Friedrich-Alexander-Universität
Programming Systems Group

Employing polyhedral methods to optimize stencils on FPGAs with stencil-specific caches, data reuse, and wide data bursts

Florian Mayer, Julian Brandner, and Michael Philippsen

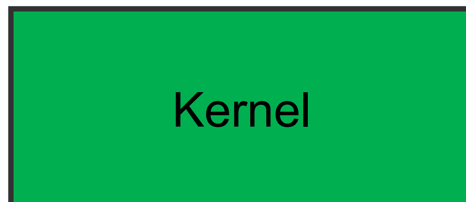
Introduction
SCoP, Tiling, and Cache Buffers
Approach
Evaluation and Results

Why offload to the FPGA?

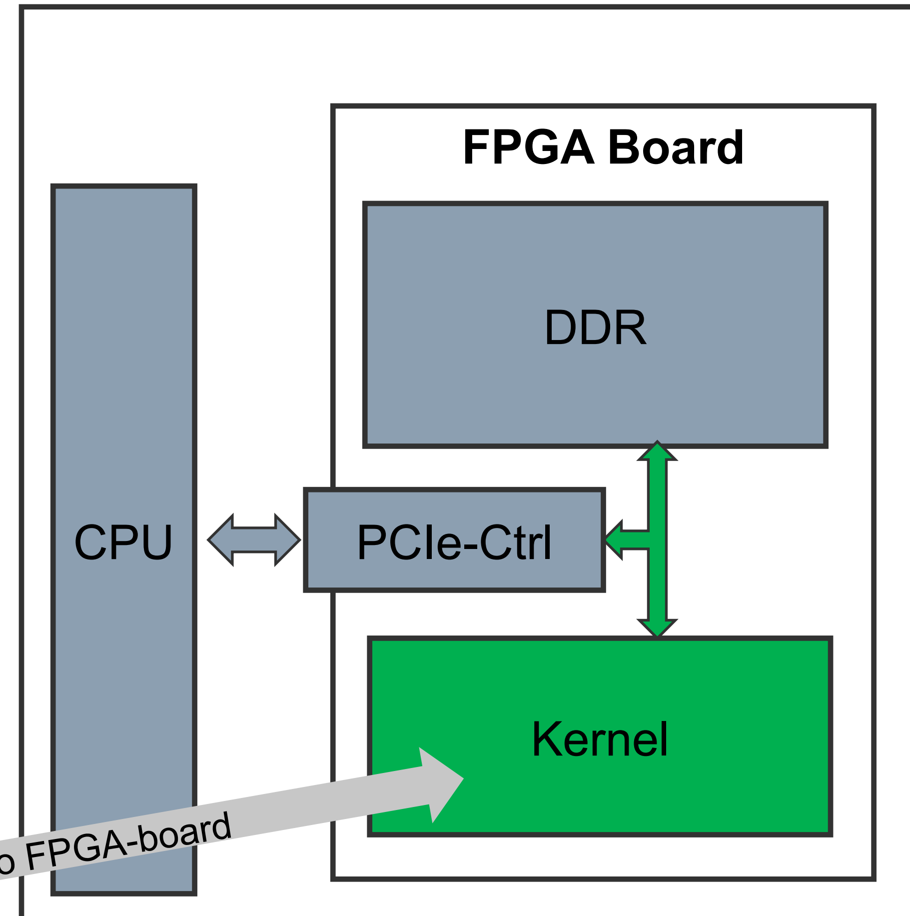
```
init_data(...);

#pragma offload ...
{
  for (i=1; i<N-1; i++)
    for (j=1; j<N-1; j++)
      for (k=1; k<N-1; k++)
        S(i,j,k);
}
```

HLS



placed onto FPGA-board

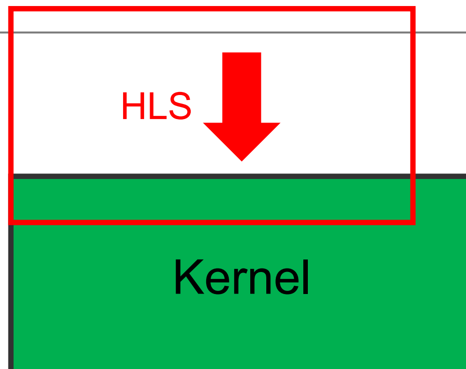


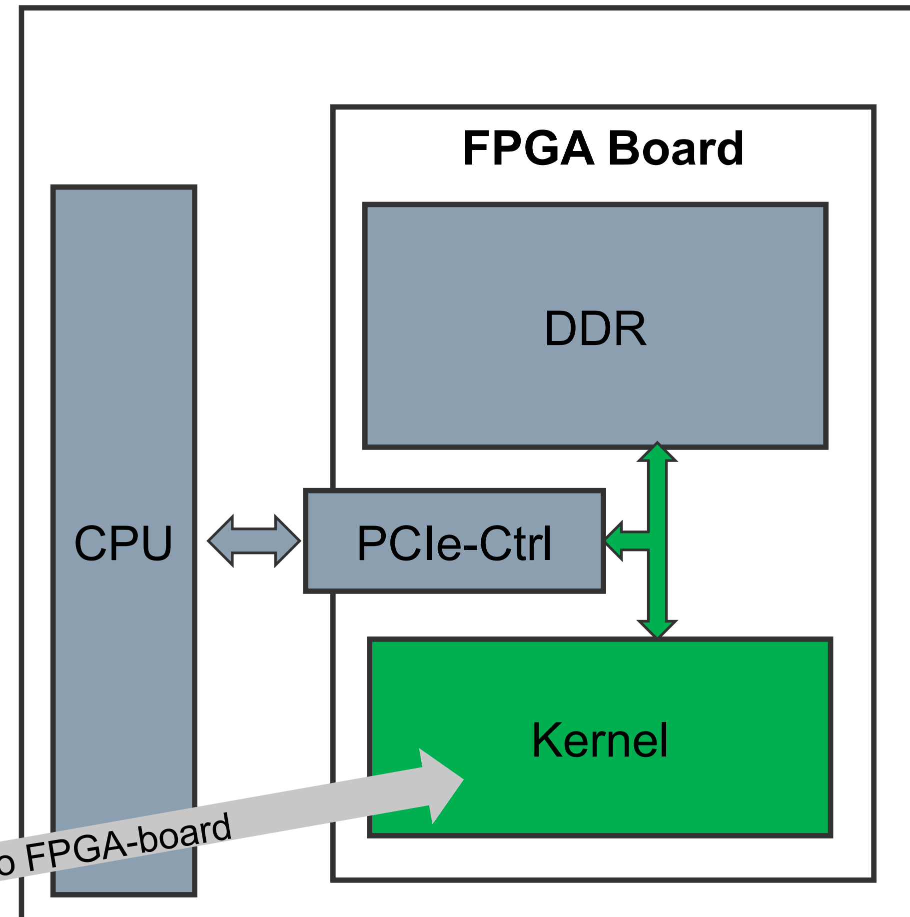
Introduction

Why offload to the FPGA?

```
init_data(...);

#pragma offload ...
{
  for (i=1; i<N-1; i++)
    for (j=1; j<N-1; j++)
      for (k=1; k<N-1; k++)
        S(i,j,k);
}
```





Problems

Does a modern HLS generate fast and efficient hardware for a stencil?

Unfortunately, no.

Current HLS do not handle memory-bound algorithms well.

Problems

FAU

Does a modern HLS generate fast and efficient hardware for a stencil?

Unfortunately, no.

Current HLS do not handle memory-bound algorithms well.

Problems:

1. **Unused parallelism.**
2. **Unnecessary data transfers.**
3. **Latencies of data transfers.**
4. **Insufficient data throughput.**

Problems

FAU

Does a modern HLS generate fast and efficient hardware for a stencil?

Unfortunately, no.

Current HLS do not handle memory-bound algorithms well.

Problems:

1. **How to exploit parallelism?**
2. **How to reduce unnecessary data transfers?**
3. **How to hide latencies of data transfers?**
4. **How to increase data throughput?**

Problem 1: How to exploit parallelism?

```
for (i=1; i<N-1, i++)
 for (j=1; j<N-1; j++)
  for (k=1; k<N-1; k++)
   S(i,j,k);
```

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+=SZ(k))
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<= min(...); k++)
      S(i,j,k)
     // Goal: Let HLS generate
     // parallel hardware circuits
```

1. How to exploit parallelism?

→ **Solution: Loop Tiling + Unrolling!**

Problem 2: How to reduce unnecessary data transfers?

```
for (i=1; i<N-1, i++)
 for (j=1; j<N-1; j++)
  for (k=1; k<N-1; k++)
   S(i,j,k);
```

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+=SZ(k))
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<= min(...); k++)
      S(i,j,k)
```

Problem 2: How to reduce unnecessary data transfers?

```
for (i=1; i<N-1, i++)
 for (j=1; j<N-1; j++)
  for (k=1; k<N-1; k++)
   S(i,j,k);
```

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+=SZ(k))
   float cache_buf[...];
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<= min(...); k++)
      S'(i,j,k)
      // Goal: Let HLS create cache-like
      // hardware circuits
      // within the kernel
```

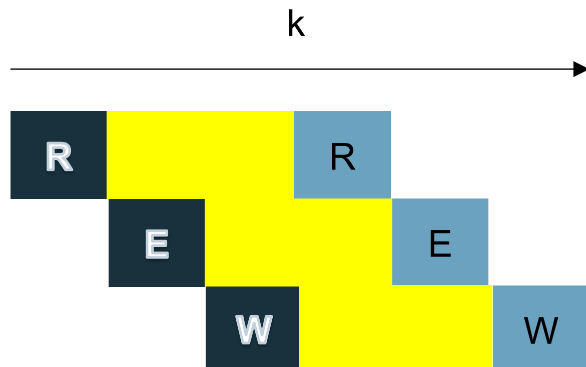
2. How to reduce unnecessary data transfers?

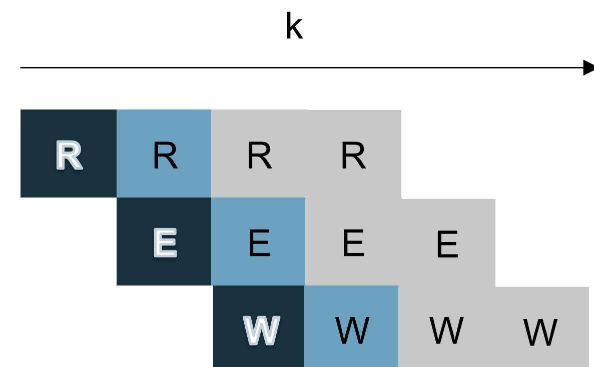
→ **Solution: Loop Tiling + Stencil-specific caches as part of kernel!**

Problem 3: How to hide latencies of data transfers?

```
for (i=1; i<N-1, i++)
 for (j=1; j<N-1; j++)
  for (k=1; k<N-1; k++)
   S(i,j,k);
```

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+=SZ(k))
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<= min(...); k++)
      S(i,j,k)
```



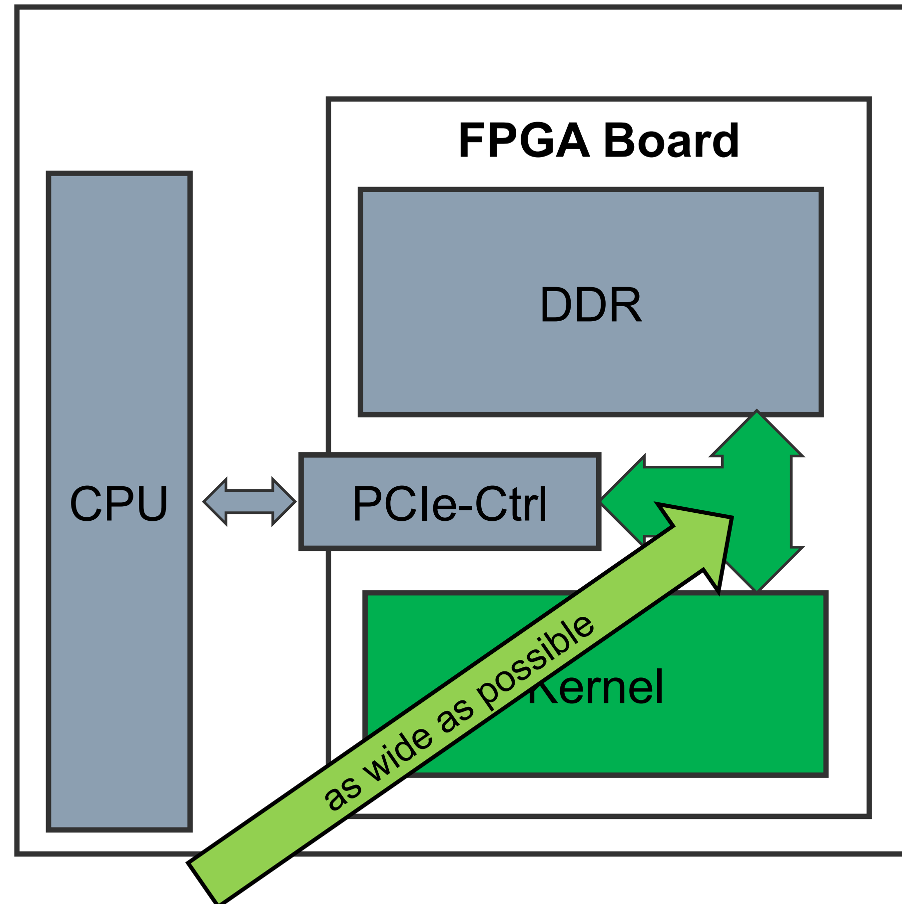


3. How to hide latencies of data transfers?

→ **Solution: Pipelined loops + concurrent cache operations!**

Problem 4: How to increase data througput?

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+           ) {
   Cache c0[.
   fill_cache(c0, ...)
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<        (...); k++)
      S(i,j,k);
   }
   flush_cache(c0, ...);
```



4. How to increase data throughput?

→ **Solution: Bursts + wide ports!**

Introduction

SCoP, Tiling, and Cache Buffers

Approach

Evaluation and Results

SCoP

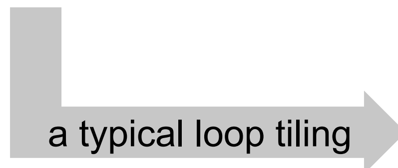
```
for (i=1; i<N-1, i++)
 for (j=1; j<N-1; j++)
  for (k=1; k<N-1; k++)
   S(i,j,k);
```

- $Dom = [N] \to \{\ S[i,j,k]\colon 1 \leq i < N-1\ and\ 1 \leq j < N-1\ and\ 1 \leq k < N-1\ \}$
- $Sch = \{S[i,j,k] \to O[i,j,k]\}$
- Access Relations:
  - $R_0 = \{S[i,j,k] \to O[i,j,k]\}$
  - $R_1 = \{S[i,j,k] \to O[v,k,j]\}$
  - ...
  - $W_0 = \{S[i,j,k] \to O[i,j,k]\}$

Loop Tiling

Original loop nest:

```
for (i=1; i<N-1, i++)
 for (j=1; j<N-1; j++)
  for (k=1; k<N-1; k++)
   S(i,j,k);
```

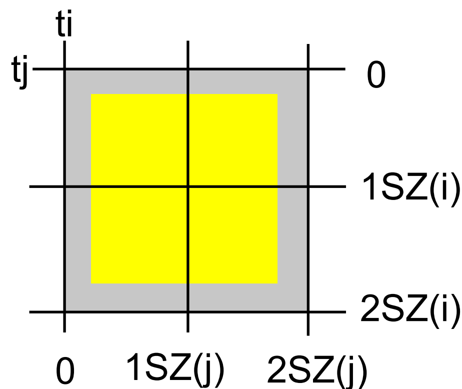


Tiled loop nest:

```
for (ti=ST(i); ti+=SZ(i))
 for (tj=ST(j); tj+=SZ(j))
  for (tk=ST(k); tk+=SZ(k))
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<= min(...); k++)
      S(i,j,k)
```

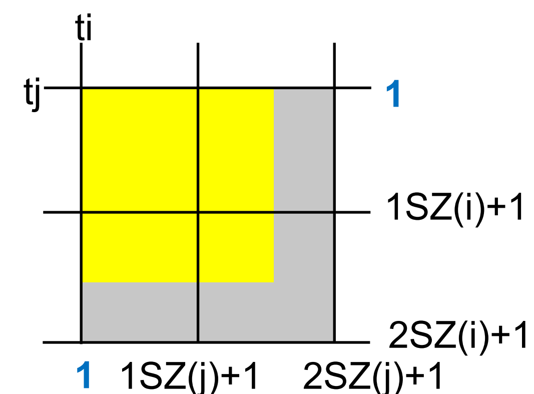
inter-tile (the three `ti`/`tj`/`tk` loops)

intra-tile (the three `i`/`j`/`k` loops)



ti/tj start at 0 → 4 partial tiles.

tk/k not shown.



ti/tj start at 1 → 3 partial, 1 full tile.

tk/k not shown.

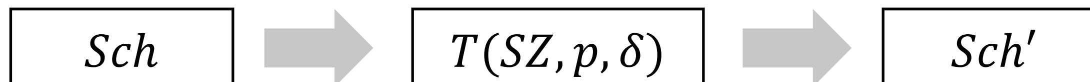
Loop Tiling

- General Tiling Transformation T:

  $$T(SZ, p, \delta) = \{\, \mathrm{O}[\mathrm{i}_1, \dots \mathrm{i}_\mathrm{d}] \rightarrow O\left[ti_1, \dots, ti_d, p(i_1) + \delta^i_{p(i_1)}, \dots, p(i_d) + \delta^i_{p(i_d)}\right]: \mathrm{C} \}$$

  $$\text{where } C = \bigwedge^{d} (ti_x + \delta^o_x)\, mod\, SZ_x = 0 \;\wedge \mathrm{ti_x} \leq \mathrm{i_x} < \mathrm{ti_x} + \mathrm{SZ_x}$$

- We find the parameters:
  - $SZ = (SZ_1, \dots, SZ_d)$ // tile sizes
  - $p = (i_1, \dots, i_d)$ // permutation
  - $\delta = \left(\delta^o_1, \delta^i_1, \dots, \delta^o_d, \delta^i_d\right)$ // deltas (inner + outer)



Loop Tiling: δ matters on FPGAS

FAU

```
for (i=1; i<N; i+=1) { A[i] = B[i-1] + B[i+1]; }
```

Normal form:

$$T\big(SZ = (32), p = (i), \delta = (0,0)\big) = \{O[i] \rightarrow O[ti, i+0] : (ti+0)\ mod\ 32 = 0\ and\ ti \leq i < ti + 32\}$$

```
for (ti=0; ti<N; ti+=32)
  for (i=max(1, ti); i<=min(N-1, ti+31); i+=1)
    A[i] = B[i-1] + B[i+1];
```

HLS cannot handle max and min.

Reduce number of partial tiles by eliminating the max functions.

With outer delta:

$$T\big(SZ = (32), p = (i), \delta = (-\mathbf{1}, 0)\big) = \{O[i] \rightarrow O[ti, i+0] : (ti\ -1)\ mod\ 32 = 0\ and\ ti \leq i < ti + 32\}$$

```
for (ti=1; ti<N; ti+=32)
  for (i=ti; i<=min(N-1, ti+31); i+=1)
    A[i] = B[i-1] + B[i+1];
```

HLS cannot handle symbolic bounds.

Eliminate symbolic lower bounds.

With outer + inner delta:

$$T\big(SZ = (32), p = (i), \delta = (-\mathbf{1}, -\boldsymbol{ti})\big) = \{O[i] \rightarrow O[ti, i\ - ti] : (ti\ -1)\ mod\ 32 = 0\ and\ ti \leq i < ti + 32\}$$

```
for (ti=1; ti<N; ti+=32)
  for (i=0; i<=min(31, N-ti-1); i+=1)
    A[i+ti] = B[i-1+ti] + B[i+1+ti];
```

Cache Types

HLS-FPGA does not provide efficient cache hardware

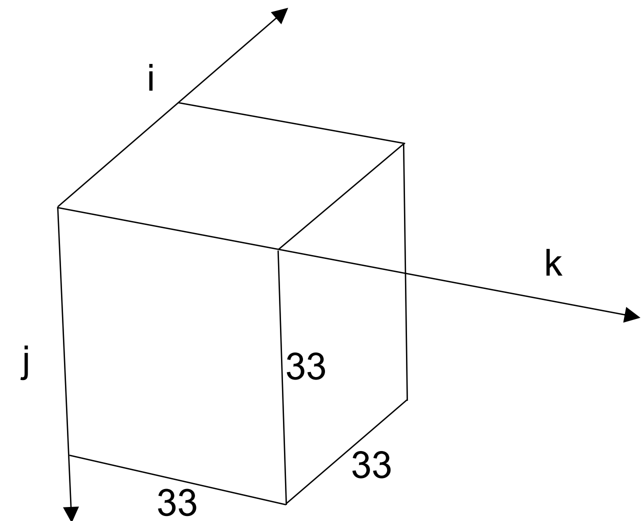
- Each array access corresponds to a working set.
  → Create one cache buffer per working set.
- A ***working set*** is the set of elements that an array access uses during a traverse through a tile.
- Depending on the loop permutation fuse working sets to use fewer caches.
- Three cache types:
  1. Full cache.
  2. Chunk cache.
  3. Line cache.

  } applicability depends on permutation of inter-tile loop.
- cost(Full) > cost(Chunk) > cost(Line).

Cache Types: Full Cache

```
for(ti...+=32) for(tj...+=32) for(tk...+=32)
 for (i=max(...); i<=min(...); i++)
  for (j=max(...); j<=min(...); j++)
   for (k=max(...); k<= min(...); k++)
S:     V[i,k,j] = V[i,k,j] + A[i,j,k]
                 + A[i+1,j+1,k+1];
```

```
for(ti...+=32) for(tj...+=32) for(tk...+=32) {
 fill_cache(A', A, ...);
 for (i=max(...); i<=min(...); i++)
  for (j=max(...); j<=min(...); j++)
   for (k=max(...); k<= min(...); k++)
S:     V[i,k,j] = V[i,k,j] + A'[...]
                 + A'[...];
 flush_cache(A', A, ...);
}
```

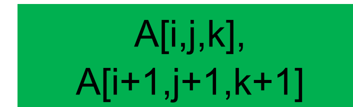


Full cache dimension: A'[33][33][33]

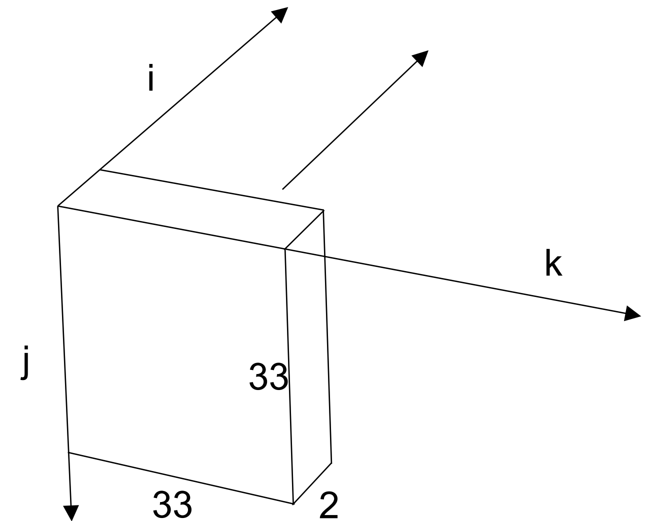
Cache Types: Chunk Cache

```
for(ti...+=32) for(tj...+=32) for(tk...+=32)
 for (i=ti+X; i<=ti+X; i++)
  for (j=max(...); j<=min(...); j++)
   for (k=max(...); k<= min(...); k++)
S:      V[i,k,j] = V[i,k,j] + A[i,j,k]
                 + A[i+1,j+1,k+1];
```

```
for(ti...+=32) for(tj...+=32) for(tk...+=32) {
 fill_cache(A', A, <init>);
 for (i=max(...); i<=min(...); i++) {
  fill_cache(A', A, <next(i)>);
  for (j=max(...); j<=min(...); j++)
   fill_cache(A', A, <next(k)>);
   for (k=max(...); k<= min(...); k++)
S:  V[i,k,j]=V[i,k,j]+A'[...]+A'[...];
   shift(A', A', ...);
 } }
```



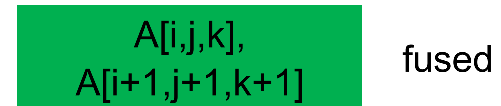
fused



Chunk cache dimension: A'[2][33][33]

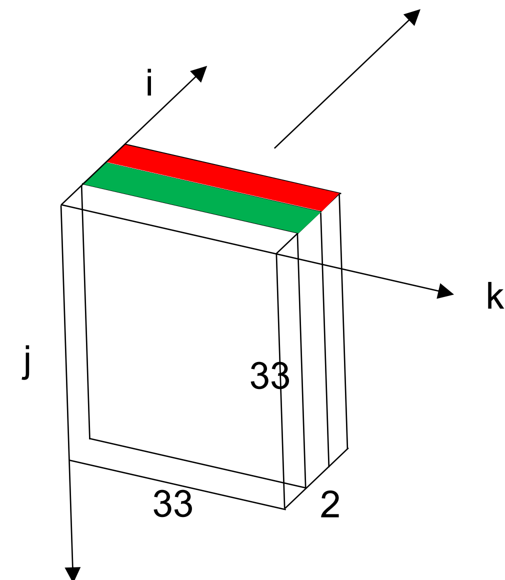
Cache Types: Chunk Cache

```
for(ti...+=32) for(tj...+=32) for(tk...+=32)
 for (i=ti+X; i<=ti+X; i++)
  for (j=max(...); j<=min(...); j++)
   for (k=max(...); k<= min(...); k++)
S:     V[i,k,j] = V[i,k,j] + A[i,j,k]
                 + A[i+1,j+1,k+1];
```



fused

```
for(ti...+=32) for(tj...+=32) for(tk...+=32) {
 fill_cache(A', A, <init>);
 for (i=max(...); i<=min(...); i++) {
  fill_cache(A', A, <next(i)>);
  for (j=max(...); j<=min(...); j++)
   fill_cache(A', A, <next(k)>);
   for (k=max(...); k<= min(...); k++)
S:  V[i,k,j]=V[i,k,j]+A'[...]+A'[...];
   shift(A', A', ...);
 } }
```

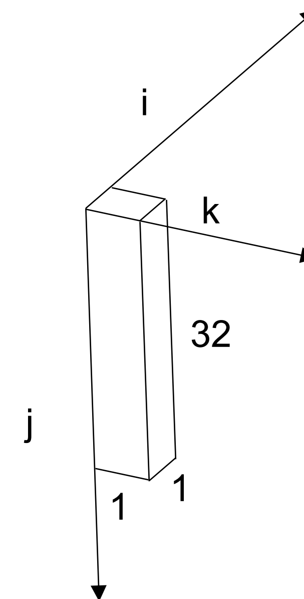


Cache Types: Line Cache

```
for(ti...+=32) for(tj...+=32) for(tk...+=32)
 for (i=ti+X; i<=ti+X; i++)
  for (j=tj+Y; j<=tj+Y; j++)
   for (k=max(...); k<= min(...); k++)
S:     V[i,k,j] = V[i,k,j] + A[i,j,k]
                + A[i+1,j+1,k+1];
```

fused

```
for(ti...+=32) for(tj...+=32) for(tk...+=32) {
 for (i=max(...); i<=min(...); i++)
  for (j=max(...); j<=min(...); j++) {
   fill_cache(V', V, <next(k)>);
   for (k=max(...); k<= min(...); k++)
    V'[...]=V'[...]+A[i,j,k]+A[i+1,j+1,k+1];
   flush_cache(V', V);
  }
}
```



Line cache dimension: V'[1][1][32]

Introduction
SCoP, Tiling, and Cache Buffers
Approach
Evaluation and Results

Approach: Applicability

- n-point stencil codes that process m-dimensional arrays.
- SCoP-encoded stencils
  - that are canonical loop nests,
  - that only use constant loop increments, and
  - that iterate through rectangular iteration domains.
- Data accesses that are uniform, i.e., of the form `i` ± `c`,
  - `i` is the loop counter and `c` is a constant.
  - Approach leaves any non-uniform access as is.
- In-place stenicls whose data dependencies allow for tiling.

Approach: Bird's eye view

1. Pick a tiling permutation p.
2. Pick cache types.
3. Pick tiling deltas δ.
4. Increase throughput via bursts over wide ports.

Approach: Bird's eye view

1. **Pick a tiling permutation *p*.**
2. Pick cache types.
3. Pick tiling deltas δ.
4. Increase throughput via bursts over wide ports.

Approach: 1. Pick permutation *p*

FAU

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+=SZ(k))
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:      V[i,k,j] = V[i,k,j] + A[i,j,k]
                 + A[i+1,j+1,k+1];
```

Intra-tile loops.

- Depending on the data dependences in S, many intra-tile permutations possible.
- Steps (incomplete):
  - Generate intra-tile loop permutations.
  - Filter out intra-tile permutations that violate data dependences.
  - Pick caches for each permutation. ← Not relevant for CPUs/GPUs as their caches are fixed
  - Pick the permutation with the best caches.

Approach: Bird's eye view

1. Pick a tiling permutation p.
2. **Pick cache types.**
3. Pick tiling deltas δ.
4. Increase throughput via bursts over wide ports.

Approach: 2. Pick cache types

We pick caches and their types in three steps:

- Reduce number of caches by fusing of working sets.
- Pick a cache type for each working set.
- Determine the size of each cache (= hardware cost).

Approach: 2. Pick cache types – Fusing

Permutation: p = (i, j, k)

```
... // inter-tile loops
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:     V[i,k,j] = V[i,k,j] + A[i,j,k]
                + A[i+1,j+1,k+1];
```

Approach: 2. Pick cache types – Fusing

Permutation: p = (i, j, k)

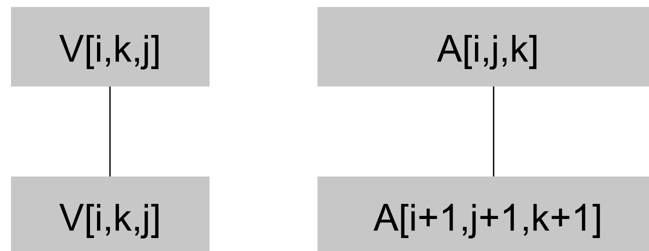
```
... // inter-tile loops
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:       V[i,k,j] = V[i,k,j] + A[i,j,k]
                  + A[i+1,j+1,k+1];
```

V[i,k,j]	A[i,j,k]
V[i,k,j]	A[i+1,j+1,k+1]

Approach: 2. Pick cache types – Fusing

Permutation: p = (i, j, k)

```
... // inter-tile loops
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:      V[i,k,j] = V[i,k,j] + A[i,j,k]
                 + A[i+1,j+1,k+1];
```



Approach: 2. Pick cache types – Fusing

Permutation: p = (i, j, k)

```
... // inter-tile loops
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:      V[i,k,j] = V[i,k,j] + A[i,j,k]
                + A[i+1,j+1,k+1];
```

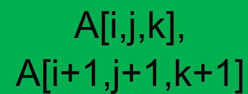
V[i,k,j],
V[i,k,j]

A[i,j,k],
A[i+1,j+1,k+1]

Approach: 2. Pick cache types – Annotation

Permutation: p = (i, j, k)

```
... // inter-tile loops
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:     V[i,k,j] = V[i,k,j] + A[i,j,k]
                + A[i+1,j+1,k+1];
```



Mark green, if accesses are aligned with the innermost loop

Approach: 2. Pick cache types – Algorithm

If (node is NOT green) → Use full cache

Else

If (node is green and data reusable by shifting)
→ Use chunk cache.

Else → Use line cache.

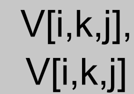
Approach: 2. Pick cache types – Algorithm

If (node is NOT green) → Use full cache

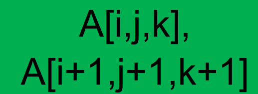
Else

- If (node is green and data reusable by shifting) → Use chunk cache.
- Else → Use line cache.

Permutation: p = (i,j,k)



full cache



chunk cache

Approach: 2. Pick cache types – Algorithm

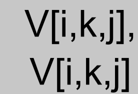
If (node is NOT green) → Use full cache

Else

If (node is green and data reusable by shifting) → Use chunk cache.

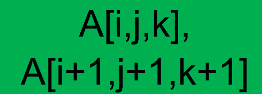
Else → Use line cache.

Permutation: p = (i,j,k)



full cache

V'[32,32,32]



chunk cache

A'[2,33,33]

Approach: Bird's eye view

1. Pick a tiling permutation p.
2. Pick cache types.
3. **Pick tiling deltas δ.**
4. Increase throughput via bursts over wide ports.

Approach: 3. Pick deltas δ

FAU

```
for (ti ...; ti+=SZ(i))
 for (tj ...; tj+=SZ(j))
  for (tk ...; tk+=SZ(k))
   for (i=max(...); i<=min(...); i++)
    for (j=max(...); j<=min(...); j++)
     for (k=max(...); k<=min(...); k++)
S:      ...
```

Approach: 3. Pick deltas δ

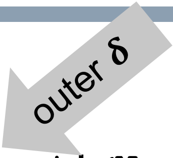
```
for (ti=0; ti<N; ti+=32)
 for (tj=0; tj<N; tj+=32)
  for (tk=0; tk<N; tk+=32)
   for (i=max(1, ti); i<=min(N-1, ti+31); i++)
    for (j=max(1, tj); j<=min(N-1, tj+31); j++)
     for (k=max(1, tk); k<=min(N-1, tk+31); k++)
S:       ...
```

Approach: 3. Pick deltas δ

```
for (ti=0; ti<N; ti+=32)
 for (tj=0; tj<N; tj+=32)
  for (tk=0; tk<N; tk+=32)
   for (i=max(1, ti); i<=min(N-1, ti+31); i++)
    for (j=max(1, tj); j<=min(N-1, tj+31); j++)
     for (k=max(1, tk); k<=min(N-1, tk+31); k++)
S:      ...
```

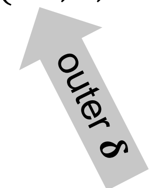
$\delta = (?,?,?,?,?,?)$

Approach: 3. Pick deltas δ




```
for (ti=1; ti<N; ti+=32)
 for (tj=1; tj<N; tj+=32)
  for (tk=1; tk<N; tk+=32)
   for (i=ti; i<=min(N-1, ti+31); i++)
    for (j=tj; j<=min(N-1, tj+31); j++)
     for (k=tk; k<=min(N-1, tk+31); k++)
S:      ...
```

$$\delta = (-1, ?, -1, ?, -1, ?)$$



Approach: 3. Pick deltas δ

inner δ

```
for (ti=1; ti<N; ti+=32)
 for (tj=1; tj<N; tj+=32)
  for (tk=1; tk<N; tk+=32)
   for (i=ti; i<=min(N-1, ti+31); i++)
    for (j=tj; j<=min(N-1, tj+31); j++)
     for (k=tk; k<=min(N-1, tk+31); k++)
S:        ...
```

$$\delta = (-1, -ti, -1, -tj, -1, -tk)$$

inner δ

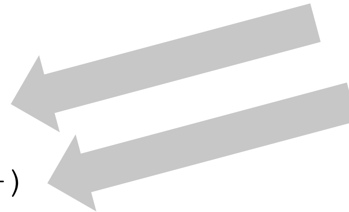
Approach: 3. Pick deltas δ

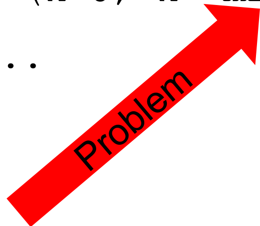
```
for (ti=1; ti<N; ti+=32)
 for (tj=1; tj<N; tj+=32)
  for (tk=1; tk<N; tk+=32)
   for (i=0; i<=min(N-1-ti, 31); i++)
    for (j=0; j<=min(N-1-tj, 31); j++)
     for (k=0; k<=min(N-1-tk, 31); k++)
S:        ...
```

$$\delta = (-1, -ti, -1, -tj, -1, -tk)$$

Approach: 3. Pick deltas δ

```
for (ti=1; ti<N; ti+=32)
 for (tj=1; tj<N; tj+=32)
  for (tk=1; tk<N; tk+=32)
   for (i=0; i<=min(N-1-ti, 31); i++)
    for (j=0; j<=min(N-1-tj, 31); j++)
     for (k=0; k<=min(N-1-tk, 31); k++)
S:     ...
```





Approach: 3. Pick deltas δ

```
for (ti=1; ti<N; ti+=32)
 for (tj=1; tj<N; tj+=32)
  for (tk=1; tk<N; tk+=32)
   for (i=0; i<=min(N-1-ti, 31); i++)
    for (j=0; j<=min(N-1-tj, 31); j++)
     for (k=0; k<=min(N-1-tk, 31); k++)
S:      ...
```

$$Dom = [N] \rightarrow \{ S[i,j,k]: 1 \leq i < N-1 \; and \; 1 \leq j < N-1 \; and \; 1 \leq k < N-1 \}$$

Iteration padding

$$Dom' = [N] \rightarrow \{ S[i,j,k']: \ldots and \; k' > 0 \; and \; 32 * floor(\frac{-1+k'}{32}) <= -3 + N\}$$

Approach: 3. Pick deltas δ

```
for (ti=1; ti<N; ti+=32)
 for (tj=1; tj<N; tj+=32)
  for (tk=1; tk<N; tk+=32)
   for (i=0; i<=min(N-1-ti, 31); i++)
    for (j=0; j<=min(N-1-tj, 31); j++)
     for (k=0; k<=31; k++)
S:      ...
```

Approach: Bird's eye view

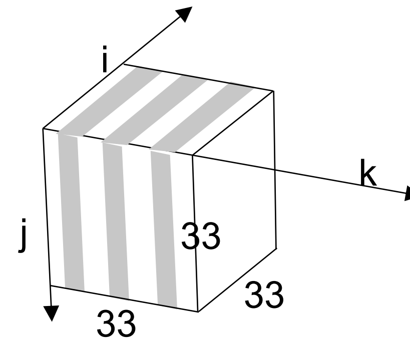
1. Pick a tiling permutation p.
2. Pick cache types.
3. Pick tiling deltas δ.
4. **Increase throughput via bursts over wide ports.**

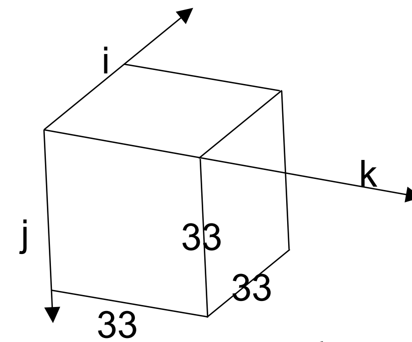
Approach: 4. Data throughput with wide bursts

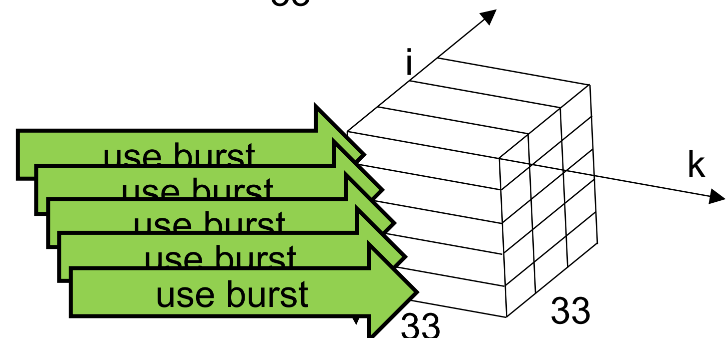
- To increase data throughput for filling up the caches:
  - **Aggregate transfers to bursts.**
  - Widen the port that the burst uses.

Approach: 4. Data throughput with wide bursts

- To increase data throughput for filling up the caches:
  - **Aggregate transfers to bursts.**
  - Widen the port that the burst uses.
- To enable bursts:
  1. Determine the working set (polyhedral methods).
  2. If any: Fill up gaps in gray (polyhedral hull).
  3. Use polyhedral scanning to cut the working set into 1D contiguous strips.
  4. Use a burst for each 1D strip.



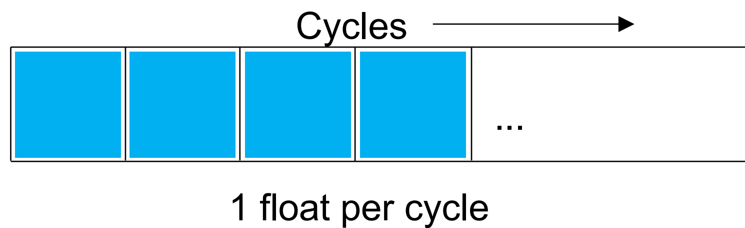


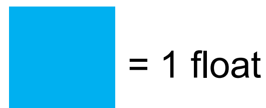


Approach: 4. Data throughput with wide bursts

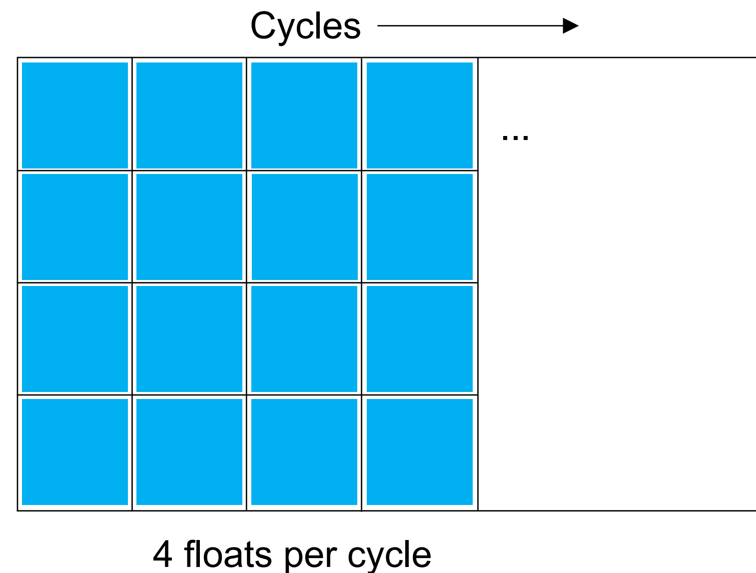
- To increase data throughput for filling up the caches:
  - Aggregate transfers to bursts.
  - **Widen the port that the burst uses.**

Burst over normal 32 bit port. Port width = 1





Burst over normal 32 bit port. Port width = 4

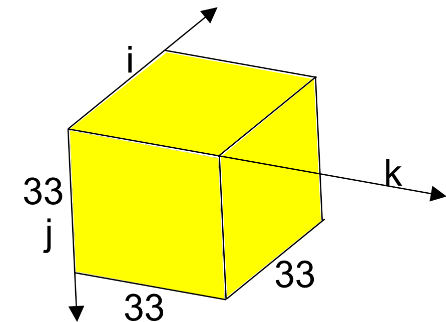


Approach: 4. Data throughput with wide bursts

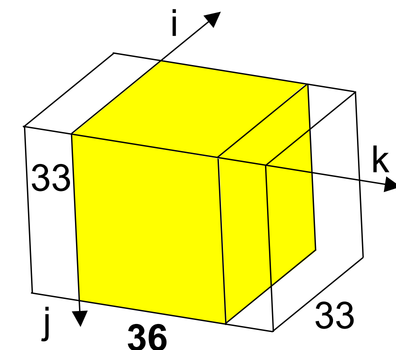
- To increase data throughput for filling up the caches:
  - Aggregate transfers to bursts.
  - **Widen the port that the burst uses.**

- Widening ports is easy → HLS Directives
- But
  - the length
  - and the start address

  of bursts over wide ports must be divisible by the port width.



Data cube



Data cube, padded for port width 4

Introduction
SCoP, Tiling, and Cache Buffers
Approach
Evaluation and Results

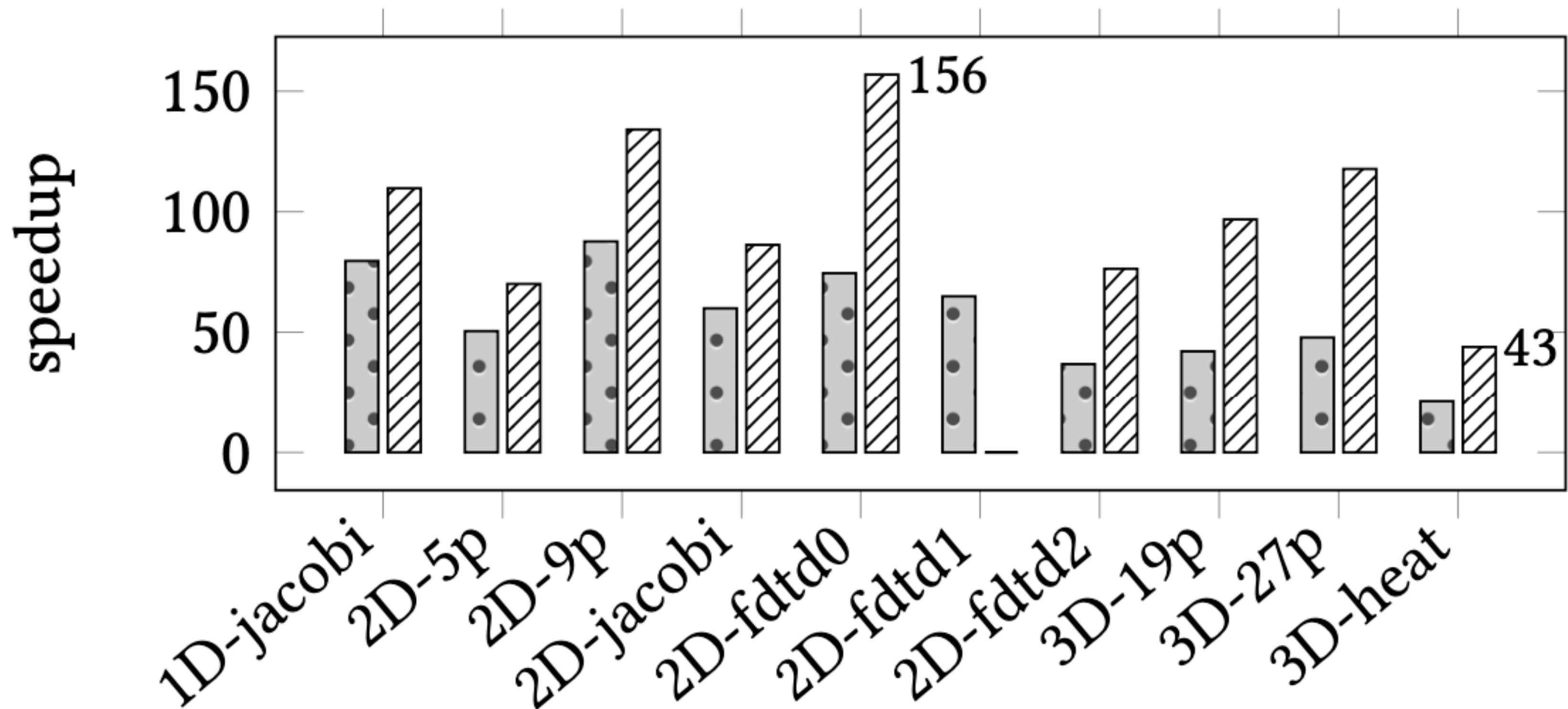
Evaluation

- 10 different benchmark stencil codes
  - Adept benchmarks → 4 Stencils.
  - Polybench benchmarks → 6 Stencils.
- Measured pure runtimes on real FPGA hardware only (VCU118).
- Regardless of data-dimensionality: Stencils work on 64 MiB float arrays.
- Vitis HLS 2021.2 created one 50 MHz FPGA per measurement.

Results

- Speedups from **43x** up to **156x** compared to standard FPGA hardware (generated via HLS without our transformation).
- Findings:
  - Tile sizes have the largest (positive) impact.
  - The port widths have the 2nd largest (positive) impact.

Results – Effect of tile sizes



Large tile sizes: SZ(1D)=(1024), SZ(2D) = (128,128), SZ(3D) = (32,32,32). Port width = 16.

Small tile sizes: SZ(1D)=(512), SZ(2D) = (64,64), SZ(3D) = (16,16,16). Port width = 16.

Empty bar, "__" → HLS fails (with timing error).

Results – Effect of port widths

speedup

150
100
50
0

156
43

1D-jacobi
2D-5p
2D-9p
2D-jacobi
2D-fdtd0
2D-fdtd1
2D-fdtd2
3D-19p
3D-27p
3D-heat

Empty bars, “__” → HLS fails (with timing error).

Results – Comparison with the CPU

Preliminary Data

Stencil	FPGA	i7-4770 3.9 Ghz	Speedup	Fmax
Adept 2d5p	0.0456	0.0872	1.90	261
Adept 2d9p	0.0495	0.1571	3.21	205
Adept 3d19p	0.1449	0.3841	2.65	145
Adept 3d27p	0.3376	0.5522	1.63	55
Poly FDTD0	0.0746	0.0747	1.00	345
Poly FDTD1	0.1098	0.0823	0.75	320
Poly FDTD2	0.1139	0.1026	0.90	260
Poly Heat3d	0.2960	0.2677	0.90	50
Poly Jacobi1d	0.0191	0.0562	2.94	124
Poly Jacobi2d	0.0487	0.1189	2.43	200

seconds

- max(Fmax) ~ 400 MHz.
- Host: Compiled with `-O0`.
- FPGA:
  - Port width: 16
  - SZ(1D)=(1024), SZ(2D) = (128,128), SZ(3D) = (32,32,32)
- Still room for improvement:
  - Increase port width.
  - Increase tile sizes.
- *Limiting factor is the synthesis time.*

Contributions

- Speedup of 43x – 156x over FPGAs generated by vendor HLS with default settings.
- How to use polyhedral methods to accelerate stencils on FPGAs.
- Three FPGA-specific in-kernel cache types and when/how to fuse and pick them.
- How to derive address mapping, padding, bursts, and wide ports for best performance.
- Provide a prototype of the transformation
  including a reproduction package and benchmark codes: