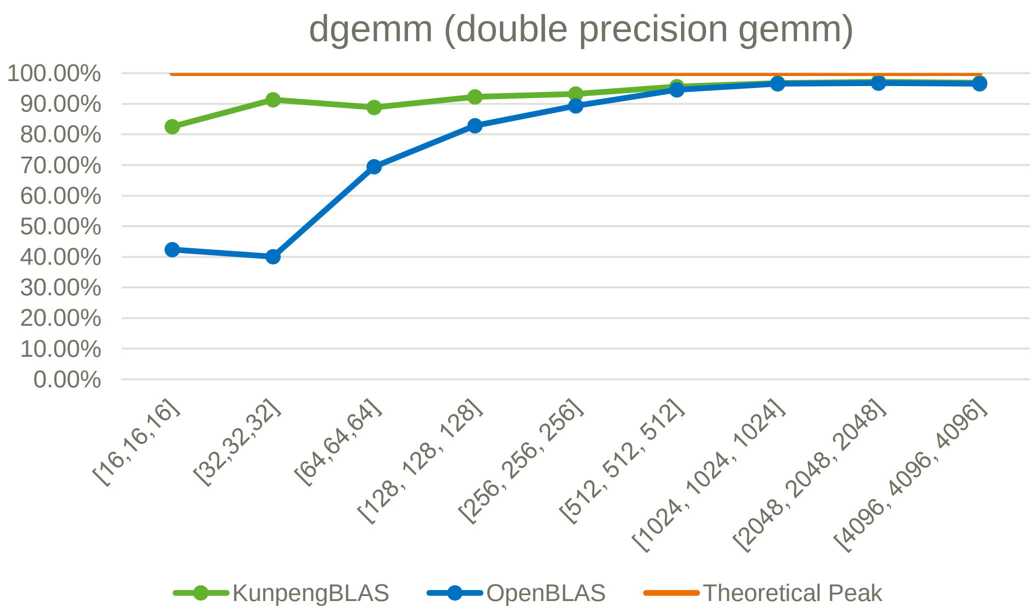
Objective: Explore what performance results we can get from this approach (expectation: reach 90% of the performance of an in-house hand-tuned BLAS library).

HUAWEI

Context: KunpengBLAS library

- BLAS: specification that defines a set of linear algebra functions (e.g. dot product, matrix multiplication).
- Reference implementation of BLAS: KunpengBLAS (“*KPL*”) library (we use the *single-thread* version).
- Hardware for measurements: Huawei Kunpeng 920 (64bits ARMv8-based processor).
- We particularly *focus on GEMM* (*General Matrix-Matrix multiplication*): performance critical.
  - → GEMM is $\boldsymbol{C = \alpha AB + \beta C}$ (A, B and C are matrices, α and β are scalars)
  - → KPL is able to reach >90% of the theoretical peak of the hardware for sgemm/dgemm:





M=N=K=16
A:MxK | B:KxN | C:MxN

HUAWEI

Context: GEMM Core Transformations

We rely on the following core transformations:

> Tiling – Apply the operation on subsets (*tiles*) of the matrices.

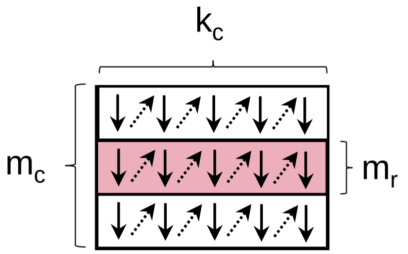
> Packing – Re-mapping data in the A and B tiles to get sequential memory accesses.

This follows the work of Goto & Van De Geijn [2] to compile an efficient GEMM.

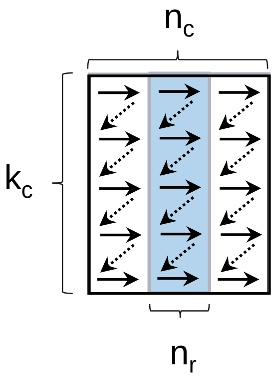
Their use in an MLIR pipeline has been described by Bondhugula [1].

```
for j = 0 to N-1 by steps of n_c:
  for p = 0 to K-1 by steps of k_c:
    Bc = B(p:p+kc-1, j:j+nc-1) // Pack into Bc
    for i = 0 to M-1 by steps of m_c:
      Ac = A(i:i+m-1,p:p+k-1)  // Pack into Ac
      for jj = 0 to nc-1 by steps of n_r:
        for ii = 0 to mc-1 by steps of m_r:
          for pp = 0 to kc-1 by steps of 1: // Microkernel
            C(ii:ii+mr-1, jj:jj+nr-1) += Ac(ii,ii+mr-1,pp) * Bc(pp,jj:jj+nr-1)
```

optimized matrix multiplication (pseudocode)



packed tile of A



packed tile of B

[1] Bondhugula, Uday. "High performance code generation in MLIR: An early case study with gemm." *arXiv:2003.00532* (2020).

[2] Goto, Kazushige, and Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication." *ACM Transactions on Mathematical Software (TOMS)* 34.3 (2008): 1-25.

Project Overview: Compilation Pipeline

Full pipeline to generate/optimize/compile BLAS functions:

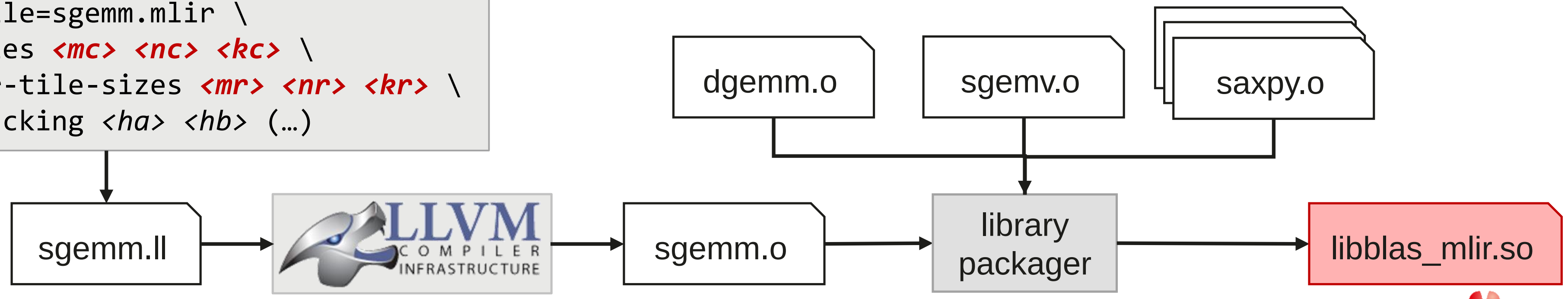
- A high-level definition of the function is generated directly in the `linalg` dialect (does *not* come from a frontend… *yet*).
- The generated file is given to an optimizing MLIR compiler (***mlirc***), with a list of transformations to apply and their arguments. The optimized functions are packaged into a library (*libblas_mlir.so*).

```
func.func @gemm(%A: tensor<?x?xf32>, %B: tensor<?x?xf32>, %C: tensor<?x?xf32>) -> tensor<?x?xf32> {
    %res = linalg.generic ins(%A, %B : tensor<?x?xf32>, tensor<?x?xf32>) outs(%C : tensor<?x?xf32>) {
    ^bb0(%a: f32, %b: f32, %c: f32):
      %m = arith.mulf %a, %b : f32
      %a = arith.addf %out, %m : f32
      linalg.yield %m : f32
    } -> tensor<?x?xf32>
    return %res : tensor<?x?xf32>
  }
```

sgemm.mlir

High-level definition (simplified for space)*

```
mlirc --input-file=sgemm.mlir \
      --tile-sizes <mc> <nc> <kc> \
      --register-tile-sizes <mr> <nr> <kr> \
      --hoist-packing <ha> <hb> (…)
```



*actual gemm is $C = \alpha AB + \beta C$

Multi-Kernel Approach

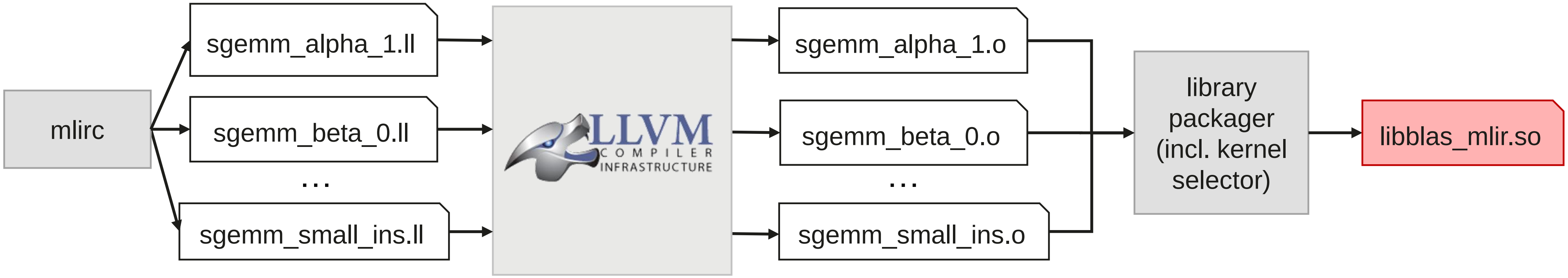
Transformations may depend on the specific inputs of the function: one set of transformations/parameters is not always good for all possible inputs. For example, packing is not always helpful for small matrices [1].

→ We use a multi-kernel approach to enhance each function's performance:

> For each BLAS function (e.g. *gemm*), we generate a set of *kernels.*

> Kernels are optimized variants of the function, tuned for specific inputs.

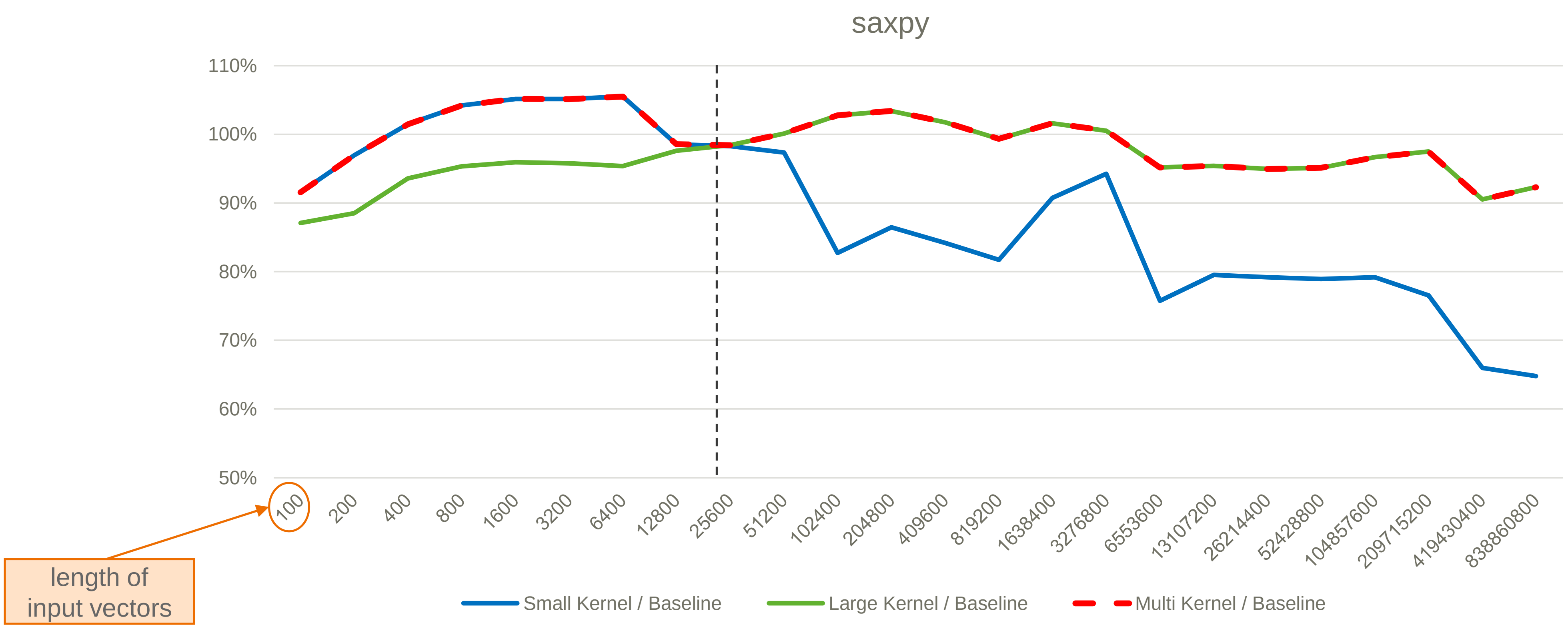
> At runtime, a kernel selector choses the "best" *kernel,* based on dynamic information.



[1] Yang, Weiling, et al. "LIBSHALOM: optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

Multi-Kernel Approach: Example (axpy)

Using a different kernel for small input vectors and large input vectors gives results consistently >90% of the baseline (KPL) for saxpy (single-precision axpy*):



*axpy is $\vec{y} = \alpha\vec{x} + \vec{y}$ (scalar multiplication + vector addition)

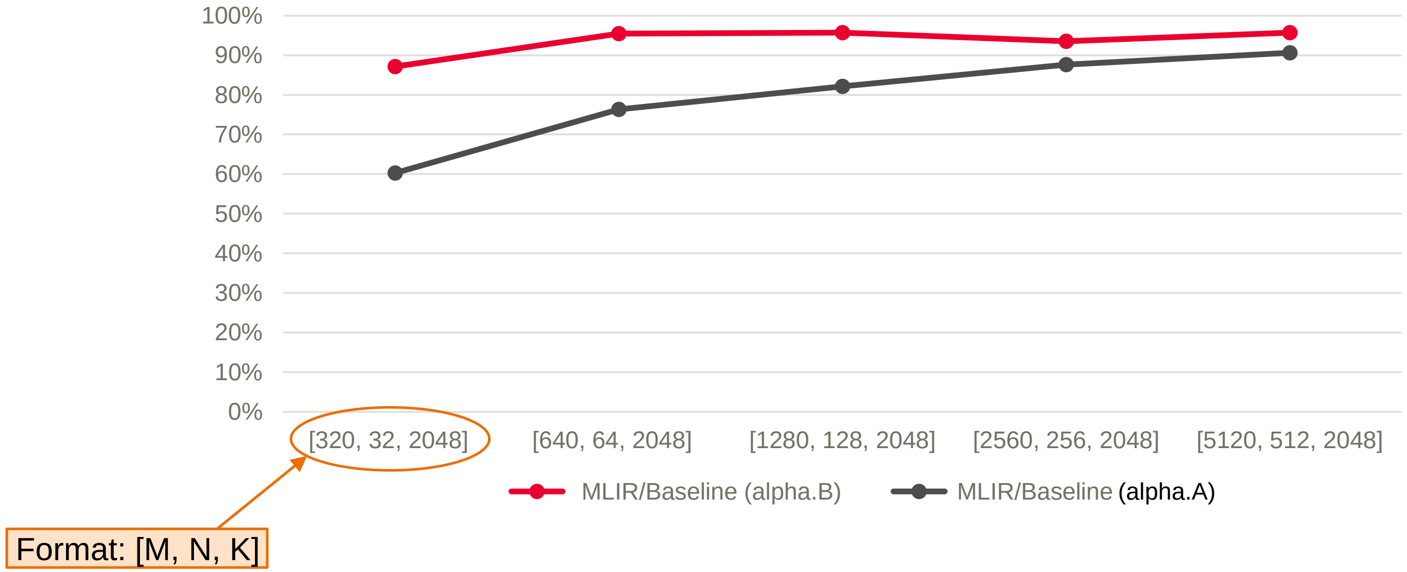
Optimisations (1)

Several optimizations have been implemented at various levels of the pipeline in order to increase performance/functionalities, such as:

High-level optimizations at linalg level:

> Dimensions of A: MxK, dimensions of B: KxN, dimensions of C: MxN

> When N<M: reordering $C = \underbrace{(\alpha A)}_{MxK}B + \beta C$ into $C = A\underbrace{(\alpha B)}_{KxN} + \beta C$ can improve performance:



Optimisations (2)

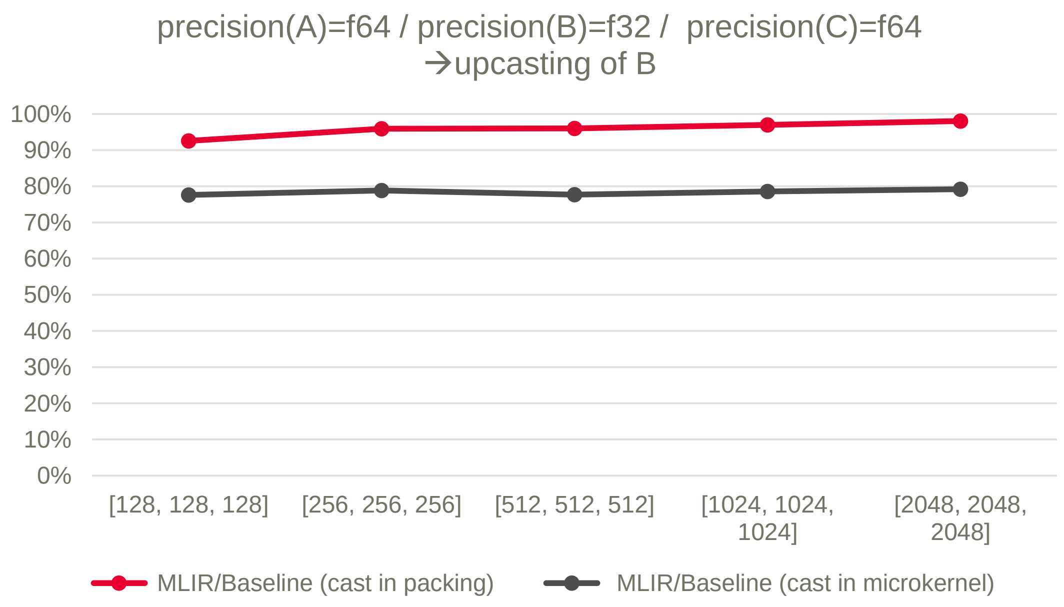
Support for extensions of BLAS and new transformations:

Example: supporting mixed-precision GEMM (i.e. element types of A, B and C can differ).

> Easily enabled in MLIR by injecting truncation/extension ops in the MLIR `linalg.generic` definition.

> Building on a similar transform for transpose operations, we hoist casting ops into the packing loops of the corresponding matrix:

```
for j = 0 to N-1 by steps of nc:
  for p = 0 to K-1 by steps of kc:
    Bc = B(p:p+kc-1, j:j+nc-1) // Pack into Bc
    for i = 0 to M-1 by steps of mc:
      Ac = A(i:i+m-1,p:p+k-1)  // Pack into Ac
      for jj = 0 to nc-1 by steps of nr:
        for ii = 0 to mc-1 by steps of mr:
          for pp = 0 to kc-1 by steps of 1: // Microkernel
            Ac’ = cast(Ac(ii,ii+mr-1,pp)): Ta into Tc
            Bc’ = cast(Bc(pp,jj:jj+nr-1)): Tb into Tc
            C(ii:ii+mr-1, jj:jj+nr-1) += Ac’ * Bc'
```



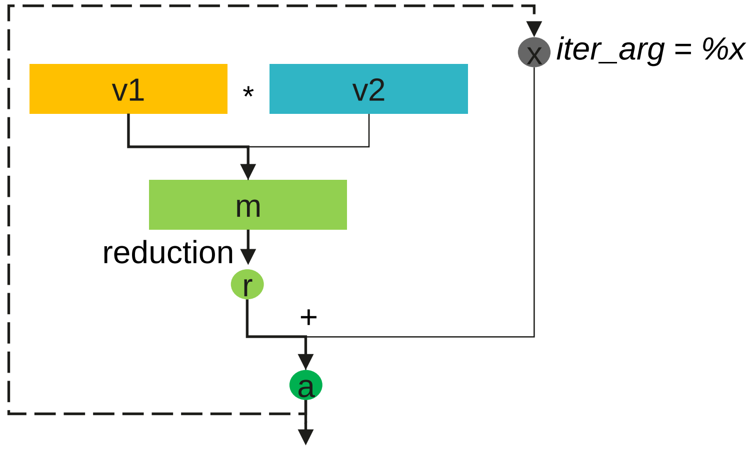
HUAWEI

Optimisations (3)

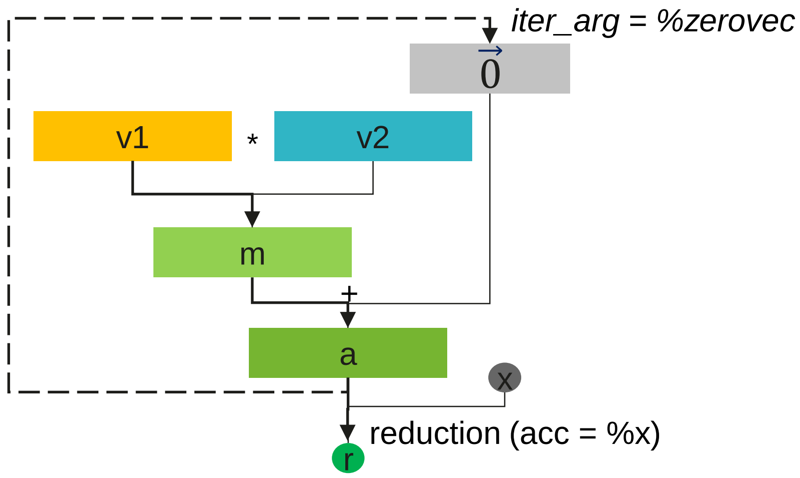
Optimisations of MLIR code:

Example: hoisting of `vector.reduction` outside of loops:

```
%x = (…) : f32
%loop = scf.for %i = %lb to %ub step %step iter_args(%arg = %x) -> f32 {
  %v1 = (…) : vector<32xf32>
  %v2 = (…) : vector<32xf32>
  %m = arith.mulf %v1, %v2 : vector<32xf32>
  %r = vector.reduction <add>, %m : vector<32xf32> into f32
  %a = arith.addf %r, %arg : f32
  scf.yield %a : f32
}
```




```
%x = (…) : f32
%zerovec = arith.constant dense<0.000000e+00> : vector<32xf32>
%loop = scf.for %i = %lb to %ub step %step iter_args(%arg = %zerovec) -> vector<32xf32> {
  %v1 = (…) : vector<32xf32>
  %v2 = (…) : vector<32xf32>
  %m = arith.mulf %v1, %v2 : vector<32xf32>
  %a = arith.addf %m, %arg : vector<32xf32>
  scf.yield %a : vector<32xf32>
}
%r = vector.reduction <add>, %loop, %x : vector<32xf32> into f32
```



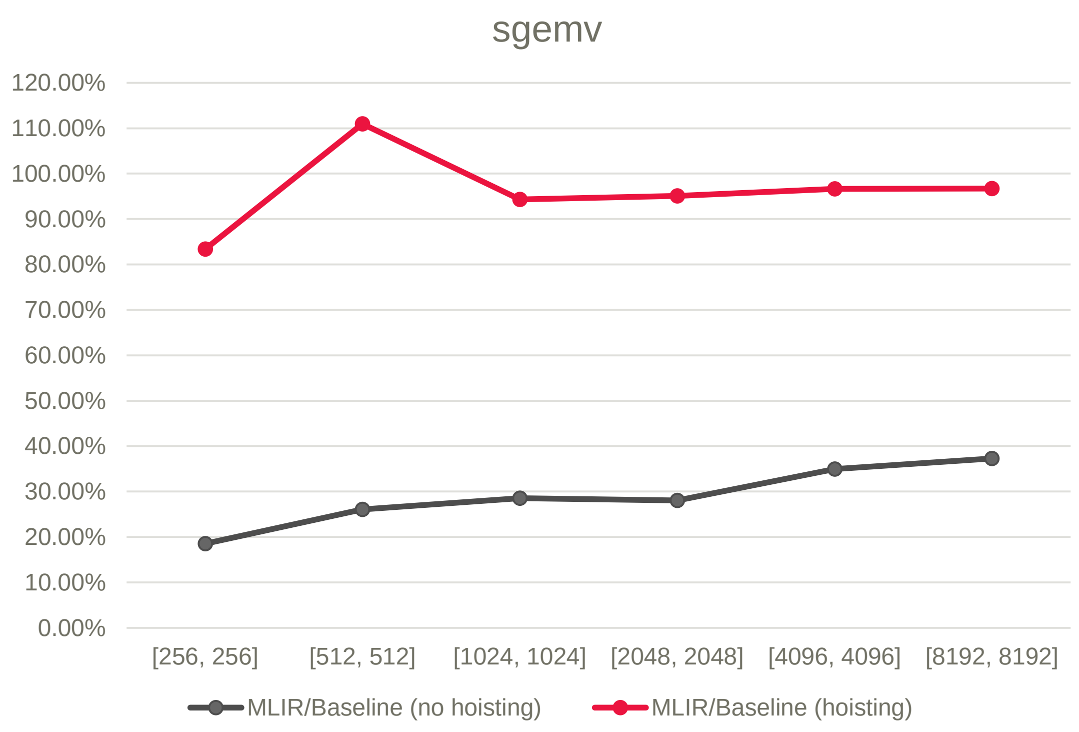
→ This also applies when the accumulator is a vector (using `vector.multi_reduction`)

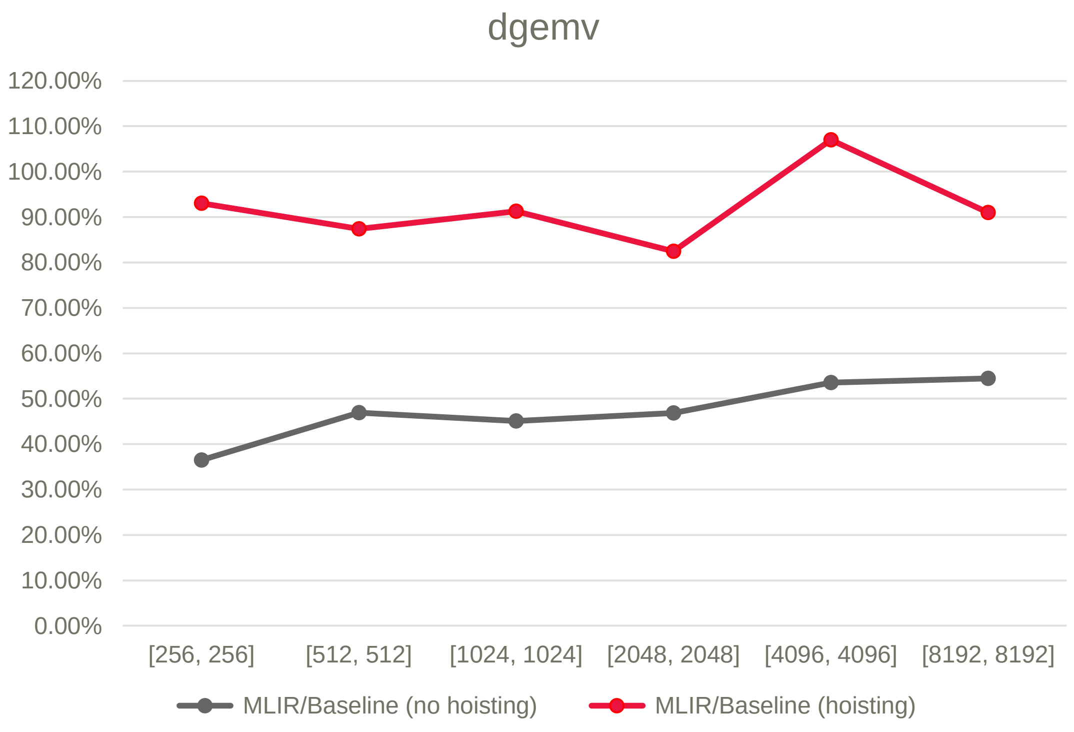
Optimisations (3 – cont.)

Optimisations of MLIR code:

Example: hoisting of `vector.reduction` outside of loops:

→ This optimisation has a significant impact on gemv* (*general matrix-vector* multiplication):





**gemv is* $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$

Handling of Complex Type

To cover the BLAS API, we need to provide operations on complex inputs (*cgemv, cgemm, zgemm*, …)

> High-level definition of the ops in linalg is straightforward: inputs with a `complex<t>` element type.

> However, compiling these operations into efficient code poses some problem:

- Complex tensors are *not* vectorized.
- The complex dialect lowers to extraction functions (`complex.im`, `complex.re`).
- Our current hardware target supports some ARMv8.3-specific complex vector instructions (e.g. `fcmla` – complex multiply and add).

  → We would like to make use of them, instead of splitting complex values.

→ Existing conversion passes gave us less than 1% of KPL's performance for cgemm.

Handling of Complex Type: Conversion into Real

We considered several options to handle complex operations:

> Transform complex GEMM into a series of real GEMM (cf. 3m and 4m methods for cgemm [1]).

- Manual implementation and analysis did not show good performance.
- Prevents use of complex-specific instructions (`fcmla`).
- Not easily extensible to other complex operations.

[1] Van Zee, Field G., and Tyler M. Smith. "Inducing complex matrix multiplication via the 3m and 4m methods FLAME Working Note# 81." (2016).

HUAWEI

Handling of Complex Type: Conversion into Real

We considered several options to handle complex operations:

> Transform complex GEMM into a series of real GEMM (cf. 3m and 4m methods for cgemm [1]).

- Manual implementation and analysis did not show good performance.
- Prevents use of complex-specific instructions (fcmla).
- Not easily extensible to other complex operations.

> Our solution/suggestion (*WIP!*):

- Support vectorization into vectors of complex<*t*>.
- Type conversion of complex ranked types into “doubled” ranked types:

vector<*M*x*N*xcomplex<*t*>> → vector<*M*x*N*x**2**x*t*>

- Extend vector.contraction/outerproduct with kind=<*complexadd*>.
- Enable lowering to fcmla in the backend by creating a new fcmuladd intrinsic.

→ ⚙ D148068 [AArch64] Lower fused complex multiply-add intrinsic to AArch64::FCMA (llvm.org)

[1] Van Zee, Field G., and Tyler M. Smith. "Inducing complex matrix multiplication via the 3m and 4m methods FLAME Working Note# 81." (2016).

Handling of Complex Type: Conversion into Real

Vector operations are updated accordingly:

```
%cst = complex.constant [0.000000e+00 : f32, 0.000000e+00 : f32] : complex<f32>
%v = vector.transfer_read %t[%c0, %c0], %cst : tensor<?x1xcomplex<f32>>, vector<8x1xcomplex<f32>>
%vt = vector.transpose %v, [1, 0] : vector<8x1xcomplex<f32>> to vector<1x8xcomplex<f32>>
%t3 = vector.transfer_write %vt, %t2[%x, %y, %c0, %c0] : vector<1x8xcomplex<f32>>, tensor<?x?x1x8xcomplex<f32>>
```

⬇

```
%cst = arith.constant 0.000000e+00 : f32
%v = vector.transfer_read %t[%c0, %c0, %c0], %cst : tensor<?x1x2xf32>, vector<8x1x2xf32>
%vt = vector.transpose %v, [1, 0, 2] : vector<8x1x2xf32> to vector<1x8x2xf32>
%t3 = vector.transfer_write %vt, %t2[%x, %y, %c0, %c0, %c0] : vector<1x8x2xf32>, tensor<?x?x1x8x2xf32>
```

Handling of Complex Type: Conversion into Real

Contraction is done with last two dimensions “flattened”:

`vector<MxNx2xt>` → `vector<Mx2Nxt>`

→ Prevents splitting between real and imaginary values when lowering vectors.

→ Adapted to the input expected by ARMv8.3 `fcmla`: interleaved real and imaginary parts.

```
%v = vector.contract {(…), kind = #vector.kind<complexadd>} %a, %b, %c : vector<1x8xcomplex<f32>>,
vector<1x4xcomplex<f32>> into vector<8x4xcomplex<f32>>
```

```
%a1 = vector.shape_cast %a : vector<1x8x2xf32> to vector<1x16xf32>
%b1 = vector.shape_cast %b : vector<1x4x2xf32> to vector<1x8xf32>
%c1 = vector.shape_cast %c : vector<8x4x2xf32> to vector<8x8xf32>
%v0 = vector.contract {(…), kind = #vector.kind<complexadd>} %a1, %b1, %c1 : vector<1x16xf32>,
vector<1x8xf32> into vector<8x8xf32>
%v = vector.shape_cast %v0 : vector<8x8xf32> to vector<8x4x2xf32>
```

Optimisations for Complex Pipeline (1)

Hoisting of vector.shape_cast operations outside of loops:

```
%loop = scf.for %i = %lb to %ub step %step iter_args(%arg = %v) -> (vector<4x4x2xf32>) {
  %c = vector.shape_cast %arg : vector<4x4x2xf32> to vector<4x8xf32>
  %w = (…) : vector<4x8xf32> // use of %c
  %r = vector.shape_cast %w : vector<4x8xf32> to vector<4x4x2xf32>
  scf.yield %r: vector<4x4x2xf32>
}
```

```
%c = vector.shape_cast %v : vector<4x4x2xf32> to vector<4x8xf32>
%loop0 = scf.for %i = %lb to %ub step %step iter_args(%arg = %c) -> (vector<4x8xf32>) {
  %w = (…) : vector<4x8xf32> // use of %c (unchanged)
  scf.yield %w: vector<4x8xf32>
}
%loop = vector.shape_cast %loop0 : vector<4x8xf32> to vector<4x4x2xf32>
```

→ This transformation moves `vector.shape_cast` operations out of the microkernel loop.

Optimisations for Complex Pipeline (2)

"Lifting" `vector.transfer_read+vector.shape_cast` **to** `tensor.collapse_shape+vector.transfer_read`**:**

```
%0 = vector.transfer_read %arg0[%c0, %c0, %c0], %arg1 : tensor<1x4x2xf32>, vector<1x4x2xf32>
%1 = vector.shape_cast %0 : vector<1x4x2xf32> to vector<1x8xf32>
```

```
%0 = tensor.collapse_shape %arg0 [[0], [1, 2]] : tensor<1x4x2xf32> into tensor<1x8xf32>
%1 = vector.transfer_read %0 [%c0, %c0], %arg1 : tensor<1x8xf32>, vector<1x8xf32>
```

> A similar transformation replaces `shape_cast+transfer_write` with `transfer_write+expand_shape`.

→ Significant performance improvement (>+50%), as `tensor.collapse/expand_shape` does not involve data copy, unlike `vector.shape_cast`.

Handling of Complex Types: Limitations

Genericity:

- Conversion assumes that the complex type layout fits with complex<*t*> → 2x*t*.
- Heavily targeted towards specific hardware with specific instructions for complex type (fcmla).

Interface changes:

- A function taking in a vector<8xcomplex<f32>> now takes in vector<8x2xf32>.

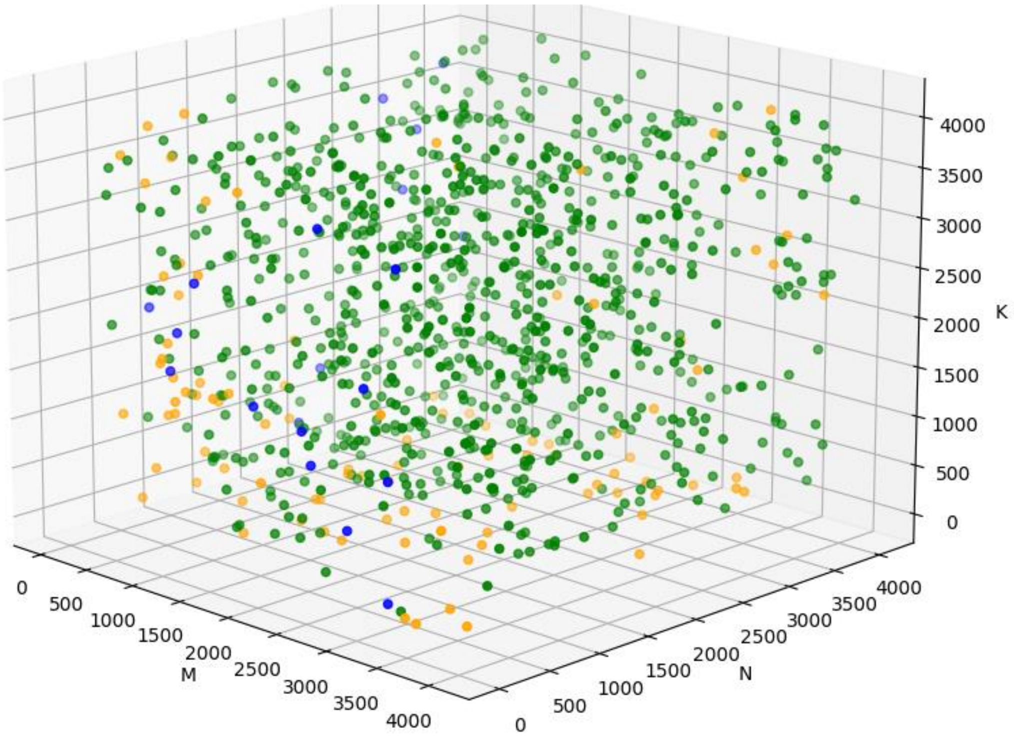
→ We use special wrappers at the interface with the packager.

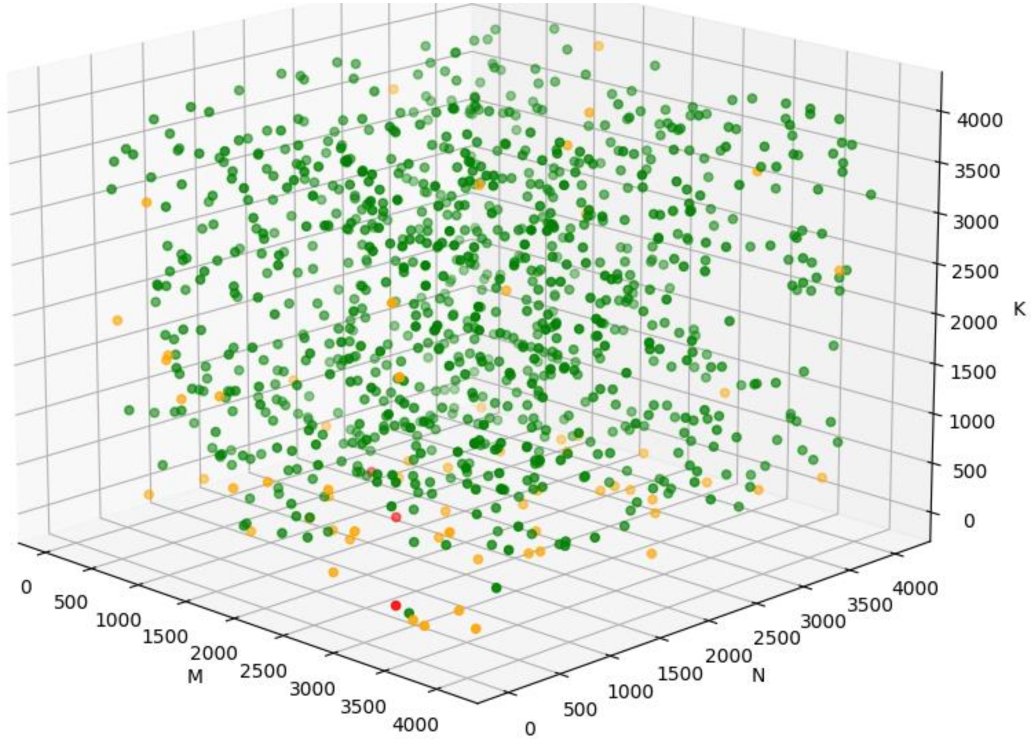
→ Working on extending the complex dialect with casting operations complex<*t*> → 2x*t* and 2x*t*→complex<*t*>.

Results: Performance vs KPL (Real GEMM)

Running sgemm/dgemm on Huawei Kunpeng 920, 1000 random points:

sgemm





dgemm

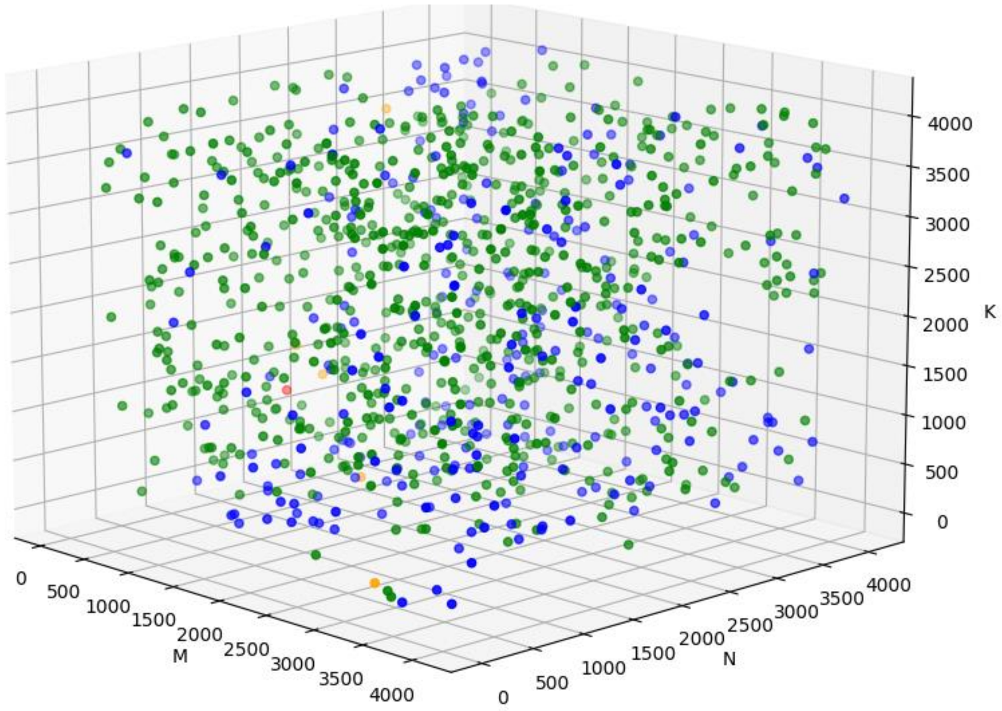
Colour (value)	sgemm (single precision)	dgemm (double precision)
Red (0% - 49% of KPL)	None	0.3% of points
Orange (50% - 89% of KPL)	5.6% of points	4.7% of points
Green (90% - 99% of KPL)	**92.4% of points**	**95% of points**
Blue (≥100% of KPL)	**2% of points**	None

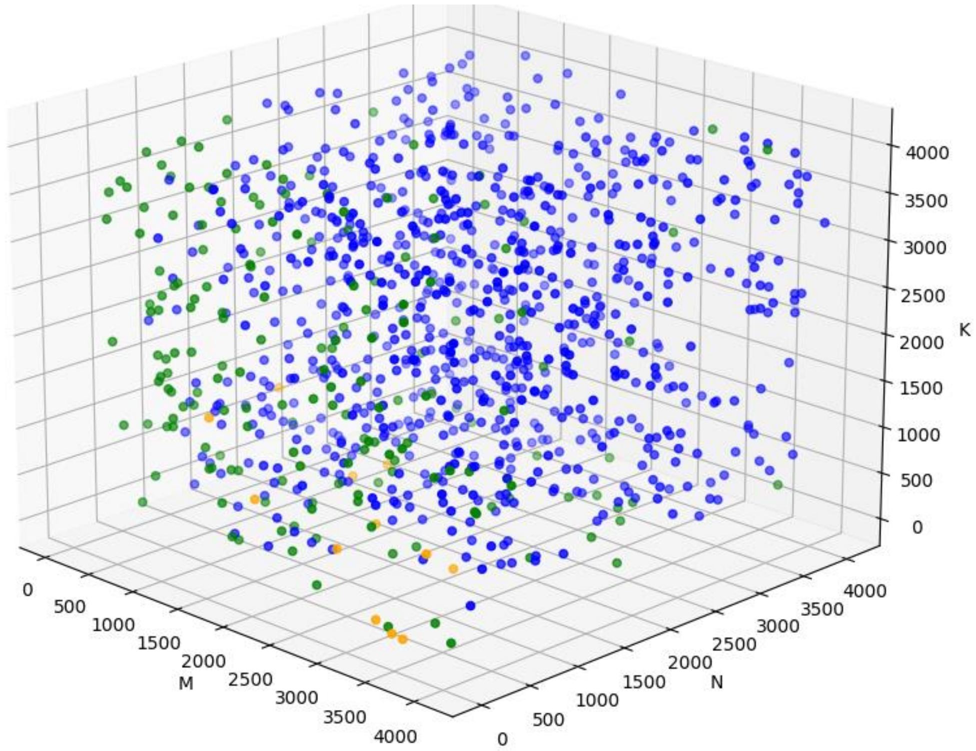
HUAWEI

Results: Performance vs KPL (Complex GEMM)

Running cgemm/zgemm on Huawei Kunpeng 920, 1000 random points:

cgemm





zgemm

Colour (value)	cgemm (single precision)	zgemm (double precision)
Red (0% - 49% of KPL)	0.1% of points	None
Orange (50% - 89% of KPL)	0.4% of points	1% of points
Green (90% - 99% of KPL)	**72.5% of points**	**18.2% of points**
Blue (≥100% of KPL)	**27% of points**	**80.8% of points**

HUAWEI

Conclusion & Future Work

We have leveraged the functionalities of the MLIR framework to:

> Build a full pipeline to generate optimized functions of a BLAS library.

> Use a multi-kernel approach able to dynamically adapt to specific inputs.

> Provide optimizations to achieve results competing with hand-written assembly code.

Ongoing/future work:

> Connect to a DSL (ALP[1]) that would lower to MLIR and use our pipeline.

→ move beyond simply building a library

> Fuse operations to improve performance (some promising results for GEMM already).

> Enable parallelism for a multithread version of the library.

> Target more diverse hardware.

[1] Algebraic Programming @ https://algebraic-programming.github.io

HUAWEI

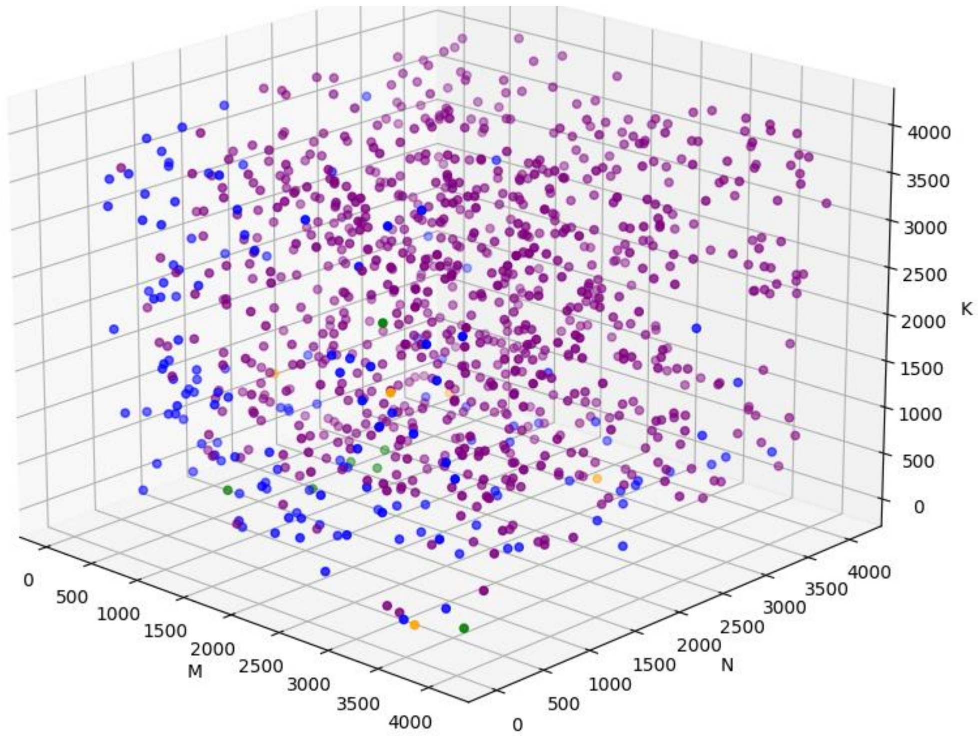
Thank you.

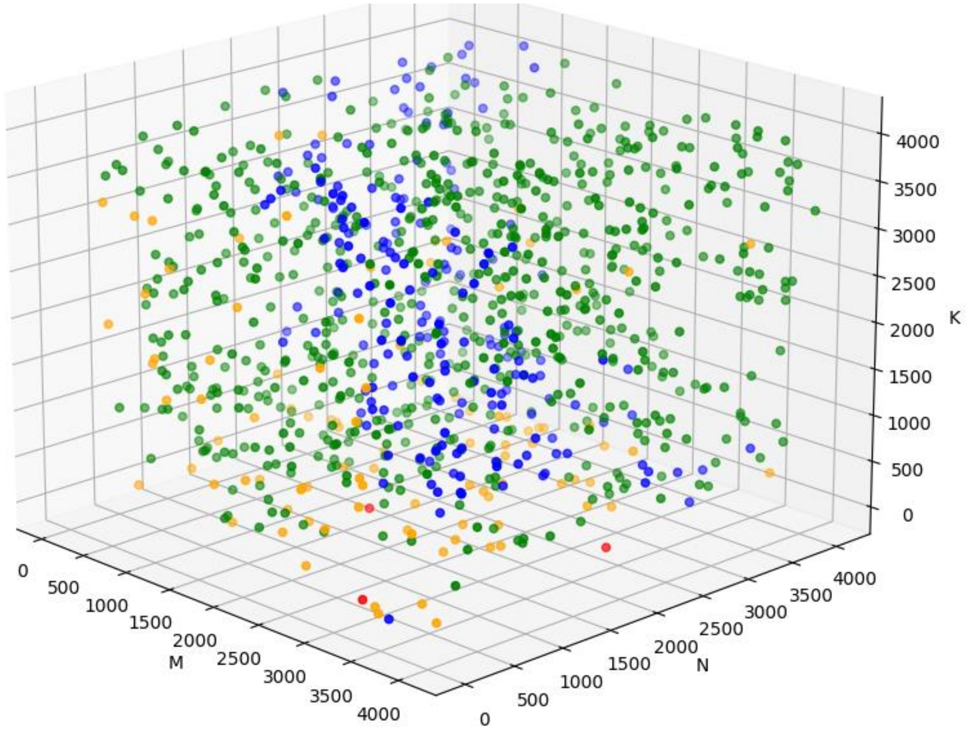
Backup Slides

Results: Performance vs OpenBLAS

Running sgemm/dgemm on Huawei Kunpeng 920, 1000 random points:

sgemm





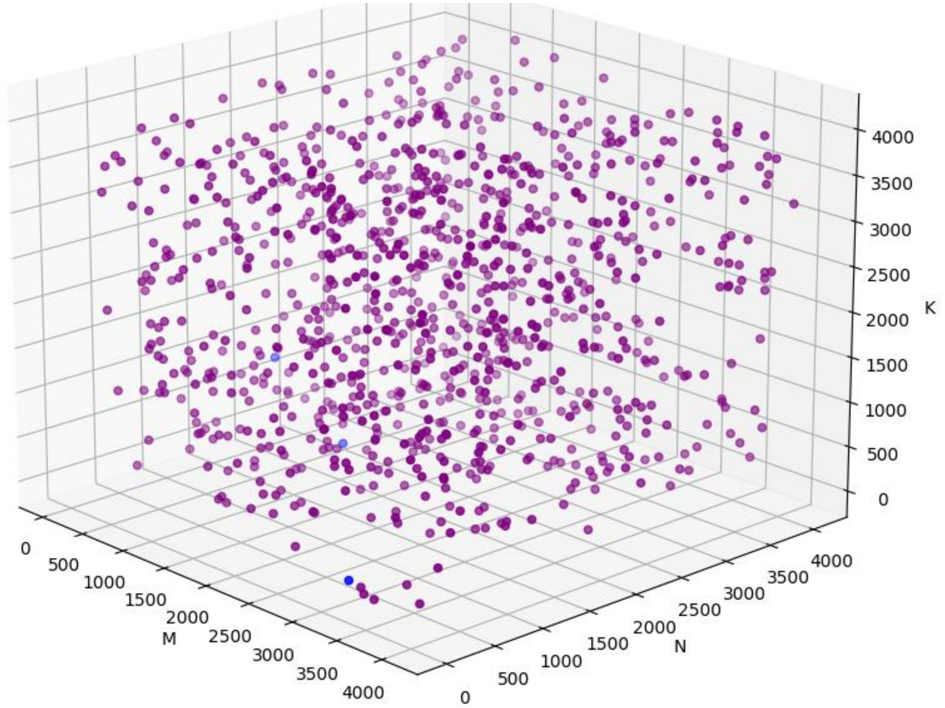
dgemm

Colour (value)	sgemm	dgemm
Red (0% - 49% of OpenBLAS)	None	0.3% of points
Orange (50% - 89% of OpenBLAS)	0.5% of points	7% of points
Green (90% - 99% of OpenBLAS)	**0.7% of points**	**72.4% of points**
Blue (100%-124% of OpenBLAS)	**15.0% of points**	**20.3% of points**
Purple (≥125% of OpenBLAS)	**83.8% of points**	None

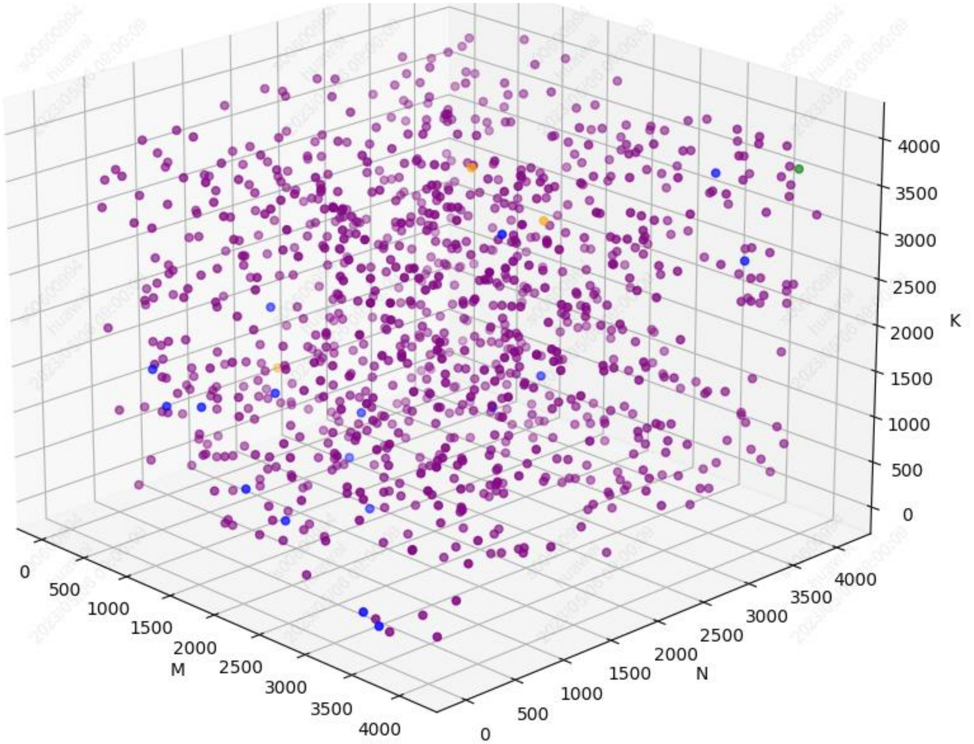
Results: Performance vs OpenBLAS (complex gemm)

Results of cgemm/zgemm (1000 random points):

cgemm

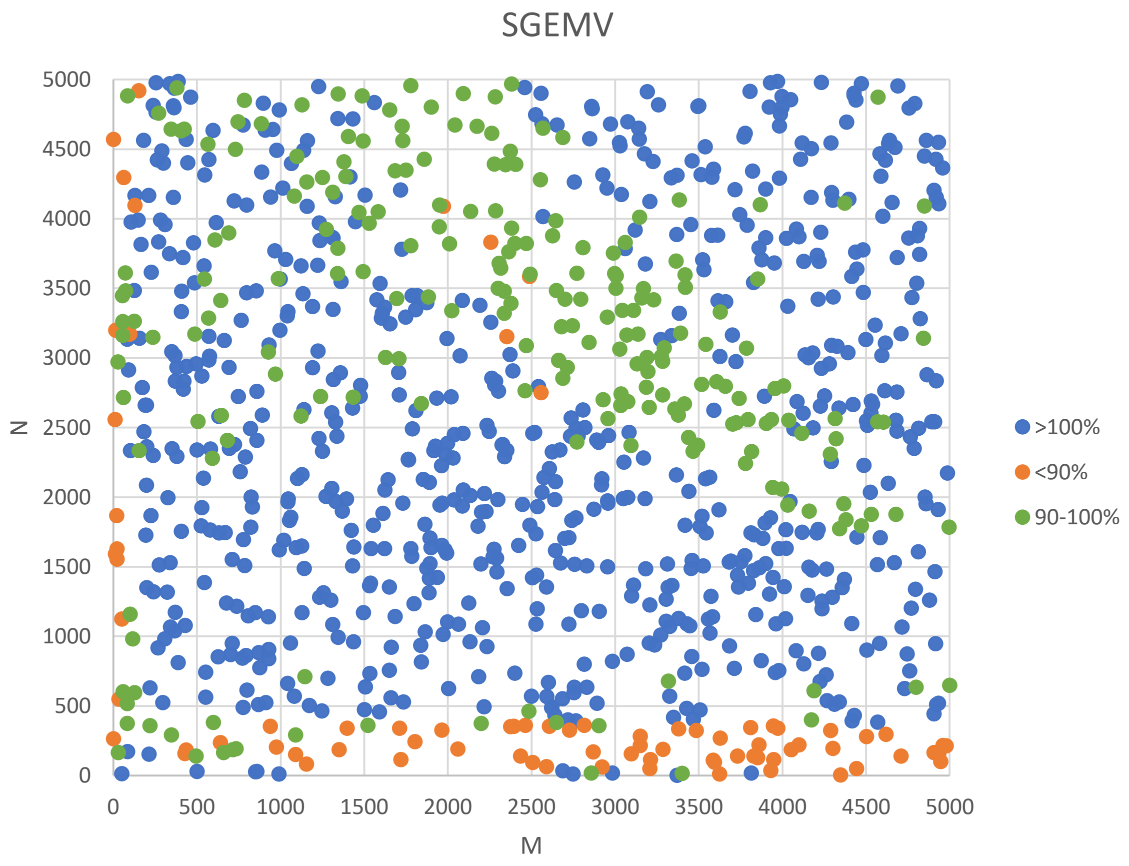


zgemm

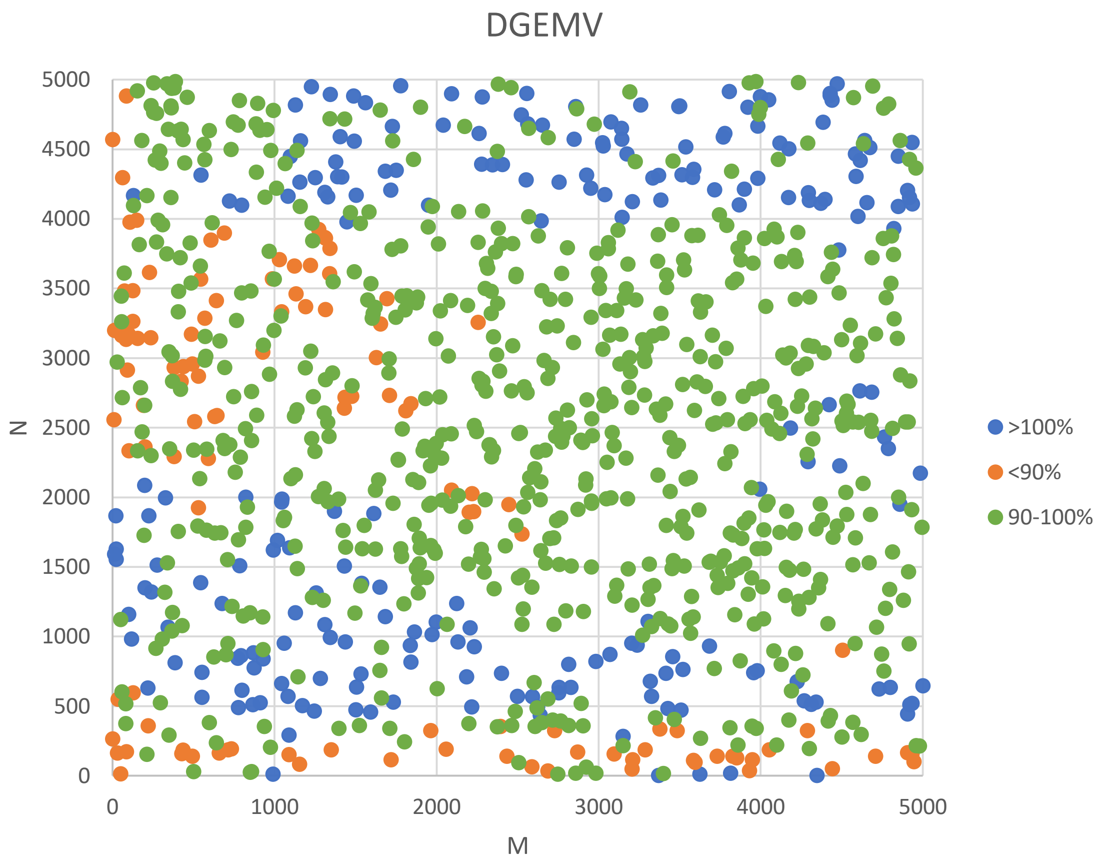


Colour (value)	cgemm	zgemm
Orange (50% - 89% of OpenBLAS)	None	0.3% of points
Green (90% - 99% of OpenBLAS)	None	0.1% of points
Blue (100% - 124% of OpenBLAS)	**0.3% of points**	**1.7% of points**
Purple (≥125% of OpenBLAS)	**99.7% of points**	**97.9% of points**

Results: gemv



92.1% of points >= 90% of KPL



88.7% of points >= 90% of KPL

HUAWEI