Polyhedral Modeling for Heterogeneous Compute

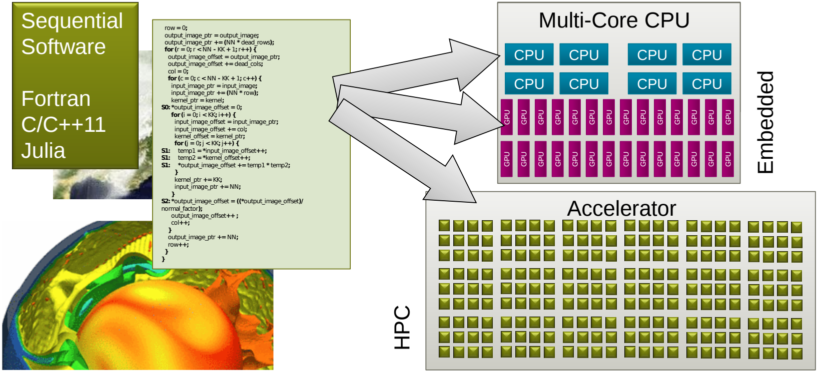
Tobias Grosser

ETH Zurich / PollyLabs

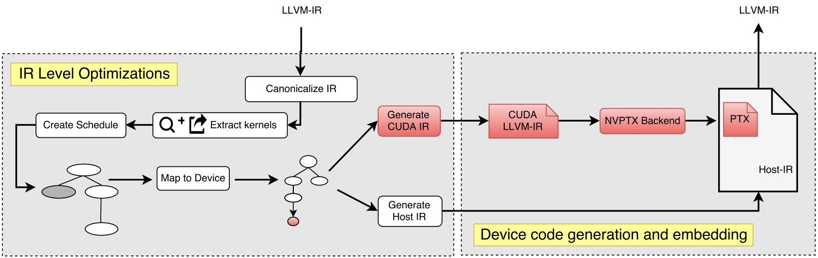
EuroLLVM 2016

19. March 2016, Barcelona, Spain

Objective


Sequential Software
Fortran
C/C++11
Julia
row = 0;
output_image_ptr = output_image;
output_image_ptr += (NN * dead_rows);
for (r = 0; r < NN - KK + 1; r++) {
output_image_offset = output_image_ptr;
output_image_offset += dead_cols;
col = 0;
for (c = 0; c < NN - KK + 1; c++) {
input_image_ptr = input_image;
input_image_ptr += (NN * row);
kernel_ptr = kernel;
S0: *output_image_offset = 0;
for (i = 0; i < KK; i++) {
input_image_offset = input_image_ptr;
input_image_offset += col;
kernel_offset = kernel_ptr;
for (j = 0; j < KK; j++) {
S1: temp1 = *input_image_offset++;
S1: temp2 = *kernel_offset++;
S1: *output_image_offset += temp1 * temp2;
}
kernel_ptr += KK;
input_image_ptr += NN;
}
S2: *output_image_offset = ((*output_image_offset)/
normal_factor);
output_image_offset++ ;
col++;
}
output_image_ptr += NN;
row++;
}
}
Multi-Core CPU
CPU CPU CPU CPU
CPU CPU CPU CPU
GPU
Embedded
Accelerator
HPC

Architecture

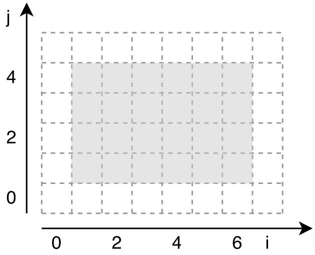


Mapping computations to thread-blocks and threads

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
               + A[i  ][j+1] + A[i  ][j-1];
```

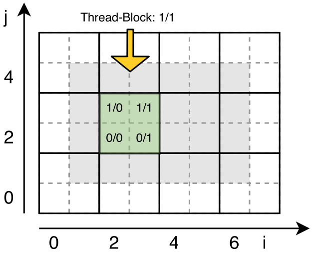
Mapping computations to thread-blocks and threads

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
             + A[i  ][j+1] + A[i  ][j-1];
```



Mapping computations to thread-blocks and threads

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                + A[i  ][j+1] + A[i  ][j-1];
```

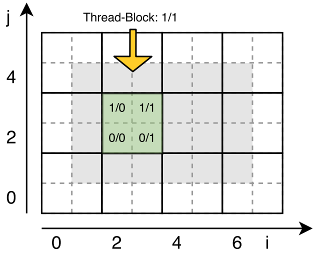


Mappings:

$$\{S[i,j] \rightarrow blocks[floor(i/2), floor(j,2)]\}$$
$$\{S[i,j] \rightarrow threads[i \bmod 2, j \bmod 2]\}$$

Mapping computations to thread-blocks and threads

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
               + A[i  ][j+1] + A[i  ][j-1];
```



Mappings:

$\{S[i,j] \rightarrow blocks[floor(i/2), floor(j,2)]\}$
$\{S[i,j] \rightarrow threads[i \bmod 2, j \bmod 2]\}$

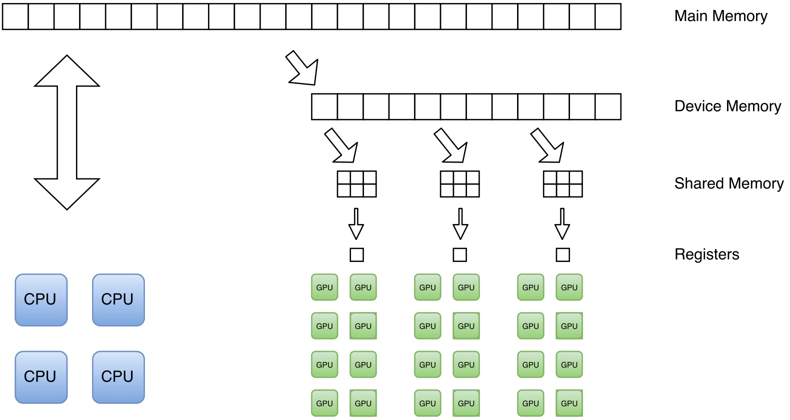
In case we create more thread-blocks than supported in hardware, thread-blocks are assigned round-robin!

Generated accelerator code

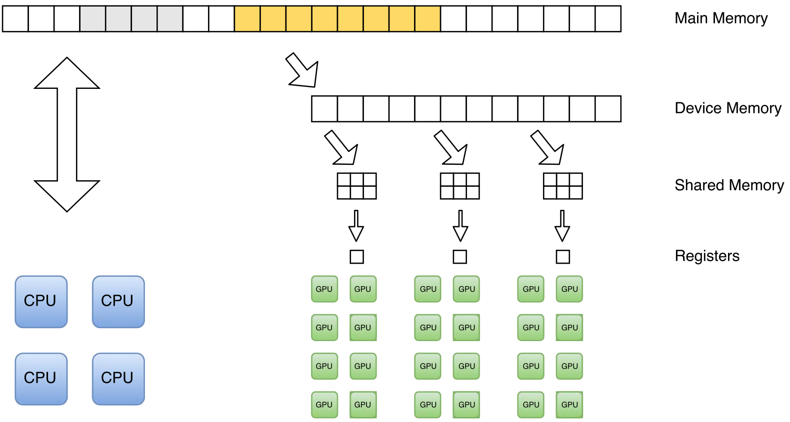
```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;

  int i = 2 * b0 + t0;
  int j = 2 * b1 + t1;
  S:  B[i][j] +=   A[i+1][j  ] + A[i-1][j  ]
                 + A[i  ][j+1] + A[i  ][j-1];
}
```

Commonly not a single computation per-kernel, but also loops/synchronizations.

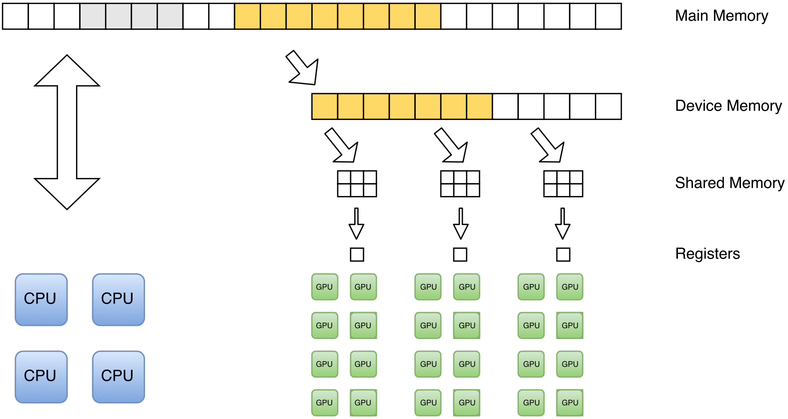
Memory hierarchy of an accelerator system



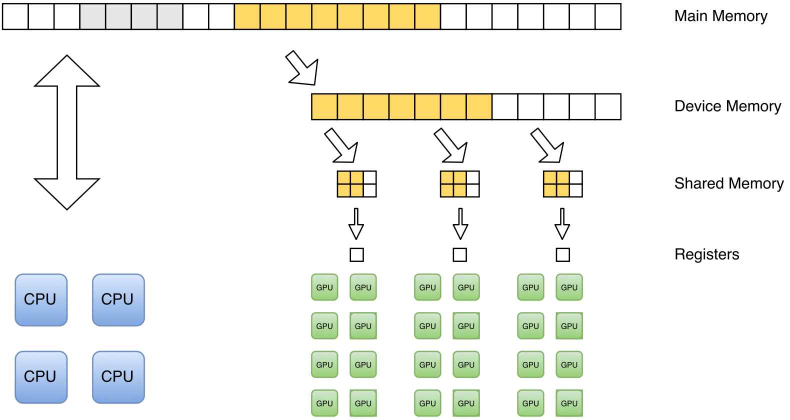
Memory hierarchy of an accelerator system



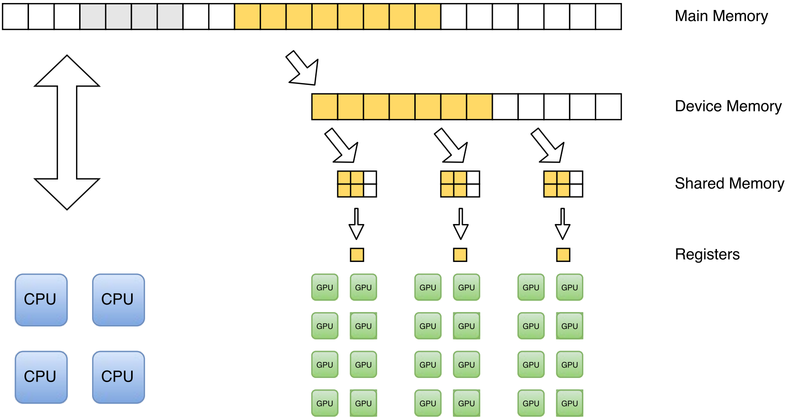
Memory hierarchy of an accelerator system



Memory hierarchy of an accelerator system

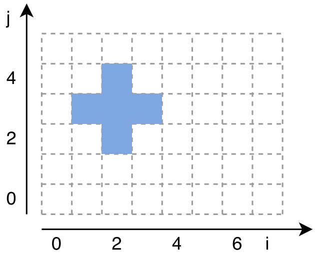


Memory hierarchy of an accelerator system



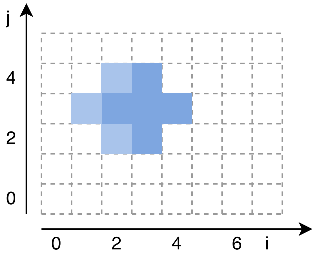
Identify array subregions accessed by threadblock

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] +=    A[i+1][j  ] + A[i-1][j  ]
            + A[i  ][j+1] + A[i  ][j-1];
```



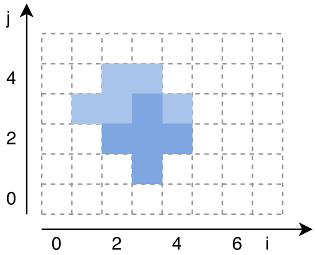
Identify array subregions accessed by threadblock

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] +=    A[i+1][j  ] + A[i-1][j  ]
              + A[i  ][j+1] + A[i  ][j-1];
```



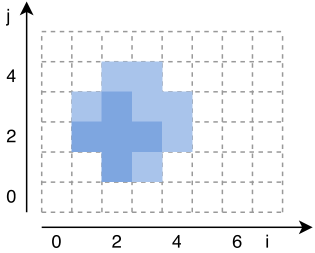
Identify array subregions accessed by threadblock

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] +=    A[i+1][j  ] + A[i-1][j  ]
                + A[i  ][j+1] + A[i  ][j-1];
```



Identify array subregions accessed by threadblock

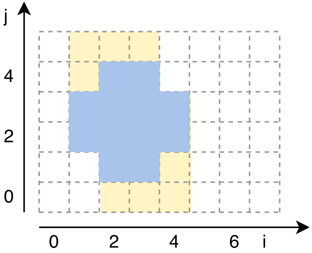
```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] +=   A[i+1][j  ] + A[i-1][j  ]
               + A[i  ][j+1] + A[i  ][j-1];
```



Maximal storage efficiency possible with counting (barvinok).
BUT, accesses become inefficient.

Identify array subregions accessed by threadblock

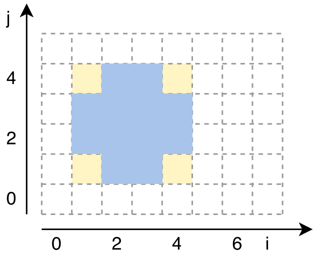
```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] +=   A[i+1][j  ] + A[i-1][j  ]
               + A[i  ][j+1] + A[i  ][j-1];
```



Copying a one-dimensional set of memory addresses (including untouched addresses in between).

Identify array subregions accessed by threadblock

```
for (i = 1; i <= 6; i++)
  for (j = 1, j <= 4; i++)
S:  B[i][j] +=    A[i+1][j  ] + A[i-1][j  ]
                + A[i  ][j+1] + A[i  ][j-1];
```



Copying a multi-dimensional set of array locations (including untouched addresses in between).

$\Rightarrow$ More efficient!

Map array subregions to shared memory

- For each array subregion identified, check if:
  - data-elements are used multiple times
    or
  - accesses to global memory are not coalesced
  - and the dataset size fits into shared memory

$\Rightarrow$ allocate shared memory for subregion

Generated code when using shared memory

Each thread-block executes:

- Copy global ⇒ shared (new)
- synchronize()
- Compute in shared memory (changed)
- synchronize()
- Copy shared ⇒ global (new)

Optimizing the copy code

Global ⇒ Shared

- Data element is read in thread-block
- … but has not been computed earlier in the same thread block
- Over approximate data to load with the rectangle to simplify code

Shared ⇒ Global

- Data element is written in thread-block
- … and is used later outside of the thread block but not overwritten in between.
- Do not over-approximate storage set.

Local memory / registers

- Algorithm mirrors shared memory mapping
- Use local memory in case data remains thread-local
- Unroll computation to ensure constant access expressions:

```
for (i = t0; i < 128; i+=32)
    A[floor(i / 32)] = i;

          ⇓

    A[0] = t0;
    A[1] = t0 + 32;
    A[2] = t0 + 64;
    A[3] = t0 + 96;
```

Lowering of arrays of parametric size in LLVM

```
void gemm(int n, int m, int p,
          float A[n][p], float B[p][m], float C[n][m]) {
L1:  for (int i = 0; i < n; i++)
L2:    for (int j = 0; j < m; j++)
L3:      for (int k = 0; k < p; ++k)
           C[i][j] += A[i][k] * B[k][j];
}
```

C99 arrays lowered to LLVM-IR

```
define void @gemm(i32 %n, i32 %m, i32 %p, float* %A, float* %B, float* %C) {
; for i:
;   for j:
;     for k:
        %A.idx = mul i32 %i, %p
        %A.idx2 = add i32 %A.idx, %k
        %A.idx3 = getelementptr float* %A, i32 %A.idx2
        %A.data = load float* %A.idx3
        %B.idx = mul i32 %k, %m
        %B.idx2 = add i32 %B.idx, %j
        %B.idx3 = getelementptr float* %B, i32 %B.idx2
        %B.data = load float* %B.idx3
        %C.idx = mul i32 %i, %m
        %C.idx2 = add i32 %C.idx, %j.0
        %C.idx3 = getelementptr float* %C, i32 %C.idx2
        %C.data = load float* %C.idx3
        %mul = fmul float %A.data, %B.data
        %add = fadd float %C.data, %mul
        store float %add, float* %C.idx3
}
```

Recovery of Index Expressions using SCEV

Recovered accesses are:

- Single dimensional
- *Polynomial*

```
void gemm(int n, int m, int p,
          float A[], float B[], float C[]) {
L1:  for (int i = 0; i < n; i++)
L2:    for (int j = 0; j < m; j++)
L3:      for (int k = 0; k < p; ++k)
           C[i * m + j] += A[i * p + k] * B[k * M + j];
}
```

The Problem

Given *a set of* **single dimensional memory accesses** *with index expressions that are multivariate polynomials and a set of iteration domains,* **derive a multi-dimensional view***:*

- A multi-dimensional array definition
- For each original array access:
  a new multi-dimensional access function

Grosser Tobias, Pop Sebastian, Pouchet Louis-Noel, Sadayappan P, Ramanujam J. **Optimistic Delinearization of Parametrically Sized Arrays**, International Conference on Supercomputing (ICS), 2015

Conditions

- **R1** - **Affine**
  New access functions are affine
- **R2** - **Equivalence**
  Addresses in original and multi-dimensional view are identical
- **R3** - **In-Bounds**
  Array subscripts are within bounds (except outer dimension)

If **R3** not statically provable $\rightarrow$ derive run-time conditions.

Example: Initialize subarray (I)

- Array size: $n_0 \times n_1 \times n_2$
- Subarray position: $o_0 \times o_1 \times o_2$
- Subarray size: $s_0 \times s_1 \times s_2$

```
void set_subarray(float A[],
                  size_t o0, size_t o1, size_t o2,
                  size_t s0, size_t s1, size_t s2,
                  size_t n0, size_t n1, size_t n2) {
  for (size_t i = 0; i < s0; i++)
    for (size_t j = 0; j < s1; j++)
      for (size_t k = 0; k < s2; k++)
S:        A[(n2 * (n1 * o0 + o1) + o2)
            + n1 * n2 * i + n2 * j + k] = 1;
      // A[o0 + i, o1 + j, o1 + k] = 1
}
```

Example: Initialize subarray (II)

1. **Start**
   $(n_2(n_1 o_0 + o_1) + o_2) + n_1 n_2 i + n_2 j + k$
2. **Expand expression**
   $n_2 n_1 o_0 + n_2 o_1 + o_2 + n_1 n_2 i + n_2 j + k$
3. **Extract Terms containing induction variables**
   $\{n_1 n_2 i, n_2 j, k\}$
4. **Drop non-parameters and sort terms by #elements**
   $\{n_1 n_2, n_2\}$
5. **Assumed size**
   `A[][n1][n2]`

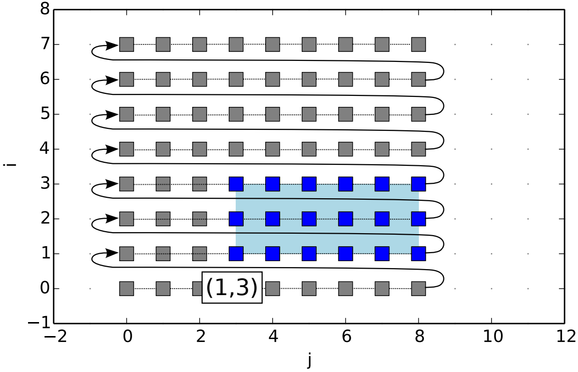
Example: Initialize subarray (III)

6. **Inner dimension**: divide by n_2
   Quotient: $n_1 o_0 + o_1 + n_1 i + n_2 j$
   Remainder: $o_2 + k$ $\rightarrow A[?][?][k + o_2]$
7. **Second inner dimension**: divide by n_1
   Quotient: $o_0 + i$ $\rightarrow A[i + o_0][?][?]$
   Remainder: $o_1 + j$ $\rightarrow A[?][j + o_1][?]$
8. **Full array access**: $A[i + o_0][j + o_1][k + o_2]$
9. **Validity conditions**:

$$\forall i, j, k : \quad 0 \leq i < s_0 \wedge 0 \leq j < s_1 \wedge 0 \leq k < s_2 :$$
$$0 \leq k + o_2 < n_2 \wedge 0 \leq j + o_1 < n_1 \wedge 0 \leq i + o_0$$
$$\Rightarrow o_1 \leq n_1 - s_1 \wedge o_2 \leq n_2 - s_2$$

Why validity conditions?

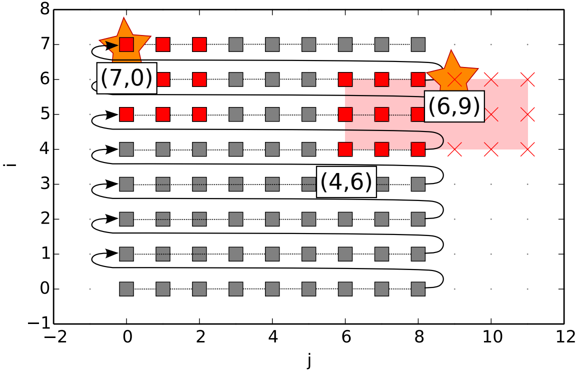
- Array size ($n_0 = 8, n_1 = 9$)
- Subarray offset ($o_0 = 1, o_1 = 3$), size ($s_0 = 3, s_1 = 6$).



- Run-time condition: $o_1 \leq n_1 - s_1 \Rightarrow 3 \leq 9 - 6 \rightarrow \top$

Why validity conditions?

- Array size ($n_0 = 8, n_1 = 9$)
- Subarray offset ($o_0 = 4, o_1 = 6$), size ($s_0 = 3, s_1 = 6$).



- Run-time condition: $o_1 \leq n_1 - s_1 \Rightarrow 6 \leq 9 - 6 \Rightarrow \bot$
- `A[6][9]` and `A[7][0]` alias ↯

Delinearization in LLVM's ScalarEvolution

```
// Delinearization of a single access
void delinearize(const SCEV *Expr,
    SmallVectorImpl<const SCEV *> &Subscripts,
    SmallVectorImpl<const SCEV *> &Sizes,
    const SCEV *ElementSize);

// Functions to derive a delinearization for a set of accesses:
void collectParametricTerms(const SCEV *Expr,
    SmallVectorImpl<const SCEV *> &Terms);
void findArrayDimensions(SmallVectorImpl<const SCEV *> &Terms,
    SmallVectorImpl<const SCEV *> &Sizes,
    const SCEV *ElementSize);
void computeAccessFunctions(
    const SCEV *Expr, SmallVectorImpl<const SCEV *> &Subscripts,
    SmallVectorImpl<const SCEV *> &Sizes);
```

! Validity conditions still need to be generated (available in Polly) !

Using shared memory: Apply a simple mapping function

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  int i = 2 * b0 + t0; int j = 2 * b1 + t1;
  S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                         + A[i  ][j+1] + A[i  ][j-1];
}
```

Original access relation: $\{S[i,j] \to A[i,j]\}$

Using shared memory: Apply a simple mapping function

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  int i = 2 * b0 + t0; int j = 2 * b1 + t1;
  S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                         + A[i  ][j+1] + A[i  ][j-1];
}
```

Original access relation: $\{S[i,j] \to A[i,j]\}$
Block mapping: $\{S[i,j] \to blocks[floor(i/2), floor(j,2)]\}$

Using shared memory: Apply a simple mapping function

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  int i = 2 * b0 + t0; int j = 2 * b1 + t1;
  S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                         + A[i  ][j+1] + A[i  ][j-1];
}
```

Original access relation: $\{S[i,j] \rightarrow A[i,j]\}$

Block mapping: $\{S[i,j] \rightarrow blocks[floor(i/2), floor(j,2)]\}$

Per-block accesses: $\{blocks[b0, b1] \rightarrow A[i,j] \mid$

$$2 * b0 - 1 \leq i \leq 2 * b0 + 1 \wedge$$
$$2 * b1 - 1 \leq j \leq 2 * b1 + 1\}$$

Using shared memory: Apply a simple mapping function

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  int i = 2 * b0 + t0; int j = 2 * b1 + t1;
  S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                         + A[i  ][j+1] + A[i  ][j-1];
}
```

Original access relation: $\{S[i,j] \rightarrow A[i,j]\}$

Block mapping: $\{S[i,j] \rightarrow blocks[floor(i/2), floor(j,2)]\}$

Per-block accesses: $\{blocks[b0, b1] \rightarrow A[i,j] \mid$

$$2 * b0 - 1 \leq i \leq 2 * b0 + 1 \wedge$$
$$2 * b1 - 1 \leq j \leq 2 * b1 + 1\}$$

Minimal element accessed in block: $(2b0 - 1, 2b1 - 1)$

Using shared memory: Apply a simple mapping function

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  int i = 2 * b0 + t0; int j = 2 * b1 + t1;
  S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                         + A[i  ][j+1] + A[i  ][j-1];
}
```

Original access relation: $\{S[i,j] \rightarrow A[i,j]\}$

Block mapping: $\{S[i,j] \rightarrow blocks[floor(i/2), floor(j,2)]\}$

Per-block accesses: $\{blocks[b0, b1] \rightarrow A[i,j] \mid 2*b0-1 \leq i \leq 2*b0+1 \wedge 2*b1-1 \leq j \leq 2*b1+1\}$

Minimal element accessed in block: $(2b0-1, 2b1-1)$

Extend of accessed region: $(3,3)$

Using shared memory: Apply a simple mapping function

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  int i = 2 * b0 + t0; int j = 2 * b1 + t1;
  S:  B[i][j] =  A[i][j] + A[i+1][j  ] + A[i-1][j  ]
                         + A[i  ][j+1] + A[i  ][j-1];
}
```

Original access relation: $\{S[i,j] \rightarrow A[i,j]\}$

Block mapping: $\{S[i,j] \rightarrow blocks[floor(i/2), floor(j,2)]\}$

Per-block accesses: $\{blocks[b0, b1] \rightarrow A[i,j] \mid$
$2 * b0 - 1 \leq i \leq 2 * b0 + 1 \wedge$
$2 * b1 - 1 \leq j \leq 2 * b1 + 1\}$

Minimal element accessed in block: $(2b0 - 1, 2b1 - 1)$

Extend of accessed region: $(3, 3)$

Map to shared memory: $\{A[i,j] \rightarrow A_{\text{shared}}[i - 2b0 + 1, j - 2b1 + 1]\}$

Kernel code using shared memory

```
void kernel(float A[][6], float B[][6]) {
  int b0 = blockIdx.y; int b1 = blockIdx.x;
  int t0 = threadIdx.y; int t1 = threadIdx.x;
  __shared A_shared[3][3];

  A_shared[t0][t1] = A[2 * b0 + t0 - 1][2 * b1 + t1 - 1];
  if (t0 < 1)
    A_shared[t0+2][t1] = A[2 * b0 + t0 + 1][2 * b1 + t1 - 1];
  if (t1 < 1)
    A_shared[t0][t1+2] = A[2 * b0 + t0 - 1][2 * b1 + t1 + 1];
  if (t0 < 1 && t1 < 1)
    A_shared[t0+2][t1+2] = A[2 * b0 + t0 + 1][2 * b1 + t1 + 1];
  __sync_synchronize();
  S:  B[i][j] =   A_shared[t0+1][t1+1]
              + A_shared[t0+2][t1+1] + A_shared[t0+0][t1+1]
              + A_shared[t0+1][t1+2] + A_shared[i0+1][i1+0];
}
```

Heterogeneous Compute in Polly

- Precise memory modeling enables compiler-driven memory management.
- Polly recovers necessary information to reason about multi-dimensionality.
- Complex memory accesses transformations made easy.
- Sophisticated kernel generation with Polly