Polyhedral Rescheduling of GPU Kernels To Exploit Async Memory Movement

Ivan R. Ivanov,
William Moses, Emil Vatai, Toshio Endo, Jens Domke, Alex Zinenko

Background

Advanced features in GPUs are underused

- Legacy code
- Code not written by GPU experts
- Untuned code
- Synthesized code

GPU features

Async global->shared copies

GPU features

Faster reductions

Tensor cores

TMA

GPU features

More memory movement features

Common theme

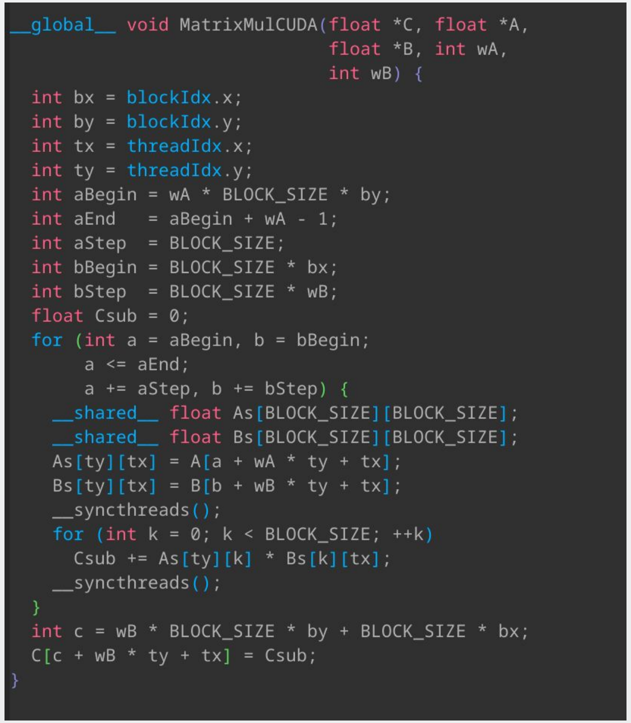
- A lot of async memory movement
- Specialized hardware for memory and computation (TMA and Tensor cores)

...

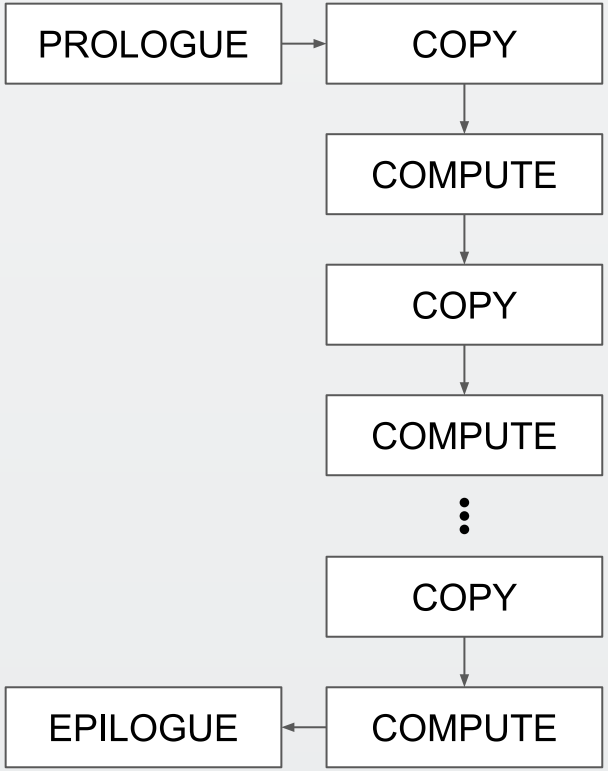
Need to overlap them and stress them to make **full** use of your modern GPU

Example usage

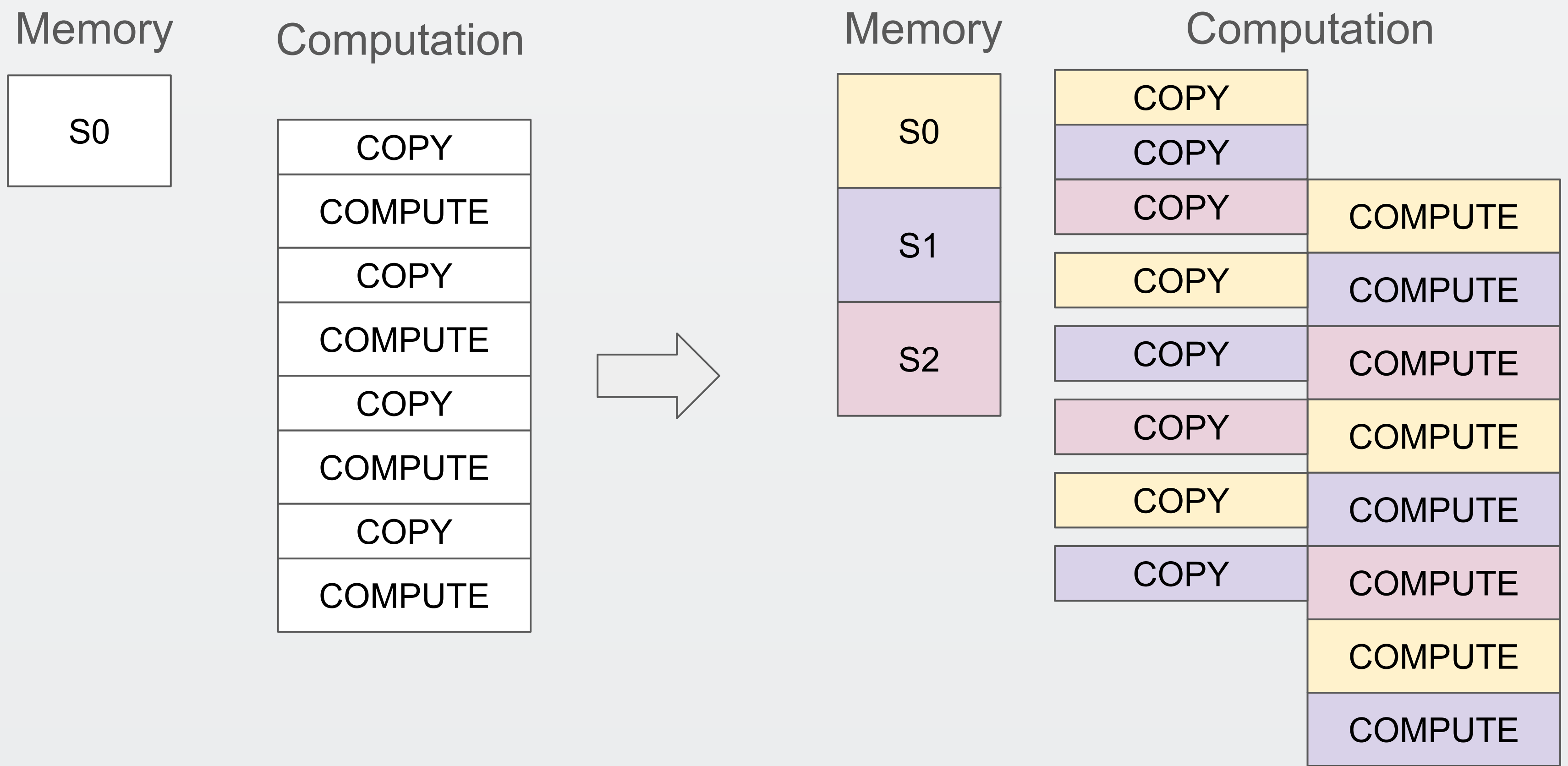
Blocked matmul



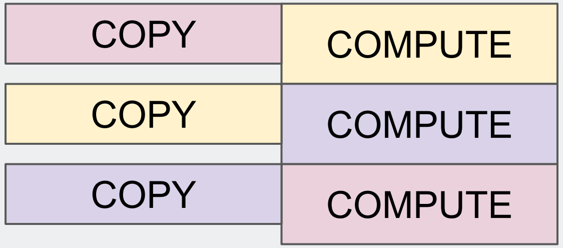
```
__global__ void MatrixMulCUDA(float *C, float *A,
                              float *B, int wA,
                              int wB) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep  = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep  = BLOCK_SIZE * wB;
  float Csub = 0;
  for (int a = aBegin, b = bBegin;
       a <= aEnd;
       a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();
    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```



Optimized matmul - pipelining



Pipelining using nvvm intrinsics



```
async.cp global[i] -> shared[j]
...
commit_group
wait_group 2

compute using shared[...]
```

Can we automatically optimize existing code to use these features?

Polyhedral model

Polyhedral model

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < i; j++) {
    S(i, j);
    R(i, j);
  }
}
```

```
domain

{ S(i, j) : 0 <= i < N,
            0 <= j < i }
{ R(i, j) : 0 <= i < N,
            0 <= j < i }
```

```
dependencies

{ S(si, sj) -> R(ri, rj) :
    si = ri, sj = rj }
```

Everything is represented using linear algebra

Existing work on polyhedral representation

- Extracting the polyhedral structure
  - Source code - pet
  - Intermediate representation - polly (LLVM), polygeist+polymer(MLIR)

What about synchronisation?

-> No prior work in importing code with synchronisation into a polyhedral representation.

It is essential to be able to handle barriers for GPU code (extremely prevalent)

Existing work on polyhedral scheduling

- CPU/Generic: polly, isl, pluto, etc
- GPU: ppcg

Parallelism **Supported**

But everything assumes synchronous execution *within* a thread of execution

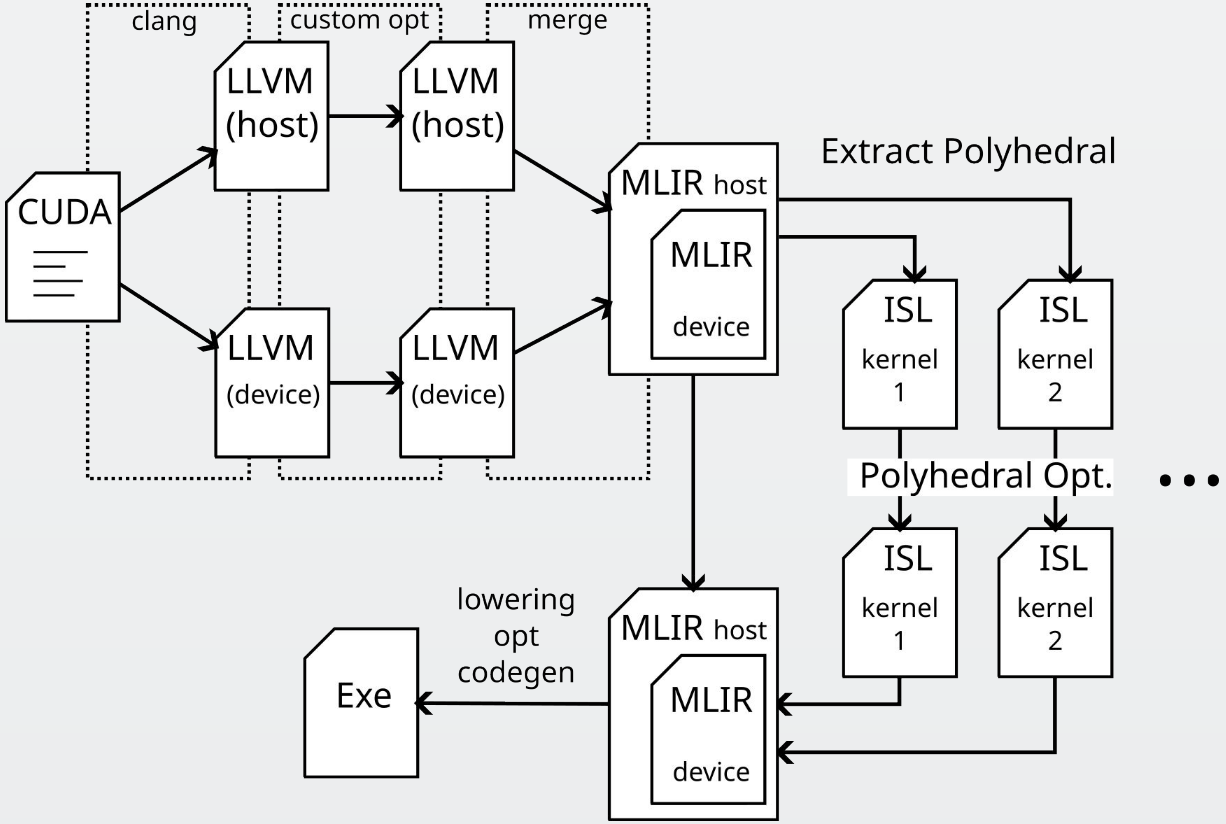
What about asynchronous execution? e.g. **nvvm.async.cp**

Our approach

Hydra: The pipeline

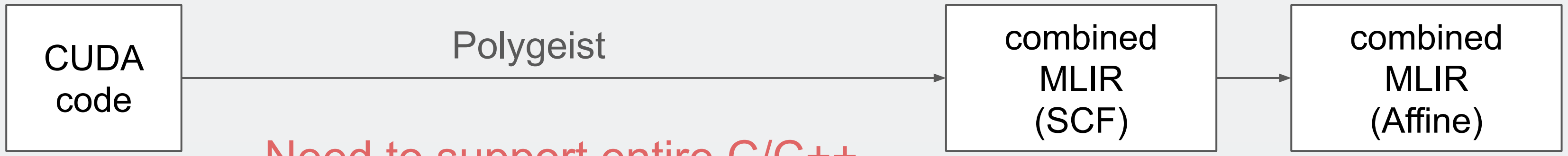
We need to get the polyhedral representation of a kernel...

- The information is split
  - In the host code (the launch configuration: grid, block dims, shared mem size)
  - In the device code (the actual kernel computation)



Comparison to earlier work

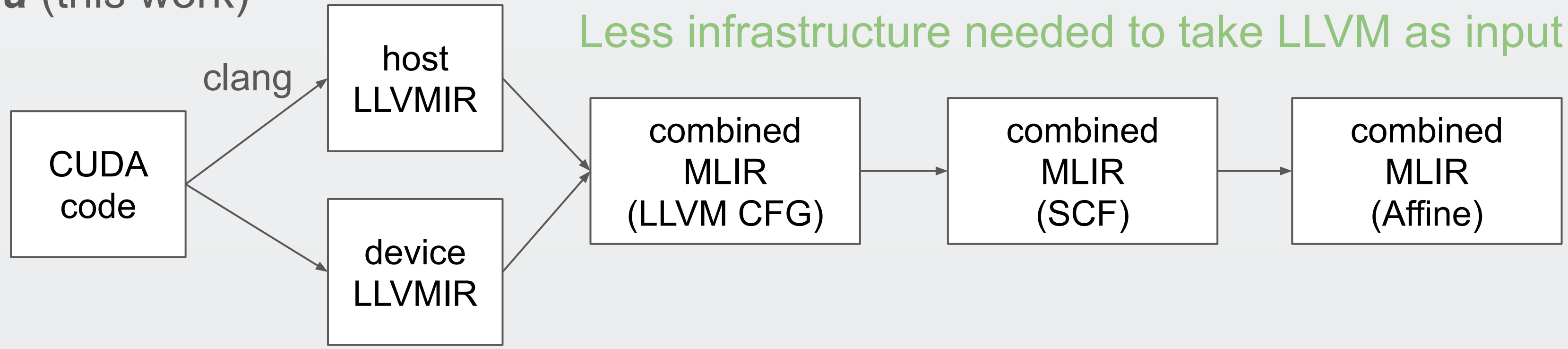
Early work



Need to support entire C/C++

Hydra (this work)

Less infrastructure needed to take LLVM as input



Compiling to LLVM-IR

```
#define BLOCK_SIZE 16

__global__ void MatrixMulCUDA(float *C, float *A,
                              float *B, int wA,
                              int wB) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep  = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep  = BLOCK_SIZE * wB;
  float Csub = 0;
  for (int a = aBegin, b = bBegin;
       a <= aEnd;
       a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();
    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}

int MatrixMultiply(const dim3 &dimsA,
                   const dim3 &dimsB,
                   float *A, float *B, float *C) {
  // Setup execution parameters
  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
  MatrixMulCUDA<<<grid, threads>>>(C, A, B, dimsA.x, dimsB.x);
}
```

`clang++ --cuda-device` ⟹

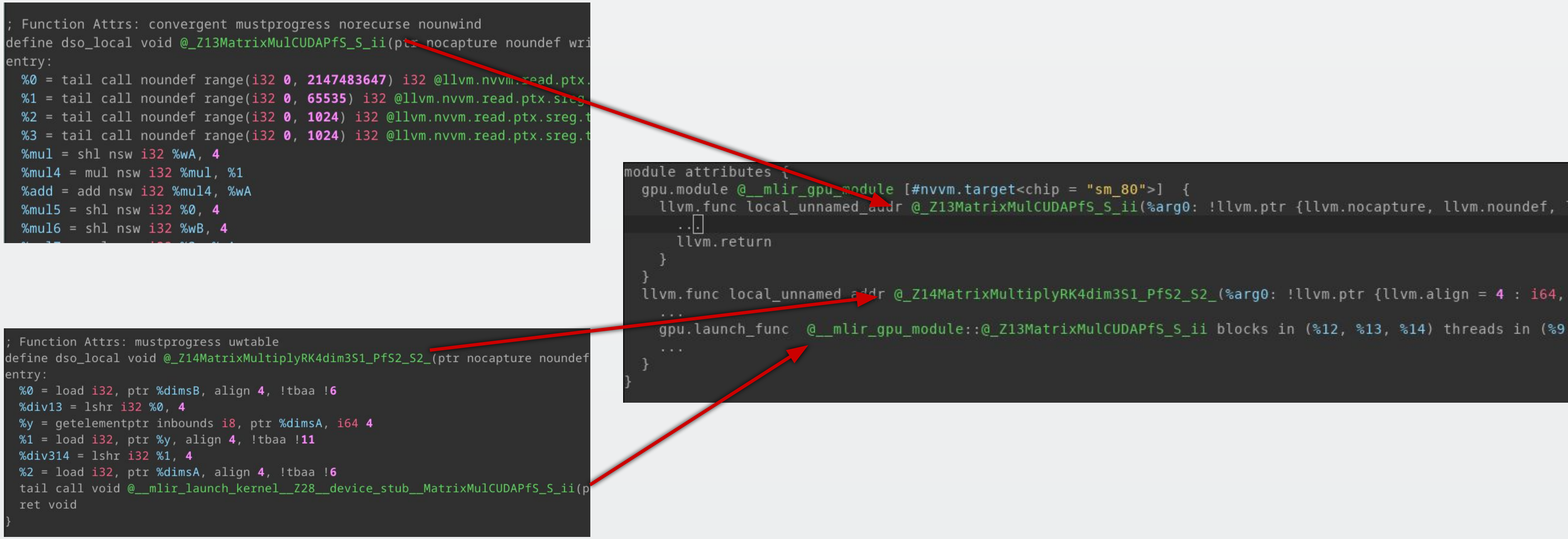
```
; Function Attrs: convergent mustprogress norecurse nounwind
define dso_local void @_Z13MatrixMulCUDAPfS_S_ii(ptr nocapture noundef wri
entry:
  %0 = tail call noundef range(i32 0, 2147483647) i32 @llvm.nvvm.read.ptx.
  %1 = tail call noundef range(i32 0, 65535) i32 @llvm.nvvm.read.ptx.sreg.
  %2 = tail call noundef range(i32 0, 1024) i32 @llvm.nvvm.read.ptx.sreg.t
  %3 = tail call noundef range(i32 0, 1024) i32 @llvm.nvvm.read.ptx.sreg.t
  %mul = shl nsw i32 %wA, 4
  %mul4 = mul nsw i32 %mul, %1
  %add = add nsw i32 %mul4, %wA
  %mul5 = shl nsw i32 %0, 4
  %mul6 = shl nsw i32 %wB, 4
```

`clang++ --cuda-host` ⟹

```
; Function Attrs: mustprogress uwtable
define dso_local void @_Z14MatrixMultiplyRK4dim3S1_PfS2_S2_(ptr nocapture noundef
entry:
  %0 = load i32, ptr %dimsB, align 4, !tbaa !6
  %div13 = lshr i32 %0, 4
  %y = getelementptr inbounds i8, ptr %dimsA, i64 4
  %1 = load i32, ptr %y, align 4, !tbaa !11
  %div314 = lshr i32 %1, 4
  %2 = load i32, ptr %dimsA, align 4, !tbaa !6
  tail call void @__mlir_launch_kernel__Z28__device_stub__MatrixMulCUDAPfS_S_ii(p
  ret void
}
```

Patched clang to preserve launch information better

Merging the GPU modules

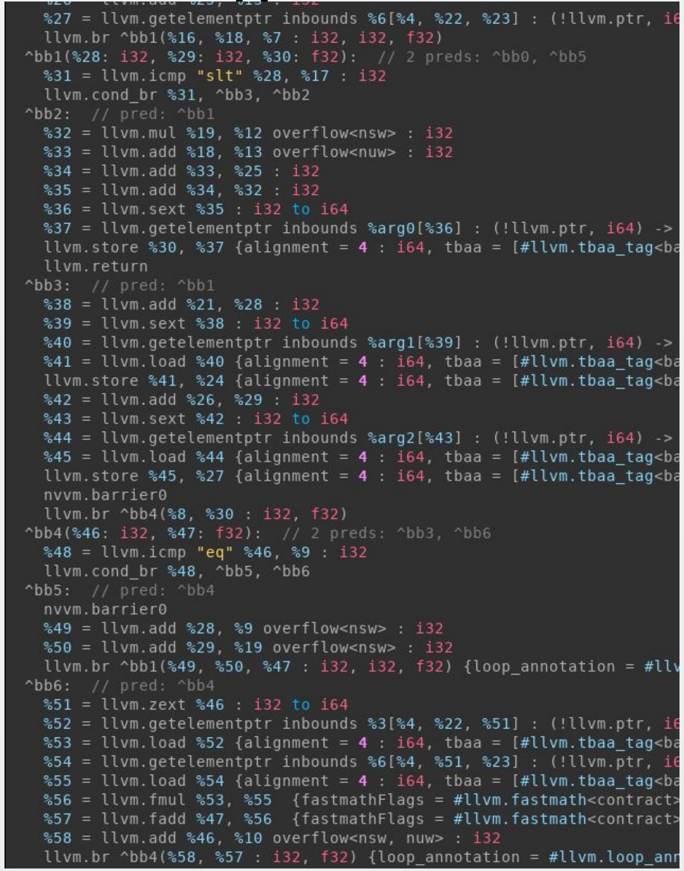


```
; Function Attrs: convergent mustprogress norecurse nounwind
define dso_local void @_Z13MatrixMulCUDAPfS_S_ii(ptr nocapture noundef wri
entry:
  %0 = tail call noundef range(i32 0, 2147483647) i32 @llvm.nvvm.read.ptx.
  %1 = tail call noundef range(i32 0, 65535) i32 @llvm.nvvm.read.ptx.sreg
  %2 = tail call noundef range(i32 0, 1024) i32 @llvm.nvvm.read.ptx.sreg.t
  %3 = tail call noundef range(i32 0, 1024) i32 @llvm.nvvm.read.ptx.sreg.t
  %mul = shl nsw i32 %wA, 4
  %mul4 = mul nsw i32 %mul, %1
  %add = add nsw i32 %mul4, %wA
  %mul5 = shl nsw i32 %0, 4
  %mul6 = shl nsw i32 %wB, 4
```

```
module attributes {
  gpu.module @__mlir_gpu_module [#nvvm.target<chip = "sm_80">]  {
    llvm.func local_unnamed_addr @_Z13MatrixMulCUDAPfS_S_ii(%arg0: !llvm.ptr {llvm.nocapture, llvm.noundef,
      ...
      llvm.return
    }
  }
  llvm.func local_unnamed_addr @_Z14MatrixMultiplyRK4dim3S1_PfS2_S2_(%arg0: !llvm.ptr {llvm.align = 4 : i64,
    ...
    gpu.launch_func  @__mlir_gpu_module::@_Z13MatrixMulCUDAPfS_S_ii blocks in (%12, %13, %14) threads in (%9
    ...
  }
}
```

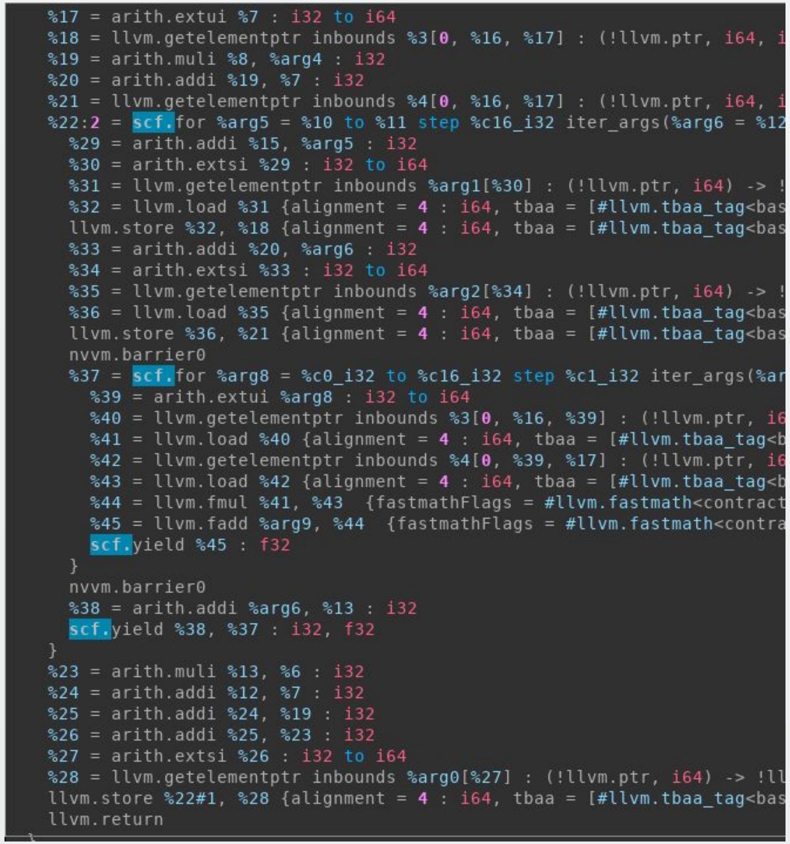
```
; Function Attrs: mustprogress uwtable
define dso_local void @_Z14MatrixMultiplyRK4dim3S1_PfS2_S2_(ptr nocapture noundef
entry:
  %0 = load i32, ptr %dimsB, align 4, !tbaa !6
  %div13 = lshr i32 %0, 4
  %y = getelementptr inbounds i8, ptr %dimsA, i64 4
  %1 = load i32, ptr %y, align 4, !tbaa !11
  %div314 = lshr i32 %1, 4
  %2 = load i32, ptr %dimsA, align 4, !tbaa !6
  tail call void @__mlir_launch_kernel__Z28__device_stub__MatrixMulCUDAPfS_S_ii(p
  ret void
}
```

The raising pipeline: finding control flow structure



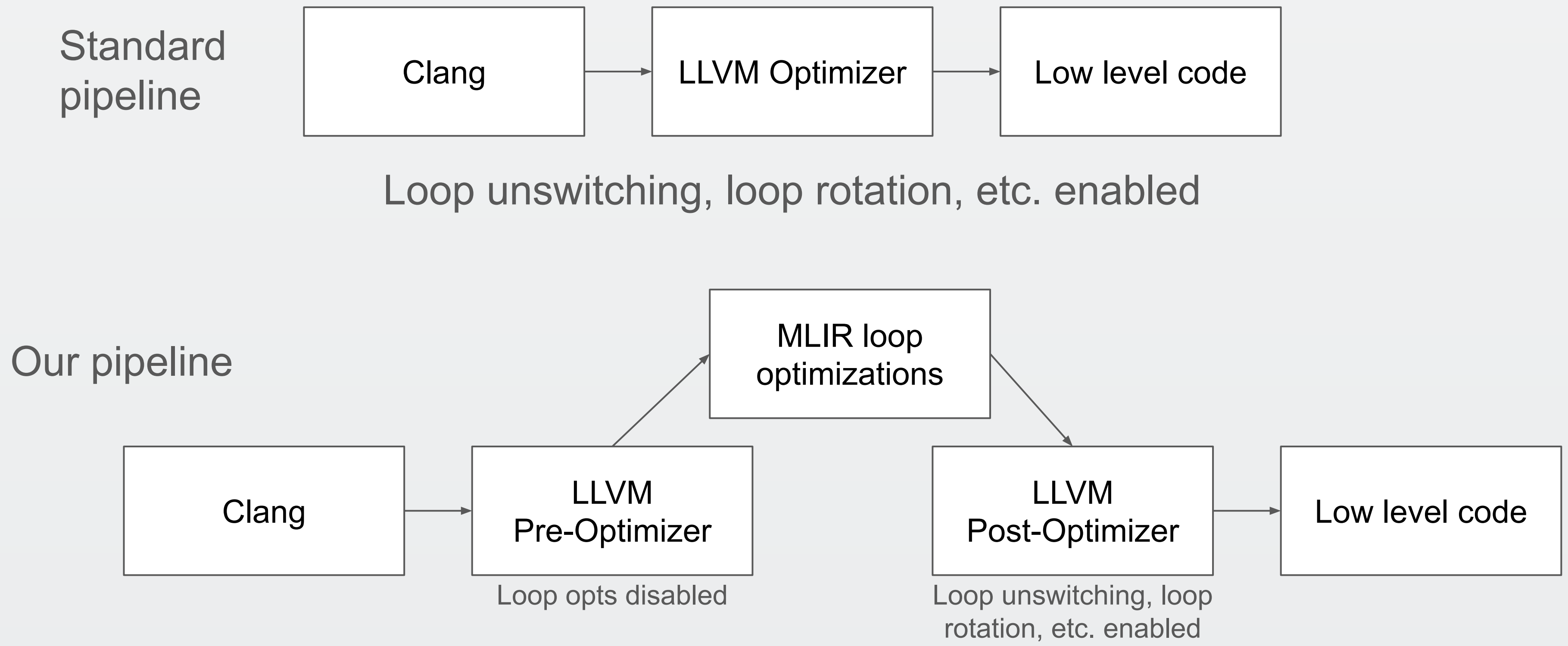
```
  %27 = llvm.getelementptr inbounds %6[%4, %22, %23] : (!llvm.ptr, i6
  llvm.br ^bb1(%16, %18, %7 : i32, i32, f32)
^bb1(%28: i32, %29: i32, %30: f32):  // 2 preds: ^bb0, ^bb5
  %31 = llvm.icmp "slt" %28, %17 : i32
  llvm.cond_br %31, ^bb3, ^bb2
^bb2:  // pred: ^bb1
  %32 = llvm.mul %19, %12 overflow<nsw> : i32
  %33 = llvm.add %18, %13 overflow<nuw> : i32
  %34 = llvm.add %33, %25 : i32
  %35 = llvm.add %34, %32 : i32
  %36 = llvm.sext %35 : i32 to i64
  %37 = llvm.getelementptr inbounds %arg0[%36] : (!llvm.ptr, i64) ->
  llvm.store %30, %37 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  llvm.return
^bb3:  // pred: ^bb1
  %38 = llvm.add %21, %28 : i32
  %39 = llvm.sext %38 : i32 to i64
  %40 = llvm.getelementptr inbounds %arg1[%39] : (!llvm.ptr, i64) ->
  %41 = llvm.load %40 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  llvm.store %41, %24 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  %42 = llvm.add %26, %29 : i32
  %43 = llvm.sext %42 : i32 to i64
  %44 = llvm.getelementptr inbounds %arg2[%43] : (!llvm.ptr, i64) ->
  %45 = llvm.load %44 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  llvm.store %45, %27 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  nvvm.barrier0
  llvm.br ^bb4(%8, %30 : i32, f32)
^bb4(%46: i32, %47: f32):  // 2 preds: ^bb3, ^bb6
  %48 = llvm.icmp "eq" %46, %9 : i32
  llvm.cond_br %48, ^bb5, ^bb6
^bb5:  // pred: ^bb4
  nvvm.barrier0
  %49 = llvm.add %28, %9 overflow<nsw> : i32
  %50 = llvm.add %29, %19 overflow<nsw> : i32
  llvm.br ^bb1(%49, %50, %47 : i32, i32, f32) {loop_annotation = #llv
^bb6:  // pred: ^bb4
  %51 = llvm.zext %46 : i32 to i64
  %52 = llvm.getelementptr inbounds %3[%4, %22, %51] : (!llvm.ptr, i6
  %53 = llvm.load %52 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  %54 = llvm.getelementptr inbounds %6[%4, %51, %23] : (!llvm.ptr, i6
  %55 = llvm.load %54 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<ba
  %56 = llvm.fmul %53, %55  {fastmathFlags = #llvm.fastmath<contract>
  %57 = llvm.fadd %47, %56  {fastmathFlags = #llvm.fastmath<contract>
  %58 = llvm.add %46, %10 overflow<nsw, nuw> : i32
  llvm.br ^bb4(%58, %57 : i32, f32) {loop_annotation = #llvm.loop_ann
```

Adapted from upstream numba-mlir

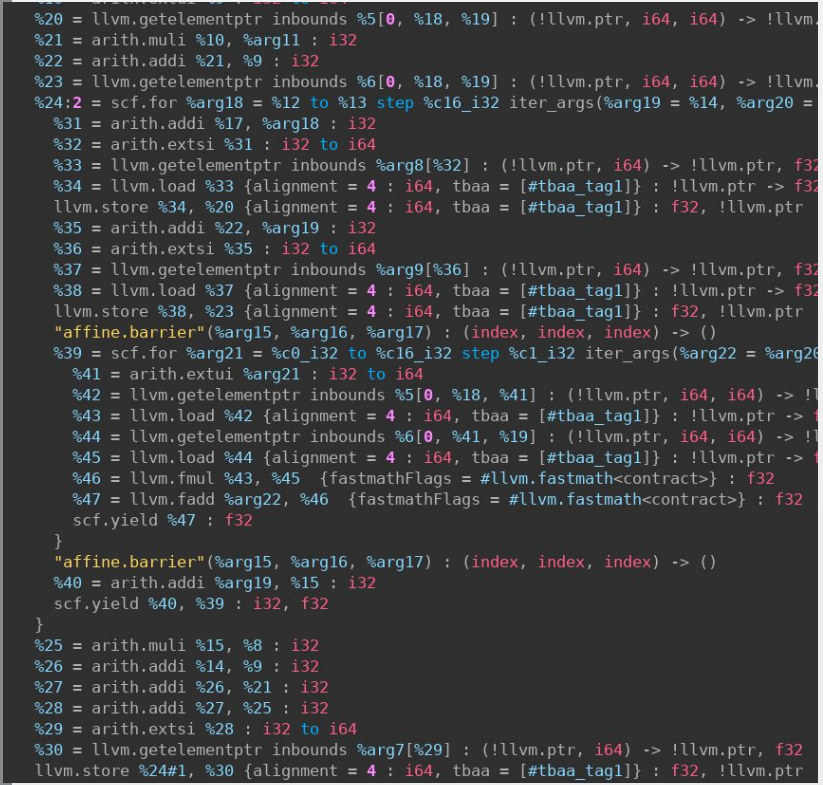


```
  %17 = arith.extui %7 : i32 to i64
  %18 = llvm.getelementptr inbounds %3[0, %16, %17] : (!llvm.ptr, i64, i
  %19 = arith.muli %8, %arg4 : i32
  %20 = arith.addi %19, %7 : i32
  %21 = llvm.getelementptr inbounds %4[0, %16, %17] : (!llvm.ptr, i64, i
  %22:2 = scf.for %arg5 = %10 to %11 step %c16_i32 iter_args(%arg6 = %12
    %29 = arith.addi %15, %arg5 : i32
    %30 = arith.extsi %29 : i32 to i64
    %31 = llvm.getelementptr inbounds %arg1[%30] : (!llvm.ptr, i64) -> !
    %32 = llvm.load %31 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<bas
    llvm.store %32, %18 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<bas
    %33 = arith.addi %20, %arg6 : i32
    %34 = arith.extsi %33 : i32 to i64
    %35 = llvm.getelementptr inbounds %arg2[%34] : (!llvm.ptr, i64) -> !
    %36 = llvm.load %35 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<bas
    llvm.store %36, %21 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<bas
    nvvm.barrier0
    %37 = scf.for %arg8 = %c0_i32 to %c16_i32 step %c1_i32 iter_args(%ar
      %39 = arith.extui %arg8 : i32 to i64
      %40 = llvm.getelementptr inbounds %3[0, %16, %39] : (!llvm.ptr, i6
      %41 = llvm.load %40 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<b
      %42 = llvm.getelementptr inbounds %4[0, %39, %17] : (!llvm.ptr, i6
      %43 = llvm.load %42 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<b
      %44 = llvm.fmul %41, %43  {fastmathFlags = #llvm.fastmath<contract
      %45 = llvm.fadd %arg9, %44  {fastmathFlags = #llvm.fastmath<contra
      scf.yield %45 : f32
    }
    nvvm.barrier0
    %38 = arith.addi %arg6, %13 : i32
    scf.yield %38, %37 : i32, f32
  }
  %23 = arith.muli %13, %6 : i32
  %24 = arith.addi %12, %7 : i32
  %25 = arith.addi %24, %19 : i32
  %26 = arith.addi %25, %23 : i32
  %27 = arith.extsi %26 : i32 to i64
  %28 = llvm.getelementptr inbounds %arg0[%27] : (!llvm.ptr, i64) -> !ll
  llvm.store %22#1, %28 {alignment = 4 : i64, tbaa = [#llvm.tbaa_tag<bas
  llvm.return
```

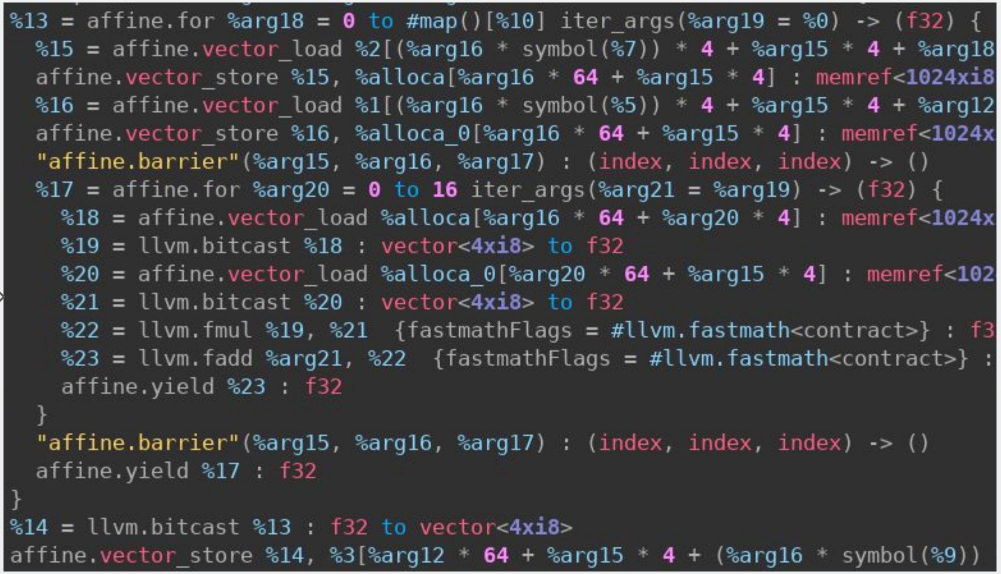
Preserving loop information through LLVM transformations



The raising pipeline: finding the polyhedral structure



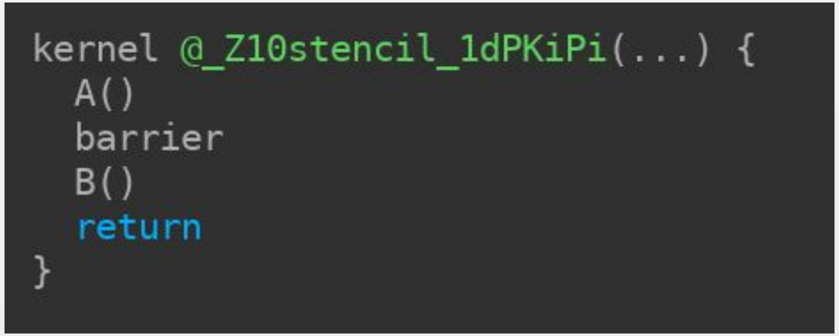
```
%20 = llvm.getelementptr inbounds %5[0, %18, %19] : (!llvm.ptr, i64, i64) -> !llvm.
%21 = arith.muli %10, %arg11 : i32
%22 = arith.addi %21, %9 : i32
%23 = llvm.getelementptr inbounds %6[0, %18, %19] : (!llvm.ptr, i64, i64) -> !llvm.
%24:2 = scf.for %arg18 = %12 to %13 step %c16_i32 iter_args(%arg19 = %14, %arg20 =
  %31 = arith.addi %17, %arg18 : i32
  %32 = arith.extsi %31 : i32 to i64
  %33 = llvm.getelementptr inbounds %arg8[%32] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %34 = llvm.load %33 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : !llvm.ptr -> f32
  llvm.store %34, %20 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : f32, !llvm.ptr
  %35 = arith.addi %22, %arg19 : i32
  %36 = arith.extsi %35 : i32 to i64
  %37 = llvm.getelementptr inbounds %arg9[%36] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %38 = llvm.load %37 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : !llvm.ptr -> f32
  llvm.store %38, %23 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : f32, !llvm.ptr
  "affine.barrier"(%arg15, %arg16, %arg17) : (index, index, index) -> ()
  %39 = scf.for %arg21 = %c0_i32 to %c16_i32 step %c1_i32 iter_args(%arg22 = %arg20
    %41 = arith.extui %arg21 : i32 to i64
    %42 = llvm.getelementptr inbounds %5[0, %18, %41] : (!llvm.ptr, i64, i64) -> !l
    %43 = llvm.load %42 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : !llvm.ptr -> f
    %44 = llvm.getelementptr inbounds %6[0, %41, %19] : (!llvm.ptr, i64, i64) -> !l
    %45 = llvm.load %44 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : !llvm.ptr -> f
    %46 = llvm.fmul %43, %45  {fastmathFlags = #llvm.fastmath<contract>} : f32
    %47 = llvm.fadd %arg22, %46  {fastmathFlags = #llvm.fastmath<contract>} : f32
    scf.yield %47 : f32
  }
  "affine.barrier"(%arg15, %arg16, %arg17) : (index, index, index) -> ()
  %40 = arith.addi %arg19, %15 : i32
  scf.yield %40, %39 : i32, f32
}
%25 = arith.muli %15, %8 : i32
%26 = arith.addi %14, %9 : i32
%27 = arith.addi %26, %21 : i32
%28 = arith.addi %27, %25 : i32
%29 = arith.extsi %28 : i32 to i64
%30 = llvm.getelementptr inbounds %arg7[%29] : (!llvm.ptr, i64) -> !llvm.ptr, f32
llvm.store %24#1, %30 {alignment = 4 : i64, tbaa = [#tbaa_tag1]} : f32, !llvm.ptr
```



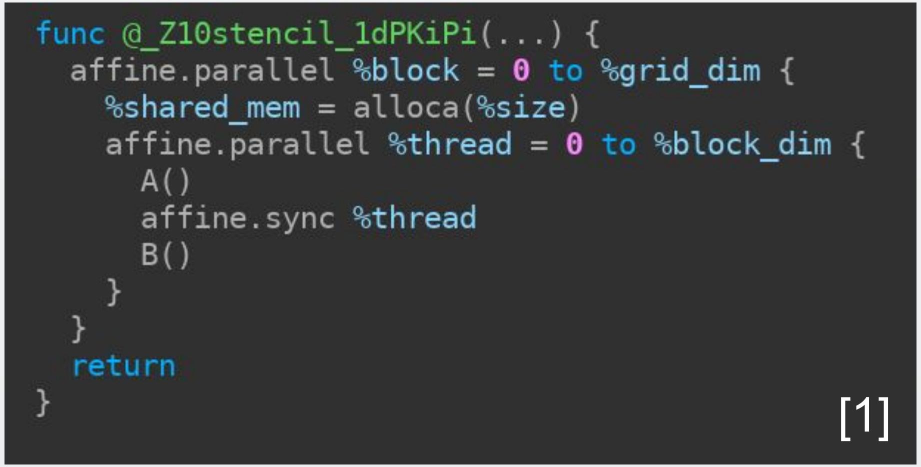
```
%13 = affine.for %arg18 = 0 to #map()[%10] iter_args(%arg19 = %0) -> (f32) {
  %15 = affine.vector_load %2[(%arg16 * symbol(%7)) * 4 + %arg15 * 4 + %arg18
  affine.vector_store %15, %alloca[%arg16 * 64 + %arg15 * 4] : memref<1024xi8
  %16 = affine.vector_load %1[(%arg16 * symbol(%5)) * 4 + %arg15 * 4 + %arg12
  affine.vector_store %16, %alloca_0[%arg16 * 64 + %arg15 * 4] : memref<1024x
  "affine.barrier"(%arg15, %arg16, %arg17) : (index, index, index) -> ()
  %17 = affine.for %arg20 = 0 to 16 iter_args(%arg21 = %arg19) -> (f32) {
    %18 = affine.vector_load %alloca[%arg16 * 64 + %arg20 * 4] : memref<1024x
    %19 = llvm.bitcast %18 : vector<4xi8> to f32
    %20 = affine.vector_load %alloca_0[%arg20 * 64 + %arg15 * 4] : memref<102
    %21 = llvm.bitcast %20 : vector<4xi8> to f32
    %22 = llvm.fmul %19, %21  {fastmathFlags = #llvm.fastmath<contract>} : f3
    %23 = llvm.fadd %arg21, %22  {fastmathFlags = #llvm.fastmath<contract>} :
    affine.yield %23 : f32
  }
  "affine.barrier"(%arg15, %arg16, %arg17) : (index, index, index) -> ()
  affine.yield %17 : f32
}
%14 = llvm.bitcast %13 : f32 to vector<4xi8>
affine.vector_store %14, %3[%arg12 * 64 + %arg15 * 4 + (%arg16 * symbol(%9))
```

affine loop bounds and access indices

Representing the parallel structure of a GPU kernel



```
kernel @_Z10stencil_1dPKiPi(...) {
  A()
  barrier
  B()
  return
}
```



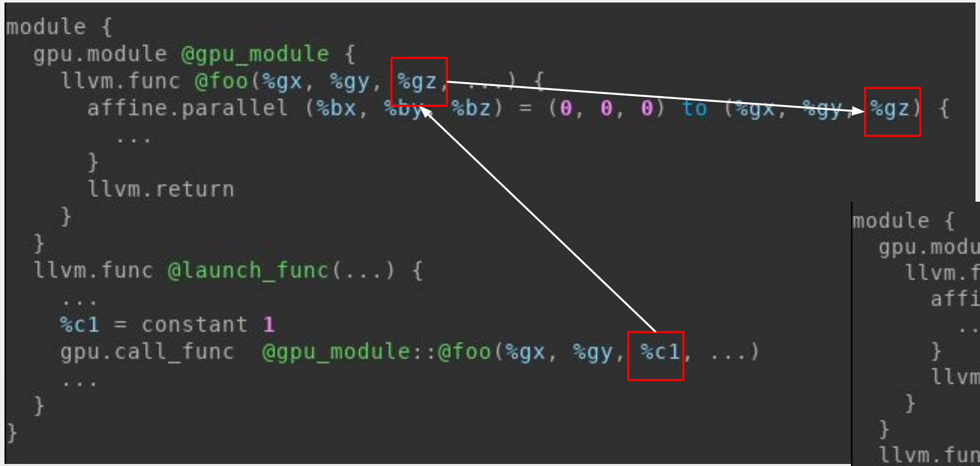
```
func @_Z10stencil_1dPKiPi(...) {
  affine.parallel %block = 0 to %grid_dim {
    %shared_mem = alloca(%size)
    affine.parallel %thread = 0 to %block_dim {
      A()
      affine.sync %thread
      B()
    }
  }
  return
}
```

[1]

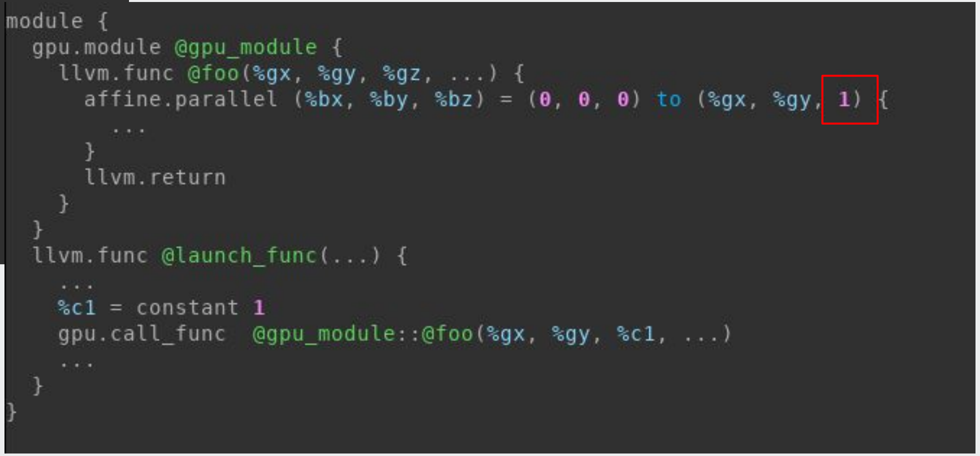
[1] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs
Authors: William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, Oleksandr ZinenkoAuthors Info & Claims

Propagating constants

Extremely important for subsequent analysis

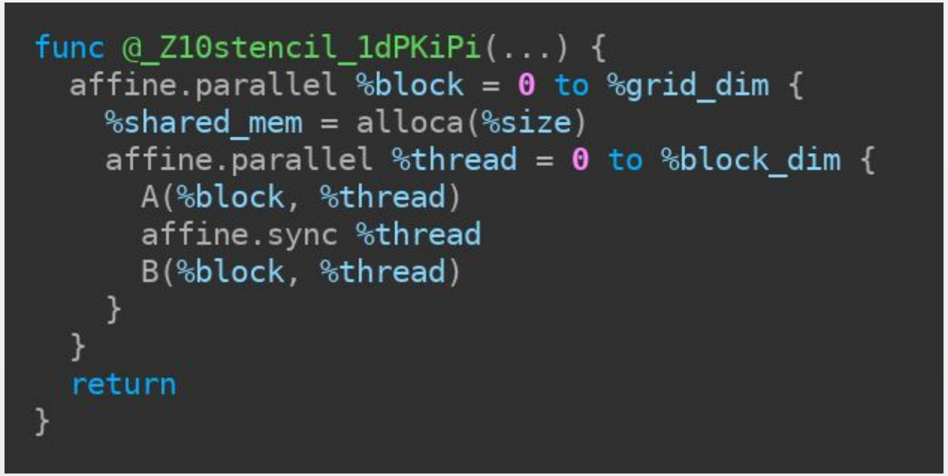


```
module {
  gpu.module @gpu_module {
    llvm.func @foo(%gx, %gy, %gz, ...) {
      affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (%gx, %gy, %gz) {
        ...
      }
      llvm.return
    }
  }
  llvm.func @launch_func(...) {
    ...
    %c1 = constant 1
    gpu.call_func  @gpu_module::@foo(%gx, %gy, %c1, ...)
    ...
  }
}
```



```
module {
  gpu.module @gpu_module {
    llvm.func @foo(%gx, %gy, %gz, ...) {
      affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (%gx, %gy, 1) {
        ...
      }
      llvm.return
    }
  }
  llvm.func @launch_func(...) {
    ...
    %c1 = constant 1
    gpu.call_func  @gpu_module::@foo(%gx, %gy, %c1, ...)
    ...
  }
}
```

Representing synchronization



```
func @_Z10stencil_1dPKiPi(...) {
  affine.parallel %block = 0 to %grid_dim {
    %shared_mem = alloca(%size)
    affine.parallel %thread = 0 to %block_dim {
      A(%block, %thread)
      affine.sync %thread
      B(%block, %thread)
    }
  }
  return
}
```

What *is* `affine.sync` ?

Transforming to pure polyhedral

No synchronisation allowed

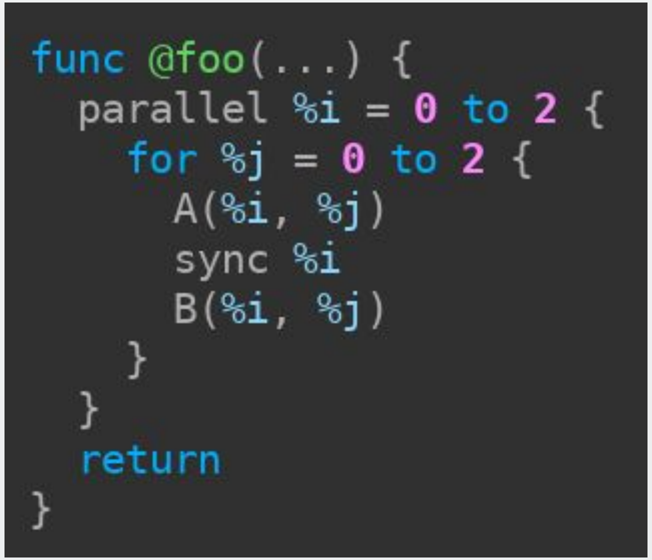
Parallel loop fission

```
parallel %i = 0 to %n
  A(%i)
  sync %i
  B(%i)
```

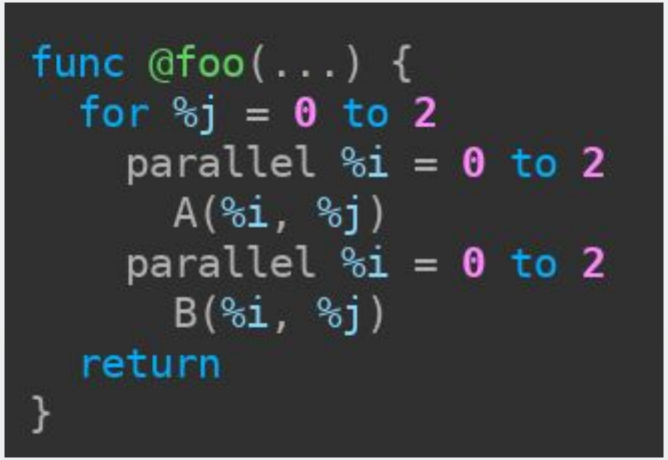
```
parallel %i = 0 to %n
  A(%i)
parallel %i = 0 to %n
  B(%i)
```

[1] Moses, W.S., Ivanov, I.R., Domke, J., Endo, T., Doerfert, J. and Zinenko, O., 2023, February. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (pp. 119-134).
[2] Stratton, J.A., Stone, S.S. and Hwu, W.M.W., 2008, July. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In International Workshop on Languages and Compilers for Parallel Computing (pp. 16-30). Berlin, Heidelberg: Springer Berlin Heidelberg.

Nested barriers?



```
func @foo(...) {
  parallel %i = 0 to 2 {
    for %j = 0 to 2 {
      A(%i, %j)
      sync %i
      B(%i, %j)
    }
  }
  return
}
```



```
func @foo(...) {
  for %j = 0 to 2
    parallel %i = 0 to 2
      A(%i, %j)
    parallel %i = 0 to 2
      B(%i, %j)
  return
}
```

[1] Moses, W.S., Ivanov, I.R., Domke, J., Endo, T., Doerfert, J. and Zinenko, O., 2023, February. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (pp. 119-134).
[2] Stratton, J.A., Stone, S.S. and Hwu, W.M.W., 2008, July. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In International Workshop on Languages and Compilers for Parallel Computing (pp. 16-30). Berlin, Heidelberg: Springer Berlin Heidelberg.

Let’s look at matmul

```
#define BLOCK_SIZE 16

__global__ void MatrixMulCUDA(float *C, float *A,
                              float *B, int wA,
                              int wB) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep  = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep  = BLOCK_SIZE * wB;
  float Csub = 0;
  for (int a = aBegin, b = bBegin;
       a <= aEnd;
       a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();
    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}

int MatrixMultiply(const dim3 &dimsA,
                   const dim3 &dimsB,
                   float *A, float *B, float *C) {
  // Setup execution parameters
  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
  MatrixMulCUDA<<<grid, threads>>>(C, A, B, dimsA.x, dimsB.x);
}
```

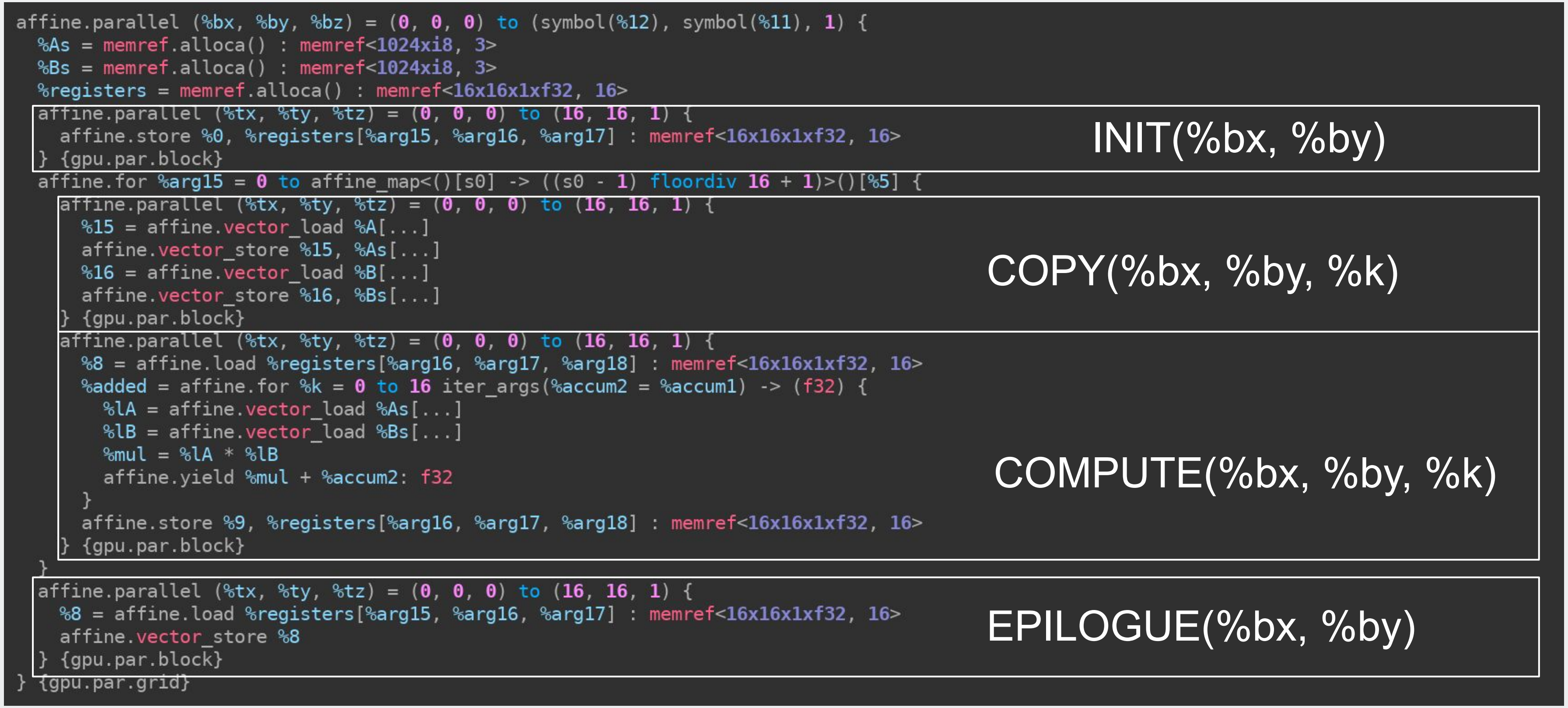
```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %res = affine.for %i = 0 to #map()[%10] iter_args(%accum1 = %0) -> (f32) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
      "affine.barrier"(%tx, %ty, %tz)
      %added = affine.for %arg20 = 0 to 16 iter_args(%accum2 = %accum1) -> (f32) {
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      "affine.barrier"(%tx, %ty, %tz)
      affine.yield %added : f32
    }
    affine.vector_store %res, %C[...]
  } {gpu.par.block}
} {gpu.par.grid}
```

```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %res = affine.for %i = 0 to #map()[%10] iter_args(%accum1 = %0) -> (f32) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
      "affine.barrier"(%tx, %ty, %tz)
      %added = affine.for %arg20 = 0 to 16 iter_args(
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      "affine.barrier"(%tx, %ty, %tz)
      affine.yield %added : f32
    }
    affine.vector_store %res, %C[...]
  } {gpu.par.block}
} {gpu.par.grid}
```

```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  %registers = memref.alloca() : memref<16x16x1xf32, 16>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    affine.store %0, %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
  } {gpu.par.block}
  affine.for %arg15 = 0 to affine_map<()[s0] -> ((s0 - 1) floordiv 16 + 1)>()[%5] {
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
    } {gpu.par.block}
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %8 = affine.load %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
      %added = affine.for %k = 0 to 16 iter_args(%accum2 = %accum1) -> (f32) {
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      affine.store %9, %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
    } {gpu.par.block}
  }
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %8 = affine.load %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
    affine.vector_store %8
  } {gpu.par.block}
} {gpu.par.grid}
```

Analyzable using standard polyhedral techniques

Example analysis of matmul



```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  %registers = memref.alloca() : memref<16x16x1xf32, 16>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    affine.store %0, %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
  } {gpu.par.block}
  affine.for %arg15 = 0 to affine_map<()[s0] -> ((s0 - 1) floordiv 16 + 1)>()[%5] {
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
    } {gpu.par.block}
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %8 = affine.load %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
      %added = affine.for %k = 0 to 16 iter_args(%accum2 = %accum1) -> (f32) {
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      affine.store %9, %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
    } {gpu.par.block}
  }
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %8 = affine.load %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
    affine.vector_store %8
  } {gpu.par.block}
} {gpu.par.grid}
```

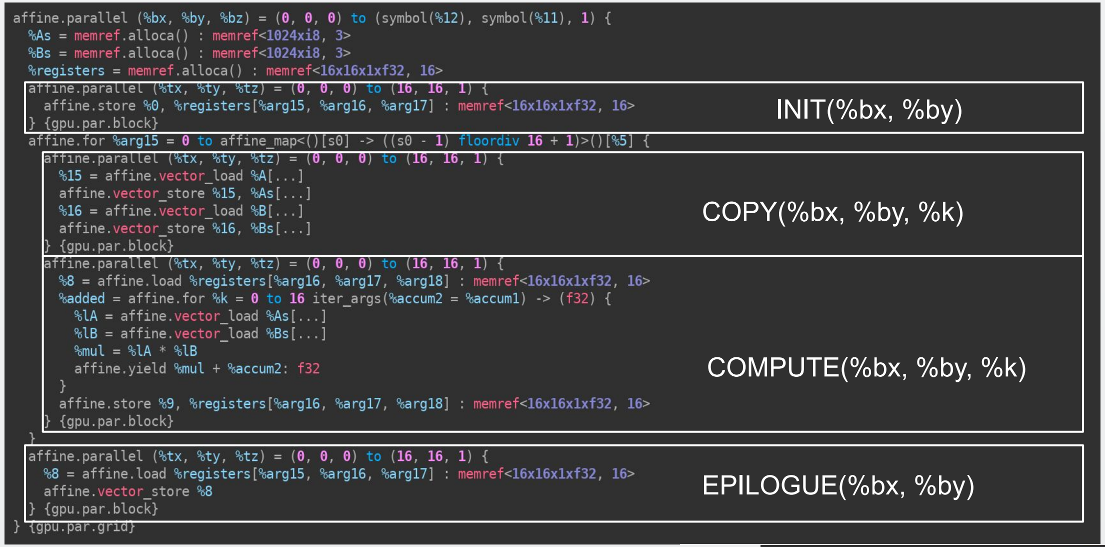
INIT(%bx, %by)

COPY(%bx, %by, %k)

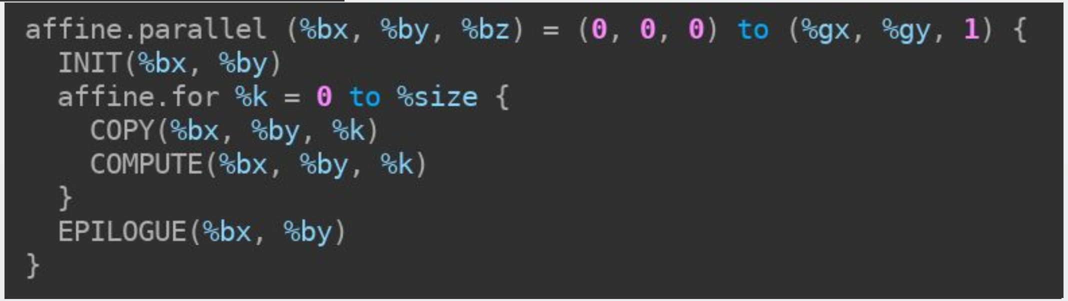
COMPUTE(%bx, %by, %k)

EPILOGUE(%bx, %by)

Example analysis of matmul



```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  %registers = memref.alloca() : memref<16x16x1xf32, 16>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    affine.store %0, %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
  } {gpu.par.block}
  affine.for %arg15 = 0 to affine_map<()[s0] -> ((s0 - 1) floordiv 16 + 1)>()[%5] {
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
    } {gpu.par.block}
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %8 = affine.load %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
      %added = affine.for %k = 0 to 16 iter_args(%accum2 = %accum1) -> (f32) {
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      affine.store %9, %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
    } {gpu.par.block}
  }
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %8 = affine.load %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
    affine.vector_store %8
  } {gpu.par.block}
} {gpu.par.grid}
```



```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx, %by)
  affine.for %k = 0 to %size {
    COPY(%bx, %by, %k)
    COMPUTE(%bx, %by, %k)
  }
  EPILOGUE(%bx, %by)
}
```

Polyhedral scheduling

Polyhedral model

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < i; j++) {
    S(i, j);
    R(i, j);
  }
}
```

```
domain

{ S(i, j) : 0 <= i < N,
            0 <= j < i }
{ R(i, j) : 0 <= i < N,
            0 <= j < i }
```

```
dependencies

{ S(si, sj) -> R(ri, rj) :
    si = ri, sj = rj }
```

Everything is represented using linear algebra

Polyhedral scheduling

Integer Linear Programming Problem

maximize parallelism
minimize temporal distance between dependencies

subject to validity constraints (dependencies)

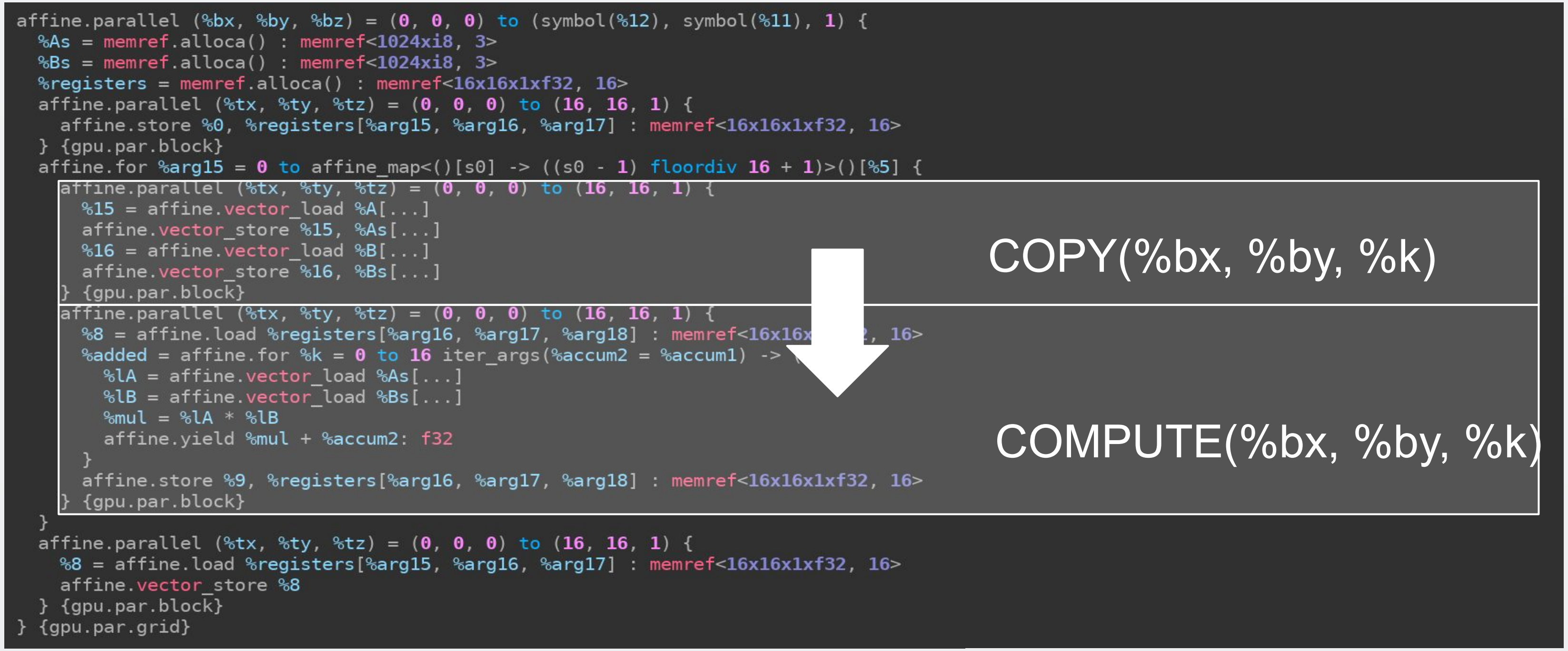
How do we optimize for async execution?

Optimizing for async: Copy detection

```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  %registers = memref.alloca() : memref<16x16x1xf32, 16>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    affine.store %0, %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
  } {gpu.par.block}
  affine.for %arg15 = 0 to affine_map<()[s0] -> ((s0 - 1) floordiv 16 + 1)>()[%5] {
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
    } {gpu.par.block}
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %8 = affine.load %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
      %added = affine.for %k = 0 to 16 iter_args(%accum2 = %accum1) -> (f32) {
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      affine.store %9, %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
    } {gpu.par.block}
  }
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %8 = affine.load %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
    affine.vector_store %8
  } {gpu.par.block}
} {gpu.par.grid}
```

COPY(%bx, %by, %k)

Optimizing for async:
Detecting dependencies on copies



```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (symbol(%12), symbol(%11), 1) {
  %As = memref.alloca() : memref<1024xi8, 3>
  %Bs = memref.alloca() : memref<1024xi8, 3>
  %registers = memref.alloca() : memref<16x16x1xf32, 16>
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    affine.store %0, %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
  } {gpu.par.block}
  affine.for %arg15 = 0 to affine_map<()[s0] -> ((s0 - 1) floordiv 16 + 1)>()[%5] {
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %15 = affine.vector_load %A[...]
      affine.vector_store %15, %As[...]
      %16 = affine.vector_load %B[...]
      affine.vector_store %16, %Bs[...]
    } {gpu.par.block}
    affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
      %8 = affine.load %registers[%arg16, %arg17, %arg18] : memref<16x16x    2, 16>
      %added = affine.for %k = 0 to 16 iter_args(%accum2 = %accum1) -> (
        %lA = affine.vector_load %As[...]
        %lB = affine.vector_load %Bs[...]
        %mul = %lA * %lB
        affine.yield %mul + %accum2: f32
      }
      affine.store %9, %registers[%arg16, %arg17, %arg18] : memref<16x16x1xf32, 16>
    } {gpu.par.block}
  }
  affine.parallel (%tx, %ty, %tz) = (0, 0, 0) to (16, 16, 1) {
    %8 = affine.load %registers[%arg15, %arg16, %arg17] : memref<16x16x1xf32, 16>
    affine.vector_store %8
  } {gpu.par.block}
} {gpu.par.grid}
```

Async deps:

```
{ COPY(i, j, k) -> COMPUTE(i, j, k) }
```

Optimizing for async: Optimization objective

```
affine.parallel (%bx, %by, %bz) =
    (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx), %by)
  affine.for %k = 0 to %size {
    COPY(%bx, %by, %k)
    COMPUTE(%bx, %by, %k)
  }
  EPILOGUE(%bx, %by)
}
```

Async deps:

```
{ COPY(i, j, k) -> COMPUTE(i, j, k) }
```

Let’s maximize the temporal distance of async deps

Let’s maximize the temporal distance of async deps

```
affine.parallel (%bx, %by, %bz) =
    (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx), %by)
  affine.for %k = 0 to %size {
    COPY(%bx, %by, %k)
    COMPUTE(%bx, %by, %k)
  }
  EPILOGUE(%bx, %by)
}
```

```
affine.parallel (%bx, %by, %bz) =
    (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx), %by)
  affine.for %k = 0 to %size {
    COPY(%bx, %by, %k)
  }
  affine.for %k = 0 to %size {
    COMPUTE(%bx, %by, %k)
  }
  EPILOGUE(%bx, %by)
}
```

We need an infinite amount of memory? Not very useful...

We need a way to constraint that

Live range overlap constraint

Upper bound on the amount of memory used.

Dependencies

```
affine.parallel (%bx, %by, %bz) = (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx, %by)
  affine.for %k = 0 to %size {
    COPY(%bx, %by, %k)
    COMPUTE(%bx, %by, %k)
  }
  EPILOGUE(%bx, %by)
}
```

True dependencies (live ranges)

```
COPY(k) -> COMPUTE(k)
```

“False” dependencies

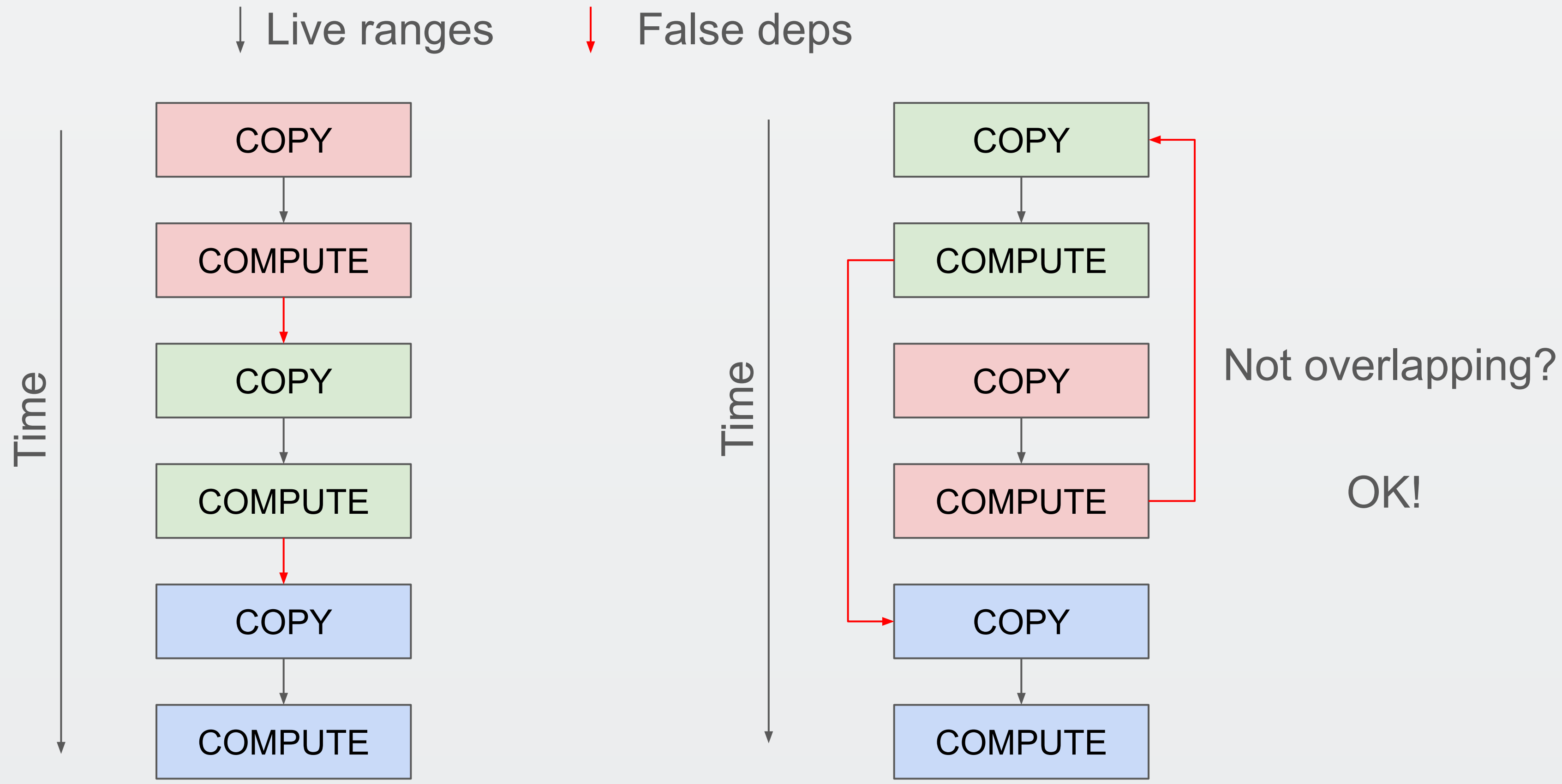
```
COMPUTE(k) -> COPY(k+1)
```

Prior work Live-range reordering: allow conditionally breaking the “false” dependencies.

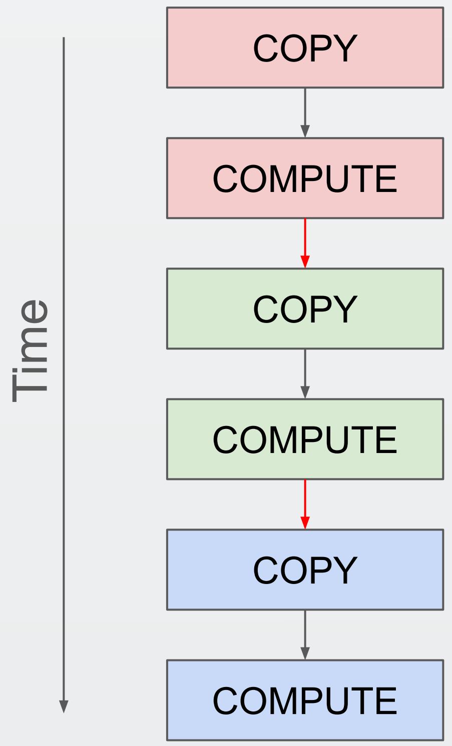
Verdoolaege, Sven, and Albert Cohen. "Live-range reordering." International Workshop on Polyhedral Compilation Techniques, Date: 2016/01/19-2016/01/19, Location: Prague, Czech Republic. 2016.

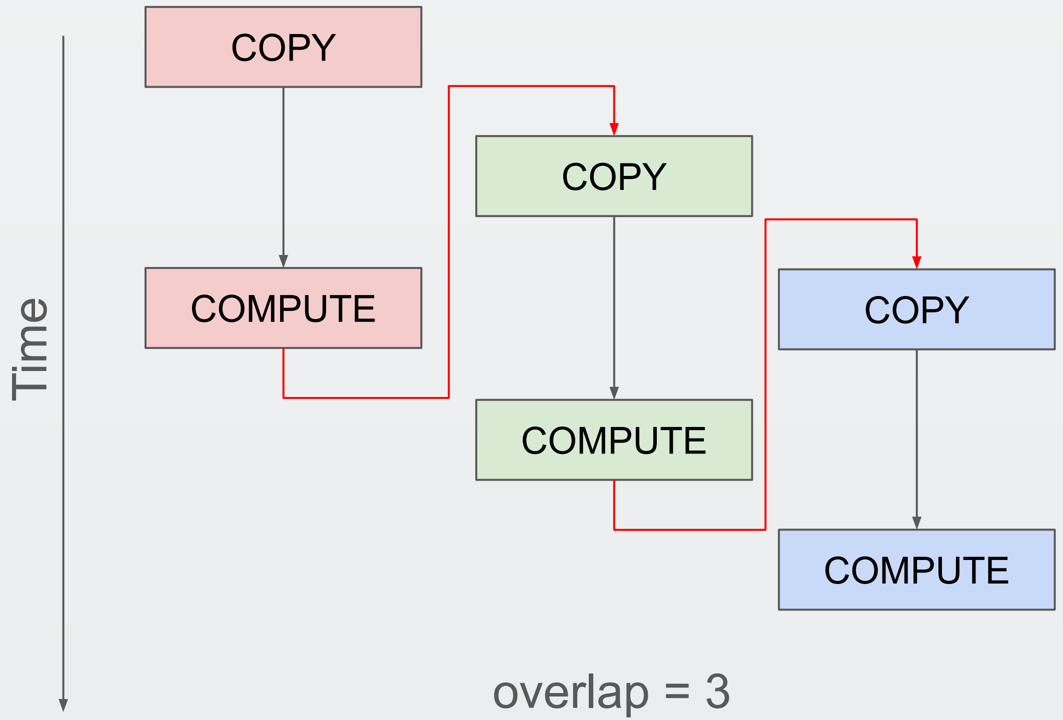
Prior work: Live-range reordering

Verdoolaege, Sven, and Albert Cohen. "Live-range reordering." International Workshop on Polyhedral Compilation Techniques, Date: 2016/01/19-2016/01/19, Location: Prague, Czech Republic. 2016.



Live-range overlapping





overlap = 3

Constraint in ILP

$$\sum_{a \in arrays} size(a) \times overlap(a) \leq AvailableSharedMemory$$

Polyhedral scheduling

ILP problem (prior work)

maximize parallelism

minimize temporal distance between dependencies

subject to validity constraints (dependencies)

ILP problem (our work)

maximize parallelism

maximize async dep distance

<u>**minimize** temporal distance between dependencies</u>

subject to validity constraints (dependencies)

<u>**subject to** overlap constraints</u>

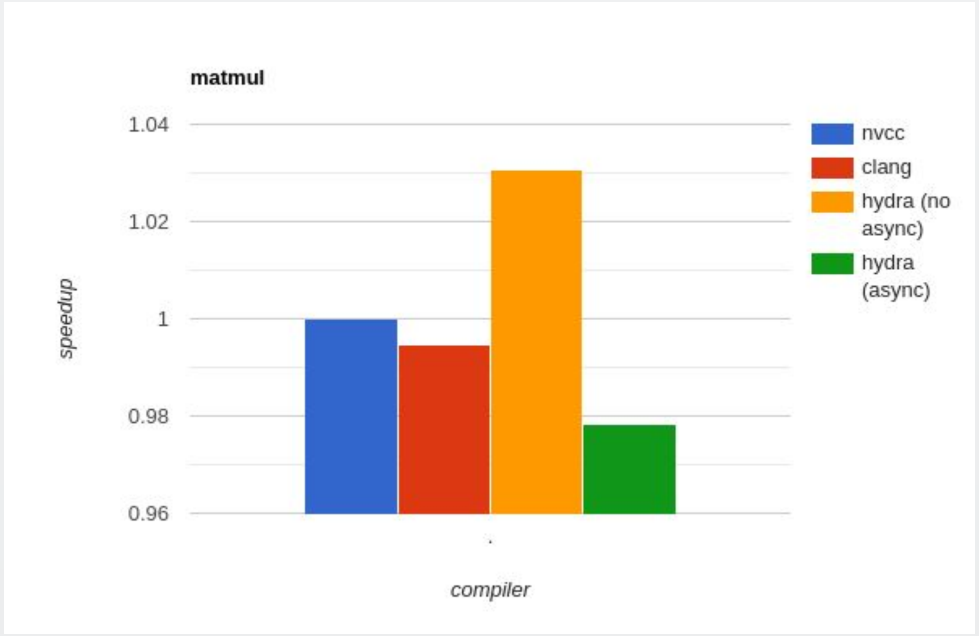
Final version

```
affine.parallel (%bx, %by, %bz) =
    (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx), %by)
  affine.for %k = 0 to %size {
    COPY(%bx, %by, %k)
    COMPUTE(%bx, %by, %k)
  }
  EPILOGUE(%bx, %by)
}
```

```
affine.parallel (%bx, %by, %bz) =
    (0, 0, 0) to (%gx, %gy, 1) {
  INIT(%bx), %by)
  COPY(%bx, %by, 0)
  COPY(%bx, %by, 1)
  affine.for %k = 0 to %size - 2 {
    COPY(%bx, %by, %k + 2)
    COMPUTE(%bx, %by, %k)
  }
  COMPUTE(%bx, %by, %size - 2)
  COMPUTE(%bx, %by, %size - 1)
  EPILOGUE(%bx, %by)
}
```

Some early stage evaluation

The matrix multiplication example



Conclusion

- LLVM -> MLIR Affine raising pipeline
- Polyhedral analysis of code with synchronisation
- Polyhedral scheduling with async

Questions?