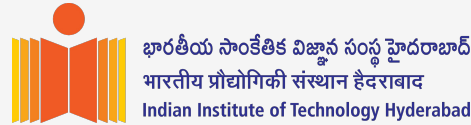


GeMS: Towards Generating Millions of SCoPs

S. VenkataKeerthy, **Nilesh Shah**, Anilava Kundu, Shikhar Jain,
Ramakrishna Upadrasta

Scalable Compilers for Heterogeneous Architectures Lab
Department of Computer Science and Engineering, IIT Hyderabad



IMPACT 2023

Outline

- Motivation
 - Polyhedral compilation using ML techniques
 - Necessity for Labelled SCoPs
- Study of available “*large*” polyhedral codes
 - PolyBench
 - Synthetic loops
- Proposal: Loop generator
 - Realistic codes with compute and memory bounded profiles
 - Initial summary of the SCoPs generated
- Future Directions
 - Improvements: Schedule trees, ...
 - Towards parallel loop generation: SIMD/GPU/OpenMP

Polyhedral Compilation meets ML

Infinite search space



which *affine transformations*?

Based on Integer linear
Programming



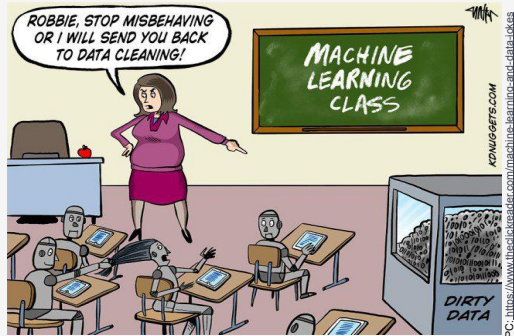
Exponential time complexity
for operations

Based on Heuristics



Can it be improved?

Can ML based techniques help ?



And, ML needs data...

★ *Curse of Dimensionality!*

- Neural networks in general are “**data hungry**”

Vision

ResNets, AlexNets,
etc.

NLP

BERT, BART, GPT3,
etc.

Programs

CodeBERT, CoPilot
(GPT3), etc.

Complex models need massive amount of data.

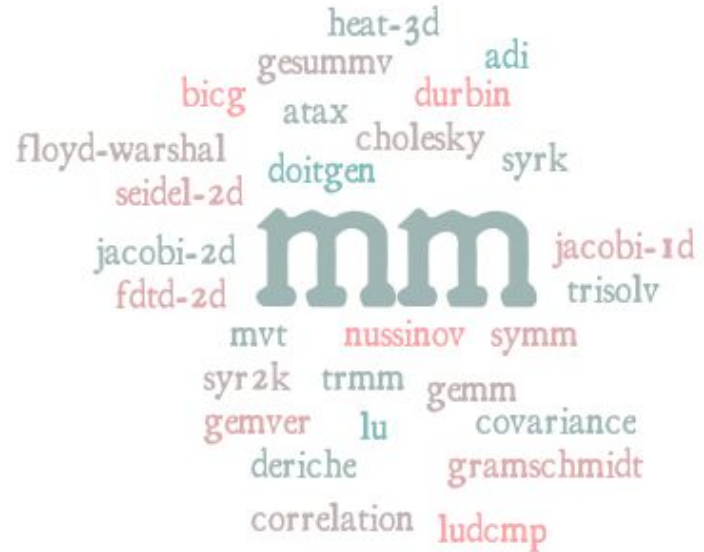
Availability of Polyhedral Programs (data)

Programs are everywhere, polyhedral codes are portions of it..

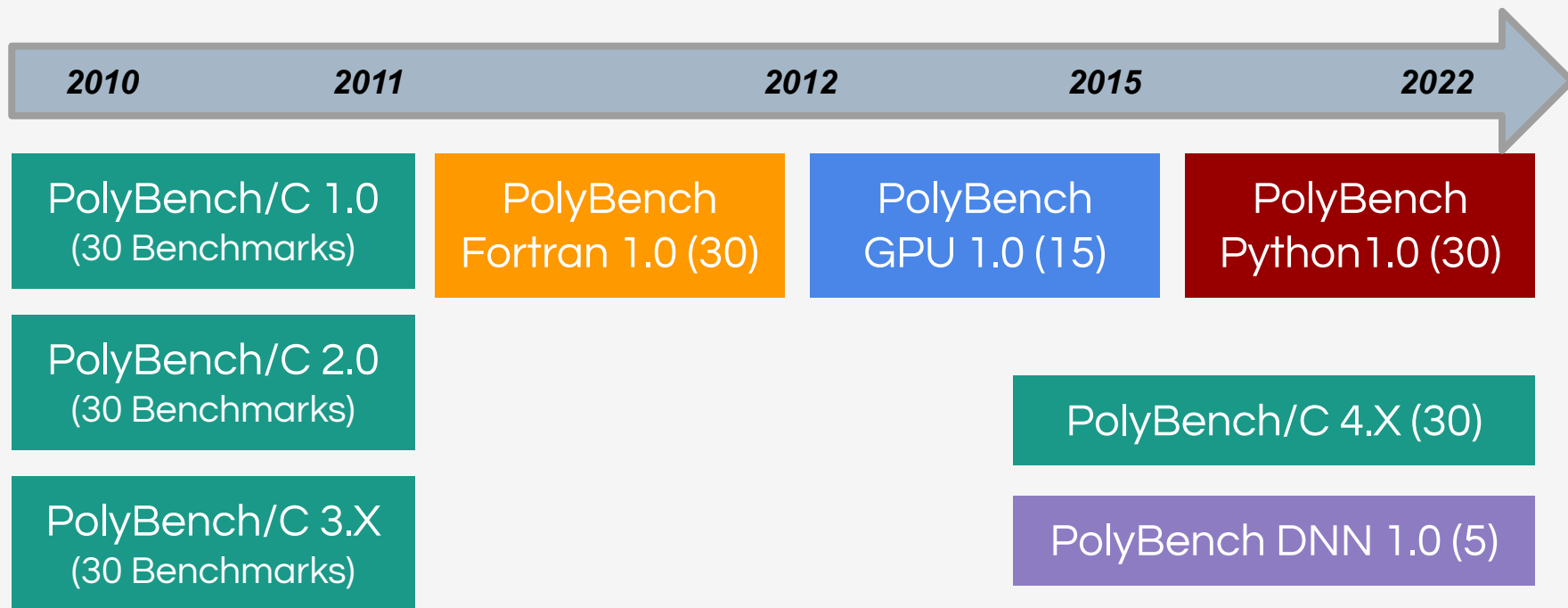
Programs have loops, not all loops are polyhedral
Because of this we cannot rely on the codes that are available prevalently...

Standard polyhedral programs ..

Like **PolyBench** - Linear algebra, Stencils, Data mining, Medley, Dynamic programming



Evolution of PolyBench





Objectives - GeMS

Creating an **automatic** loop generator

- Dump of data without proper characterization is barely useful.
- Hard to pick and choose for downstream applications.

Generating *real-world like* SCoPs with proper labels

- Current work: Categories of *Memory/Compute* bound

Syntactically **valid** codes

- Generating legal, executable binaries

Automatically Generating Compute/Memory Bound Loops

Some Observations

CacheMiss: Modeling **cache hits/misses**

ReuseDist: Modeling locality using **reuse distances**

Boundedness: Modeling (*compute/memory*) **boundedness of a kernel**

MemoryTraffic: Modeling **memory traffic/bandwidth**

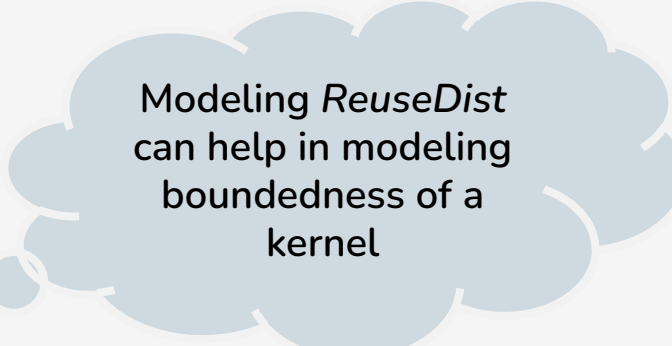
ReuseDist → *CacheMiss*

CacheMiss → *MemoryTraffic*

CacheMiss → *Boundedness*

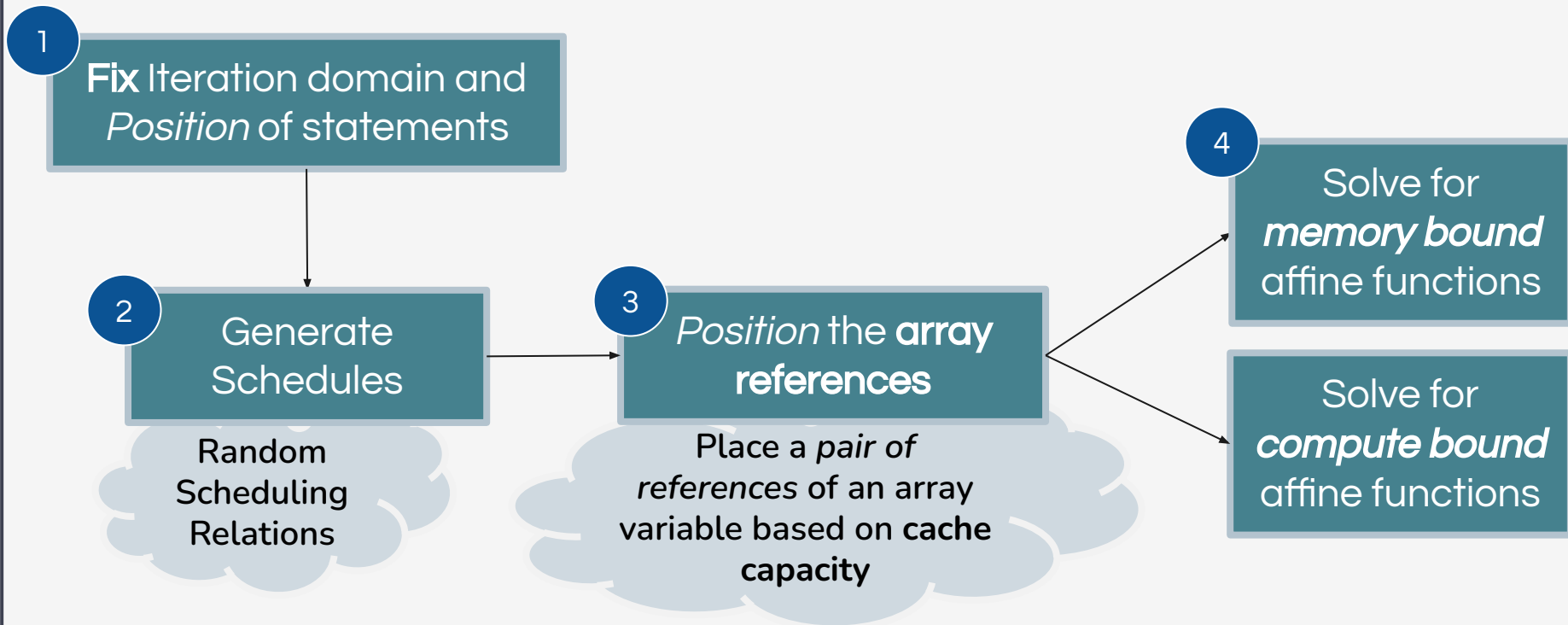
ReuseDist → *MemoryTraffic*

ReuseDist → *Boundedness* •



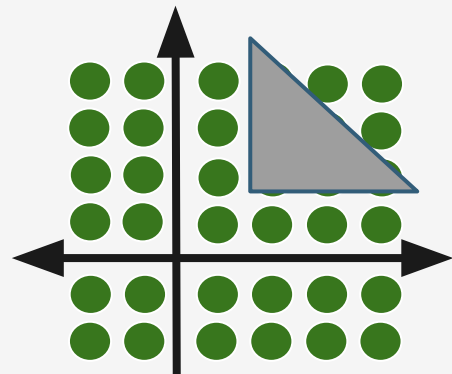
Modeling *ReuseDist*
can help in modeling
boundedness of a
kernel

GeMS: An Overview

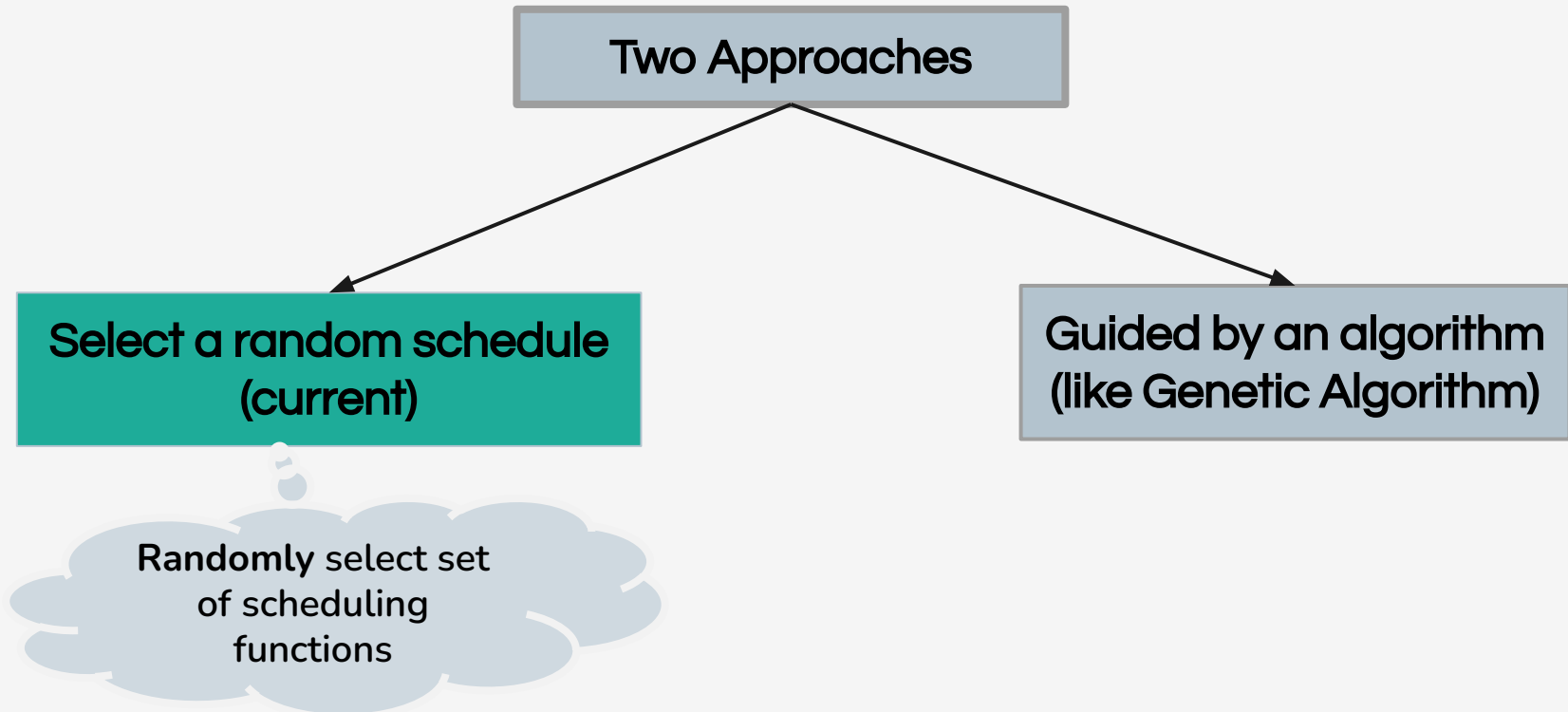


1 Iteration domains and Parameters

- We designed a set of input data sizes
 - Similar to **PolyBench**
- Parameters for data sizes are fixed
 - Iteration domains are **non-parametric**
- Parameterized on **cache size** and **line size**
 - Data reuse modeled analytically with these parameters



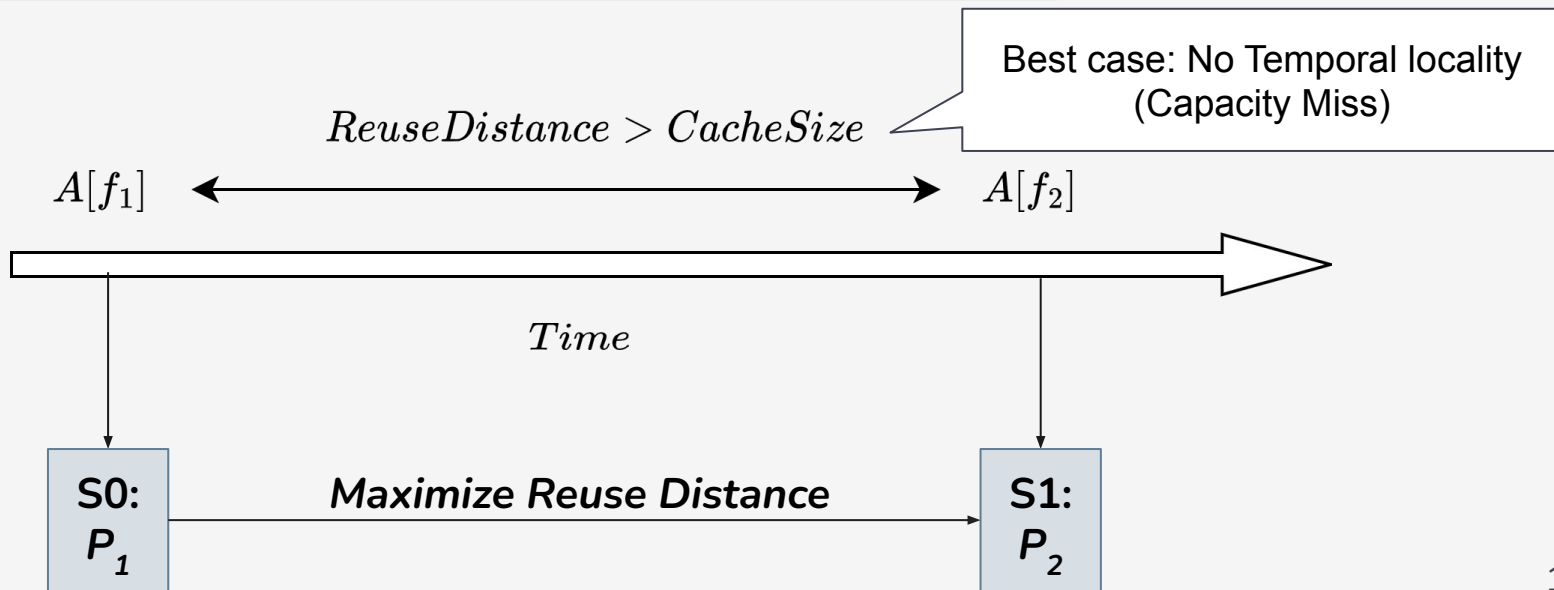
2 Schedule Generation



3 Position the Array variables

Temporal locality is modeled by reuse distance

For a pair of accesses of an array, maximize the reuse distance



Running Example

```
for(int i = 0; i<1000; i++) {  
  S0: C[] = A[] + B[] - E[];  
  for(int j = 0; j<1000; j++) {  
    S1: D[] = D[] - E[] + E[];  
    for(int k = 0; k<1000; k++) {  
      S2: E[] = A[] * B[] * C[];  
      S3: A[] = B[] + C[] * D[];  
    }  
  }  
}
```

Iteration domain for fixed
input size

Position the **array**
references

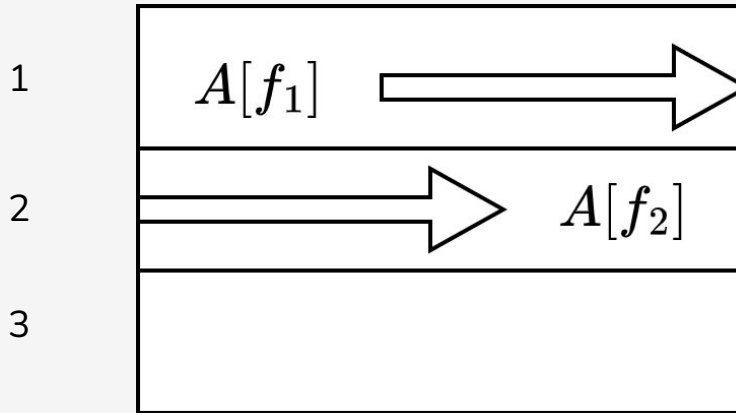
Position the set of **statements**

4

Finding the affine access functions

Each access to an array is accessed in a **different** cache line

Cache line



$$A[f_2] - A[f_1] \geq$$

Elements in cache line

Best case: No Spatial locality
(Compulsory/Conflict Miss)

For each pair of sequential access, find f_1 and f_2 based on maximum memory accesses.

Running Example

Solve for *read* and *write* accesses

```
for(int i = 0; i<1000; i++) {  
  S0: C[16*i] = A[16*i] + B[16*i] - E[16*i];  
  for(int j = 0; j<1000; j++) {  
    S1: D[16*j] = D[24*j] - E[32*j] + E[24*j];  
    for(int k = 0; k<1000; k++) {  
      S2: E[40*k] = A[24*k] * B[24*k] * C[24*k];  
      S3: A[32*k] = B[32*k] + C[32*k] * D[32*k];  
    }  
  }  
}
```

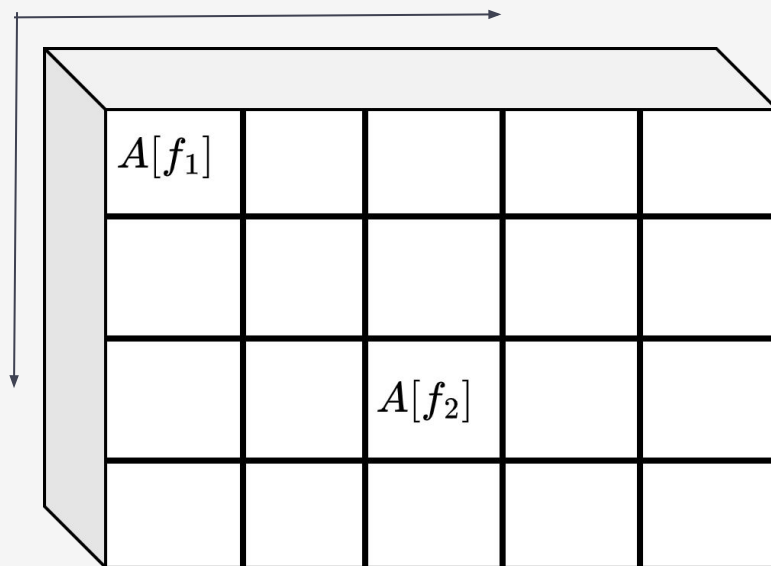

Multi-dimensional array accesses

- For extending 1-D loop to n-D loop generation
- ◆ Use loop transformation like **tiling**, or
 - ◆ **Delinearize** using array input size

$$A[\alpha_1 * t_1 + \alpha_2 * t_2 + \dots + \alpha_d * t_d + c]$$

↓

$$A[f_1][f_2][f_3] \dots [f_d]$$



Generating Compute Bound Loops

Flipping the constraints for generating memory bound loops \Rightarrow compute bound loops

*Each access to an array is accessed in a **same** cache line*

For a pair of accesses of an array, minimize the reuse distance

ReuseDistance \leq Cachesize

Best case: Maximum Temporal
locality (No Capacity Miss)

$$A[f_2] - A[f_1] \leq$$

Elements in cache line

Best case: Maximum Spatial locality
(No Compulsory/Conflict Miss)

*For each pair of sequential access, find f_1 and f_2 based on
minimum cache misses.*

Generated 1-Dimension array examples

LoopID - <LoopDim, ArrayDim, I/P Size, #Stmts, Loop ID>

```
for(int i = 0; i<1000; i++) { Memory Bound
  S0: C[16*i]=A[16*i]+B[16*i]-E[16*i];
  for(int j = 0; j<1200; j++) {
    S1: D[16*j]=D[24*j]-E[32*j]+E[24*j];
    for(int k = 0; k<1100; k++) {
      S2: E[40*k]=A[24*k]*B[24*k]*C[24*k];
      S3: A[32*k]=B[32*k]+C[32*k]*D[32*k];
    }
  }
}
```

LoopID - <3D, 1D, M, 4S, 313>

```
for(int i = 0; i<1500; i++){ Memory Bound
  S0: D[16*i]=D[24*i]-E[32*i]+E[24*i];
  S1: E[40*i] = A[24*i]-B[24*i]*C[24*i];
  for(int j = 0; j<1700; j++){
    for(int k = 0; k<1600; k++){
      S2: C[16*k] = A[16*k] * B[16*k] + E[16*j][k];
      S3: A[32*k] = B[32*k] * C[32*k] + D[32*k];
    }
  }
}
```

LoopID - <3D, 1D, L, 4S, 250>

```
for(int i = 0; i<1500; i++){ Memory Bound
  S0: E[40*i]=A[24*i]+B[24*i]-C[24*i];
  S1: A[32*i]=B[32*i]+C[32*i]*D[32*i];
  for(int j = 0; j<1700; j++){
    S2: C[16*j]=A[16*j]*B[16*j]*E[16*j];
    S3: D[16*j]=D[24*j]+E[32*j]-E[24*j];
  }
}
```

LoopID - <2D, 1D, L, 4S, 45>

```
for(int i = 0; i<1500; i++){ Compute Bound
  S0: D[94*i]=A[94*i]*B[94*i]*C[94*i];
  S1: A[14*i]=B[6*i]+C[16*i]-D[15*i];
  for(int j = 0; j<1700; j++){
    S2: D[6*j]=D[14*j]+E[15*j]-E[7*j];
    for(int k = 0; k<1600; k++){
      S3: C[6*k]=A[7*k]+B[3*k]-E[94*k];
      S4: E[13*k]=A[10*k]*B[11*k]*C[8*k];
    }
  }
}
```

LoopID - <3D, 1D, L, 5S, 743>

Experimentation

Objectives: *Evaluate the generated compute/memory bound kernels.*

- On **two** different test machines
 - ◆ Intel Rocket Lake (i5-11400 @2.6Ghz, 6 cores, 12 MB Cache)
 - ◆ Intel Skylake (Xeon W-2133 @ 3.60GHz, 6 cores, 8.25MB Cache)
- Depict **diversity** of the generated kernels
 - ◆ By estimating the IPC profiles* (Execution Time \propto IPC)
- No **compile time** or **runtime** failures/errors

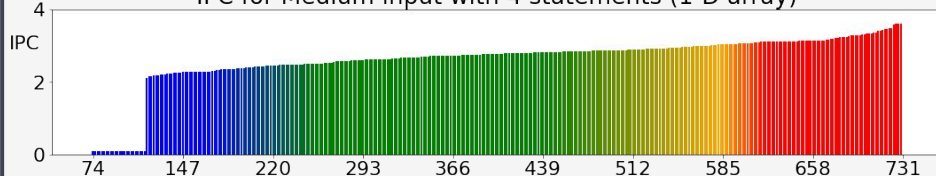
* Using some artifacts provided by *Agner fog* (<https://www.agner.org/optimize/#testp>) - read counters from Intel's PMC hardware suite (reading 1. # of instructions retired & 2. # of cycles completed).

Implementation Details

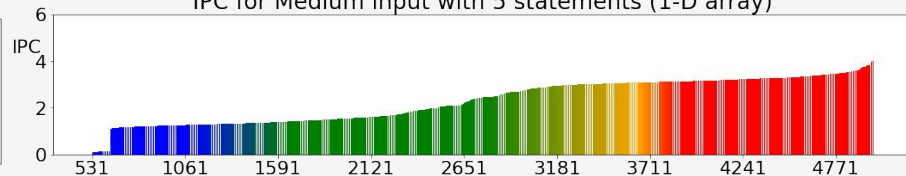
- Current implementation of GeMS
 - ◆ Uses ISLpy and CPLEX
 - ◆ Written in Python
- Labeled SCoPs generated per machine
 - ◆ Memory bound: **12K**
 - ◆ Compute bound: **12K**
- Tested on
 - ◆ 4 different input sizes: **Small(S), Medium (M), Large (L), Extra-Large(XL)**
 - ◆ 3 different loop depths
- Loop identifier: **<LoopDim, ArrayDim, I/P Size, #Stmts, Loop ID>**

Generated loops: IPC Study on Rocket Lake

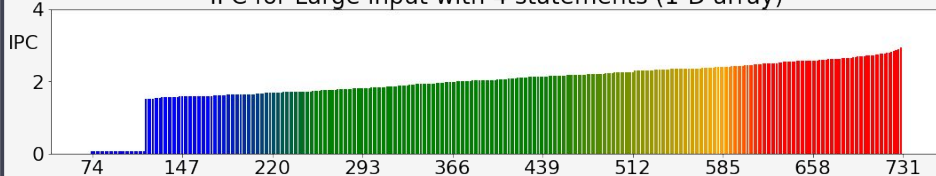
IPC for Medium input with 4 statements (1-D array)



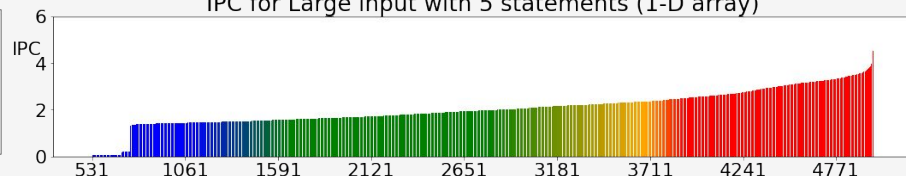
IPC for Medium input with 5 statements (1-D array)



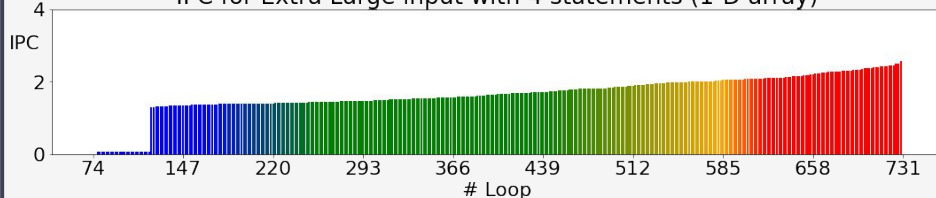
IPC for Large input with 4 statements (1-D array)



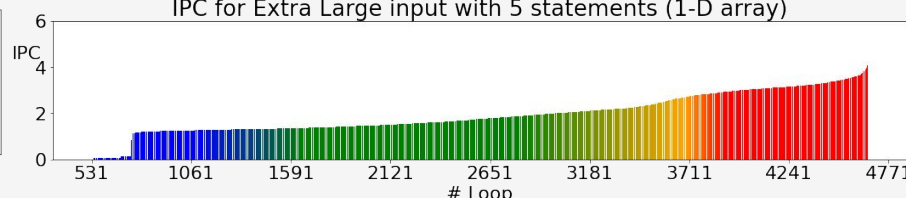
IPC for Large input with 5 statements (1-D array)



IPC for Extra Large input with 4 statements (1-D array)

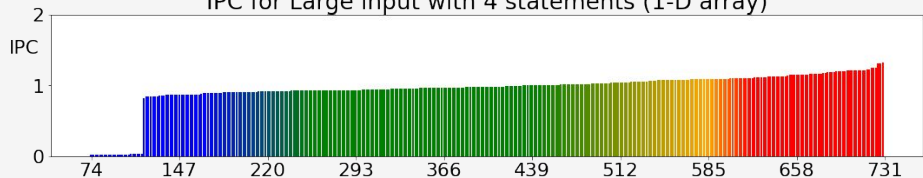


IPC for Extra Large input with 5 statements (1-D array)

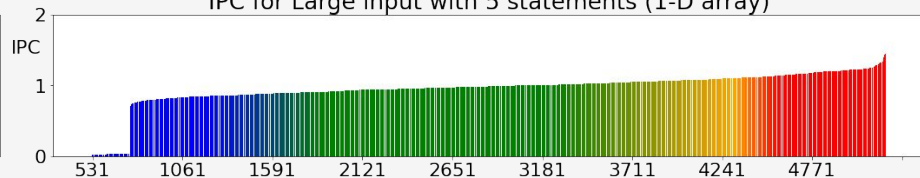


Generated loops: IPC Study on Skylake

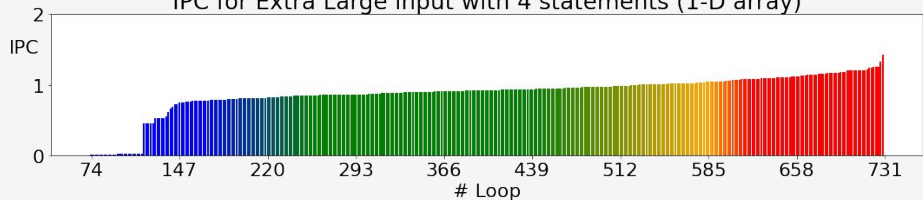
IPC for Large input with 4 statements (1-D array)



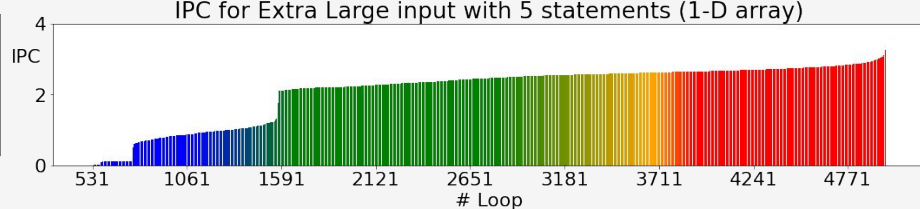
IPC for Large input with 5 statements (1-D array)



IPC for Extra Large input with 4 statements (1-D array)



IPC for Extra Large input with 5 statements (1-D array)



Limitations

1

Modeling only reuse

Currently models only reuse.
Need to include FLOPS.

2

Random Schedules

Based on selecting random schedule

3

Approximate solution

Based on approximate
memory/compute bound heuristics

4

Sequential loops

Generates only sequential loops

Future extensions

1

Model Arithmetic Intensity(AI)

Generate loops by controlling AI/OI

2

Multi-dimensional arrays

Delinearized multi-dimensional accesses

3

Schedule selection and Schedule trees

Schedule selection algorithm and implement structured schedules

4

Using ML Techniques

Generate loop variants

Summary

- Proposed a polyhedral loop generator for Generating Millions of SCoPs (GeMS)
- Generated labeled memory/compute-bound loops
 - ◆ By imposing constraints on cache locality in terms of reuse distances
 - ◆ Multi-dimensional array accesses → special case of 1D accesses using delinearization
- Showed initial evaluation of GeMS on a set of 24K kernels
 - ◆ On two machines with differing cache parameters
 - ◆ Varying numbers of statements, loops/array dimensions
 - ◆ Kernels are diverse in terms of IPC and execution time
 - ◆ No compile-time/runtime failures
- Generated kernels can be used as a labeled dataset for ML based cost models
 - ◆ For exhaustive evaluation of Polyhedral techniques

THANK YOU!

QUESTIONS?