Machine Learning and Compiler Optimization

P. (Saday) Sadayappan

Oct. 28, 2024

Compiler Optimization: Strengths & Limitations

- Compilers very effective in lowering sequential high-level language programs to minimize the number of executed low-level instructions
  - Production compilers like gcc, llvm/clang, Intel icc/icx, IBM xlc can exploit ILP effectively
  - Many mature optimizations: register allocation, common sub-expr elimination…
- However, the dominant cost is data movement and not the arithmetic ops.
  - Data movement can be orders of magnitude more expensive, in terms of energy & time

```
for (i=0; i<n; i++)
  a[i]= s*a[i]+a[i];
```

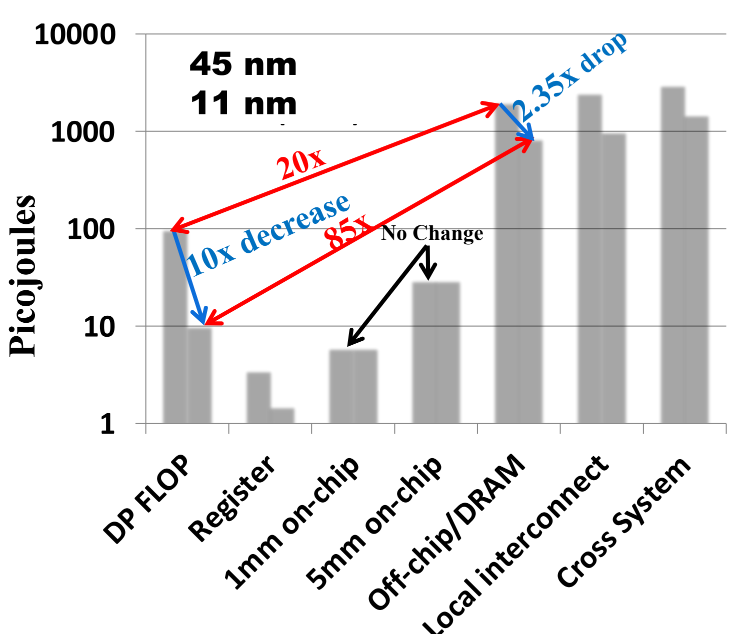
```
.L13:
    vmovaps (%r11,%rax), %ymm0
    addl    $1, %ecx
    vmulps  %ymm2, %ymm0, %ymm1
    vaddps  %ymm1, %ymm0, %ymm0
    vmovaps %ymm0, (%r11,%rax)
    addq    $32, %rax
    cmpl    %ecx, %r9d
    ja      .L13
```

AVX code

Data Movement vs. Computation (Energy and Throughput)

- Energy per FLOP is orders of magnitude lower than data movement
  - Technology scaling lowered energy for arithmetic much more than for data movement
- Peak memory bandwidth is rising slower than peak performance

Data Movement Cost: Energy Trends



Source: Jim Demmel, John Shalf

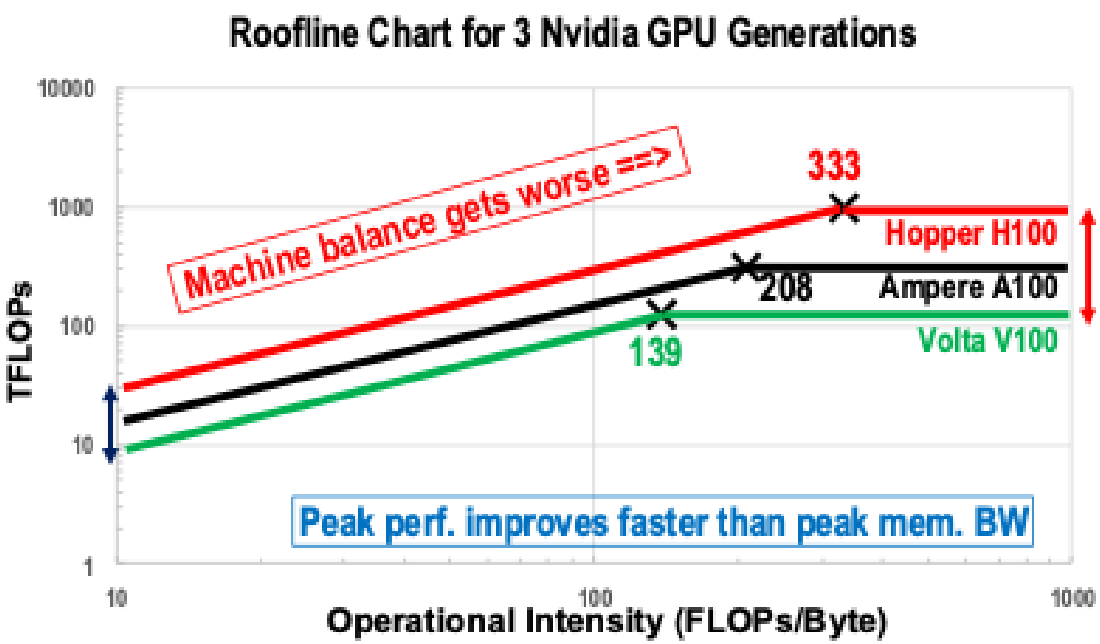
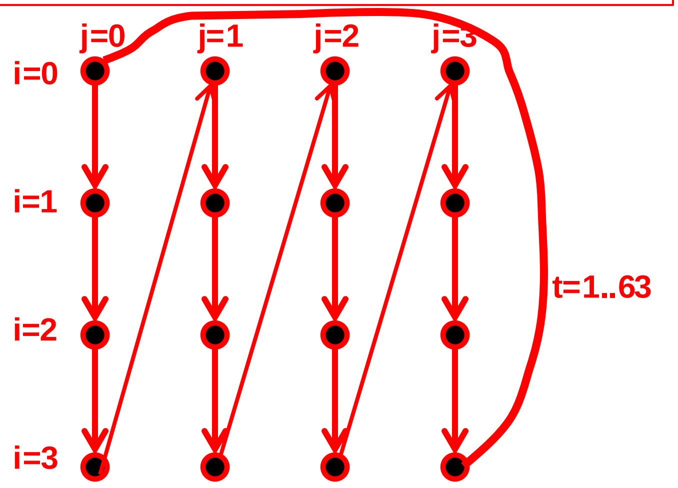


Illustration: Data Movement is Expensive

- The main cause of performance loss is data movement overheads
  - –Between nodes in a multi-node system
  - –Through the memory hierarchy at each node
- Illustrative synthetic example
  - –Functionally identical codes with very different performance on my laptop

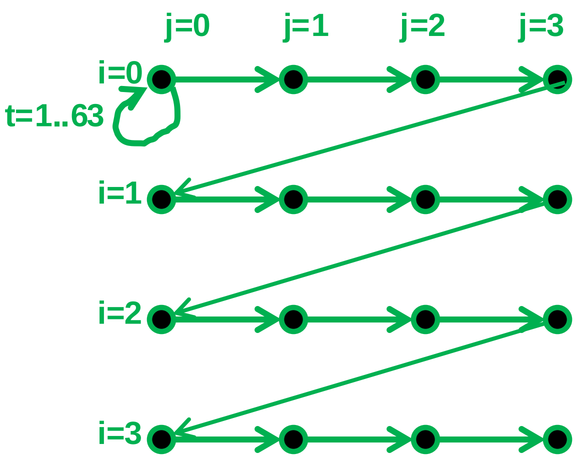
```
double A[4096][4096];
// Initialize A[][] to 0.0
for (t=0; t<64; t++)
 for (j=0; j<4096; j++)
  for (i=0; i<4096; i++)
   A[i][j] += (2*t+j)+(3*t+i);
```

6.4 sec




```
double A[4096][4096];
// Initialize A[][] to 0.0
for (i=0; i<4096; i++)
 for (j=0; j<4096; j++)
  for (t=0; t<64; t++)
   A[i][j] += (2*t+j)+(3*t+i);
```

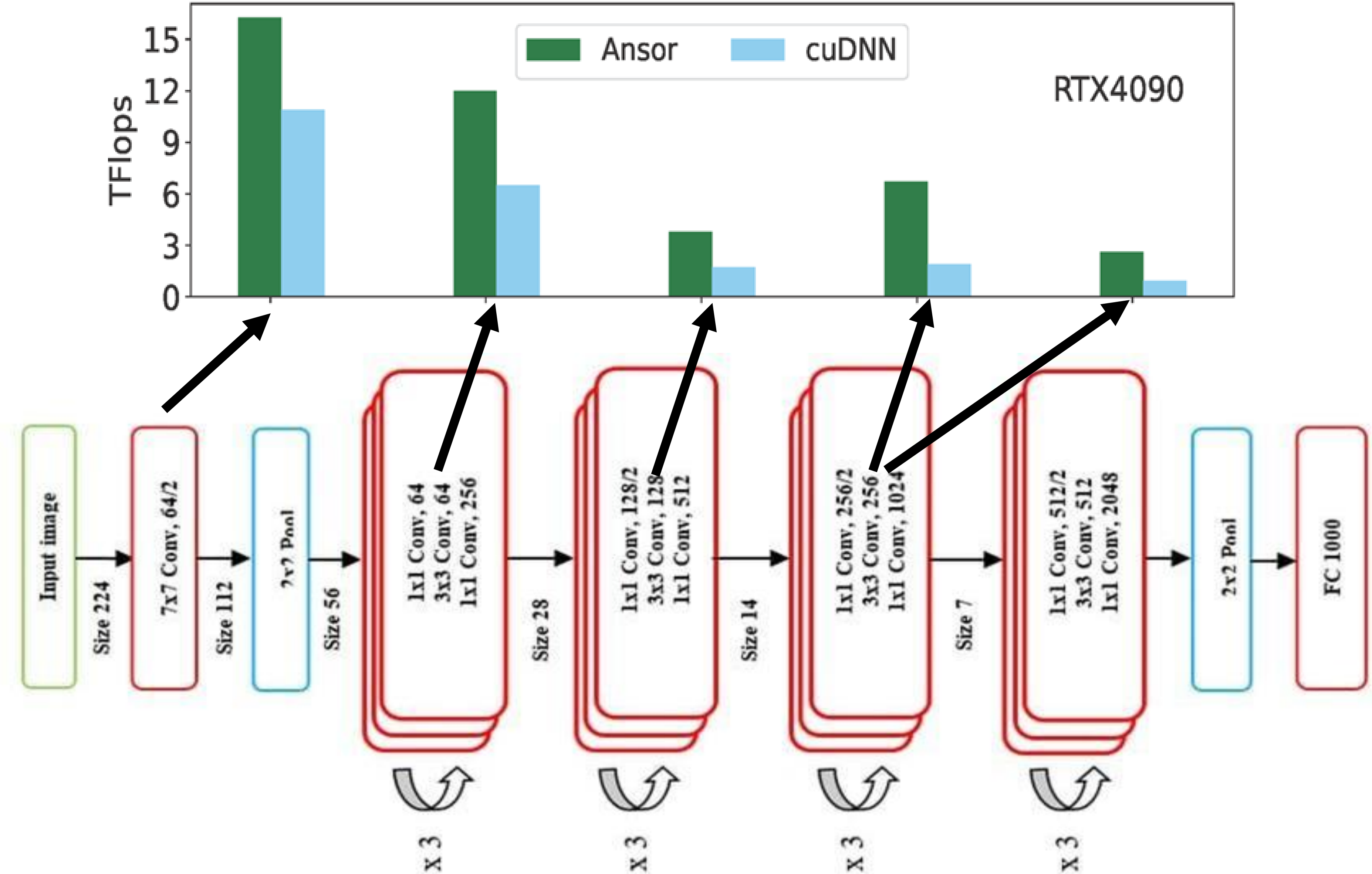
0.26 sec



High-Level Summary

- Many efforts on using ML for Compiler Optimization
  - But few (so far) have addressed problems of significance
    - Examples: Best loop-unroll factor; Device selection (CPU or GPU)
  - Or the comparison baselines have been weak
    - Ideally, performance should match/exceed manual optimization
- Significant SW Problem: Developing high-performance applications for parallel/heterogeneous systems
  - Challenges: Application developer productivity & performance portability
- One impressive ML-based compiler: TVM/Ansor
  - Automated synthesis of high-performance code
    - Multi-core CPUs, GPUs, FPGA
    - Input: High-level "Einsum" tensor expression
    - Performance exceeds highly tuned vendor libraries (MKL, cuDNN)
  - But only fixed operator sizes (OK for ML inference pipelines)
- Can we expand the scope of effective ML-based high-perf. code synthesis beyond what TVM/Ansor can now achieve?

TVM/Ansor Performance: ResNet



TVM/Ansor Auto-Tuning Compiler

(PyTorch, TensorFlow, Keras)	DL models
High Level IR	Graph Level
Tensor Expression	High Level Expression
Auto-tuning Compiler	Kernel Level
GPU, CPU, FPGA	Hardware

```
matmul = te.compute(
        (N, M),
        lambda i, j: te.sum(A[i, k] * B[k, j]),)
```

Generate configuration space

Search for optimal configuration

TVM/Ansor: Code Schema for Nvidia GPU

```
1  // blockIdx.x i.0@j.0@ (None)
2    for i.0 (None)
3      for j.0 (None)
4        //vthread i.1@j.1@ (None)
5        for i.1 (None)
6          for j.1 (None)
7            // threadIdx.x i.2@j.2@ (None)
8            for i.2 (None)
9              for j.2 (None)
10               // thread level code Line 10 - 22
11               for k.0 (None)
12                 // shared memory buffer loading Line 13- 14
13                 B.shared[...] = B[...]
14                 A.shared[...] = A[...]
15                 __syncthreads();
16                 // register level
17                 for k.1 (None) for k.2 (None)
18                   for i.3 (None) for i.4 (None)
19                     for j.3 (None) for j.4 (None)
20                       matmul.local = ...
21               // store output to global memory
22                matmul[...] = matmul.local[...]
```

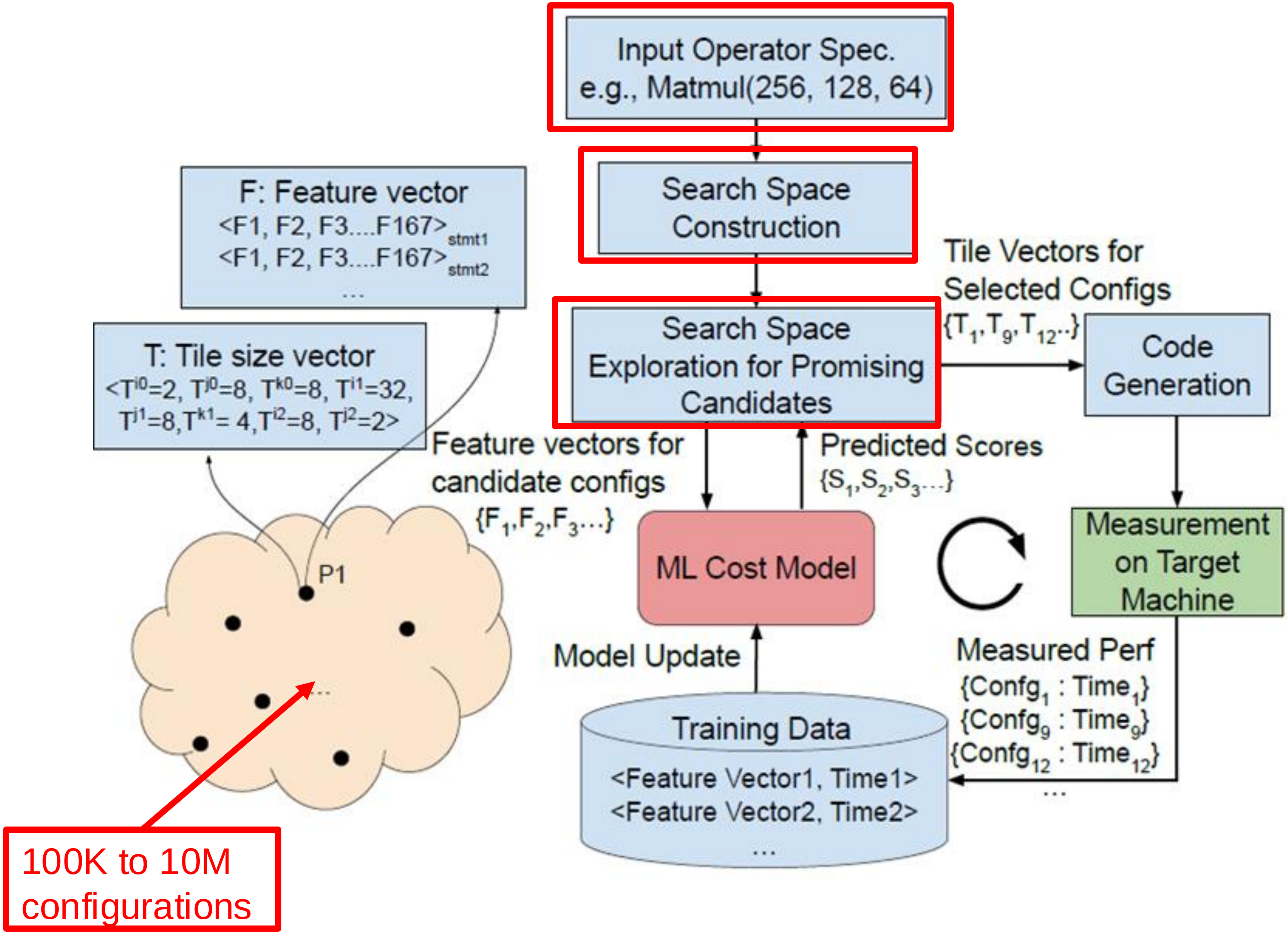
```
matmul = te.compute(
    (N, M),
    lambda i, j: te.sum(A[i, k] * B[k, j]),)
```

Generate configuration space

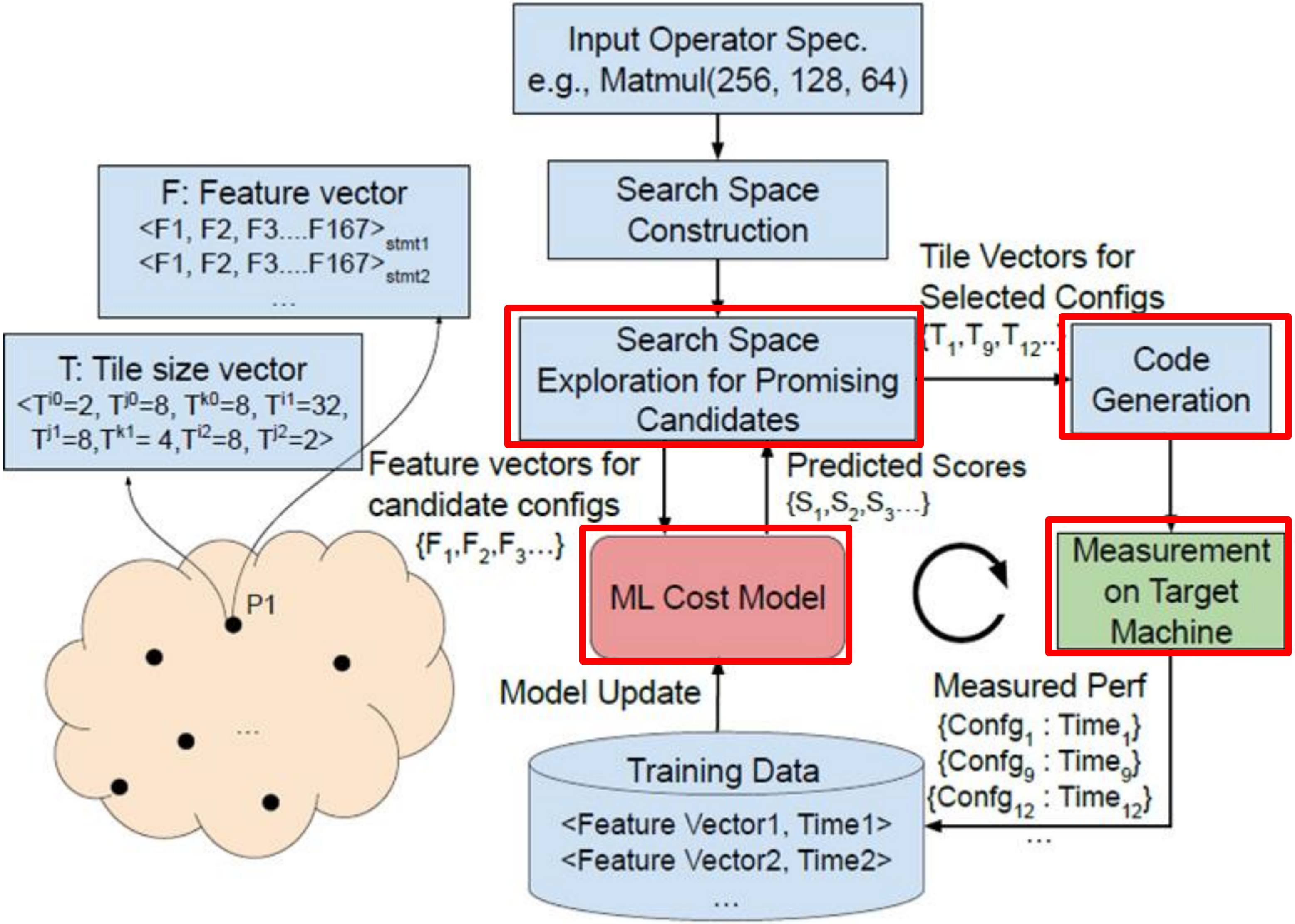
Search for optimal configuration

- Multi-level tiled code
  - 5-level tiling of each ”parallel” loop
  - 3-level tiling of ”reduction” loops
- Configuration: Set of tile sizes
  - <i0,i1,i2,i3,i4,j0,j1,j2,j3,j4,k0,k1,k2>
- XGBoost proxy perf. model
  - Ansor uses a 167-component feature vector for training model
  - <t1,t2,…,tk> => <f1,f2,….,f167>
  - 167 features generated from AST
  - op count, buffer sizes, …

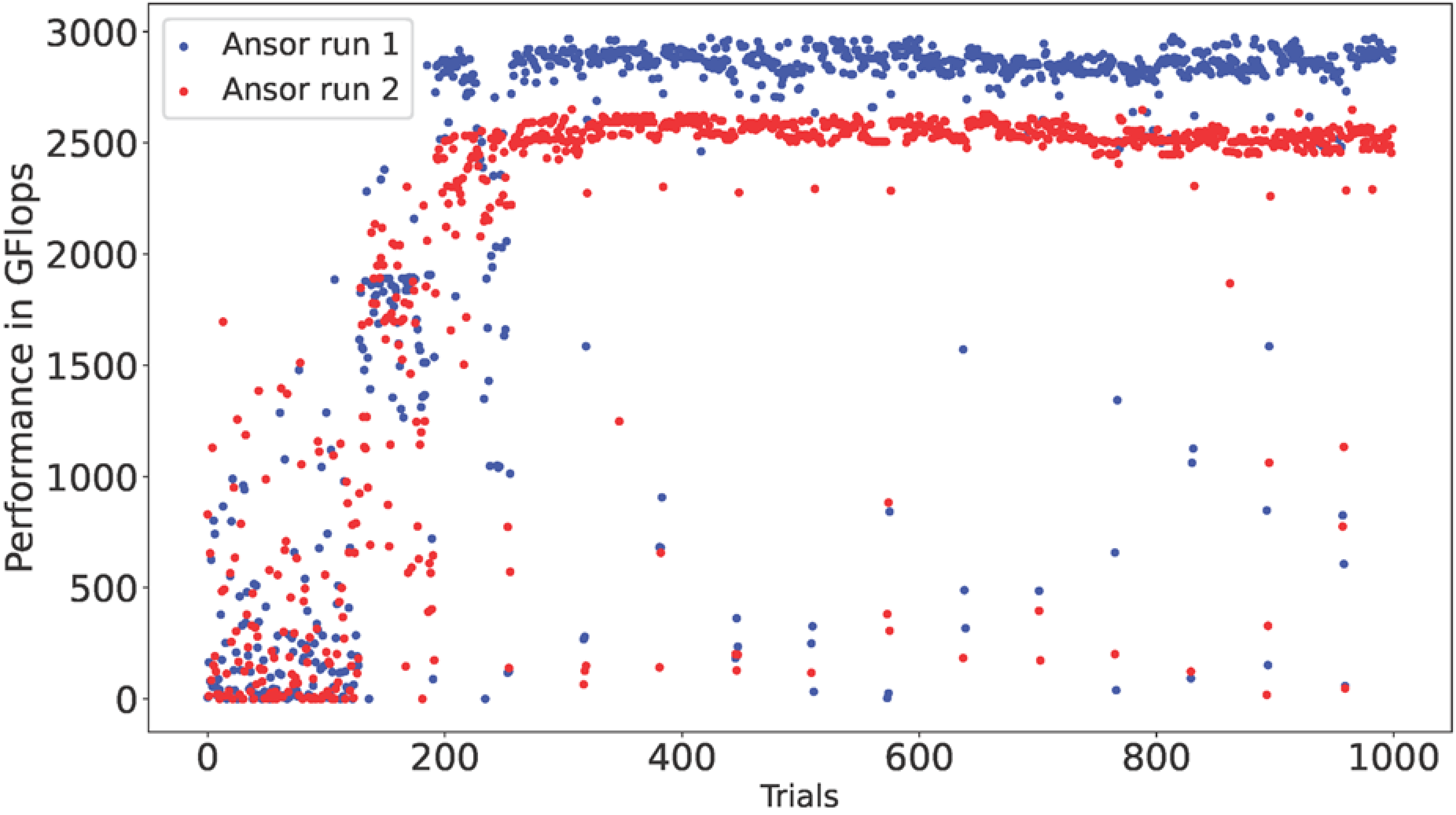
TVM/Ansor ML-Driven Compiler



TVM/Ansor ML-Driven Compiler



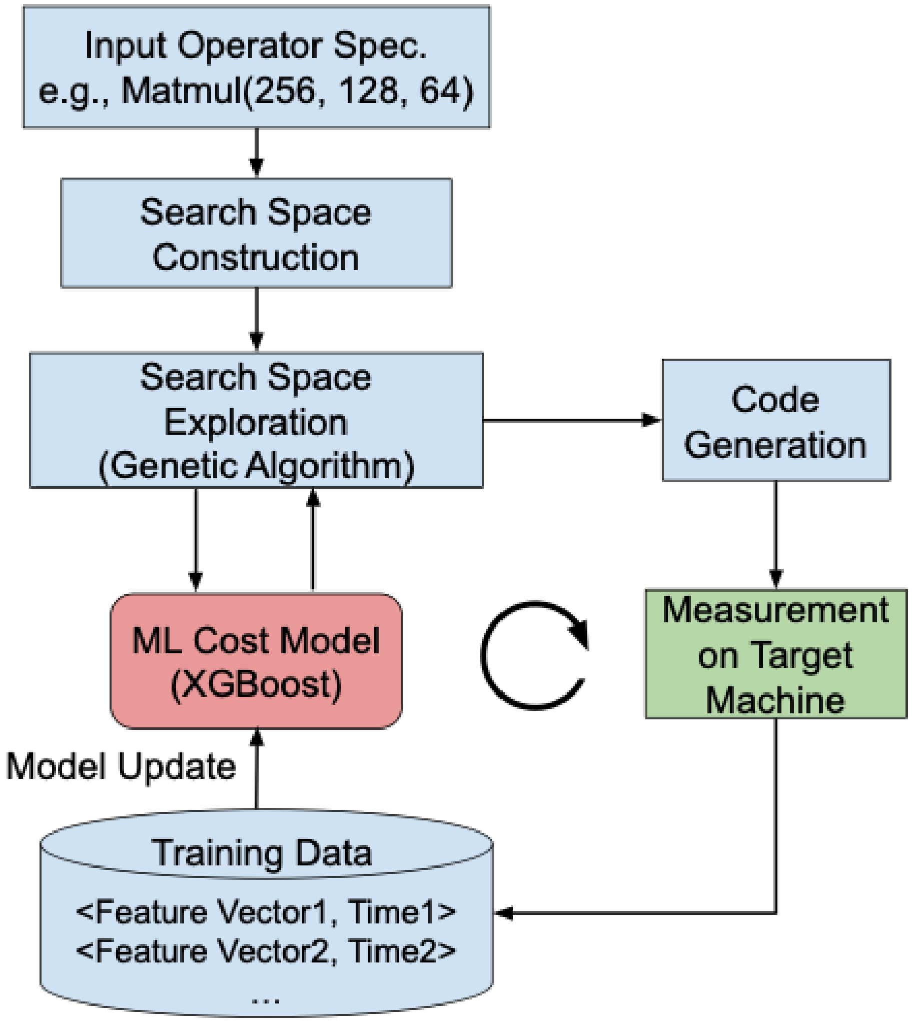
TVM/Ansor Auto-Tuning



- ML perf. model is updated after every batch of 64 configs.
  - Model very effective in filtering out bad configs. After 2-3 batches
- Across-run variability can be as high as 20%

Initial Explorations with TVM/Ansor

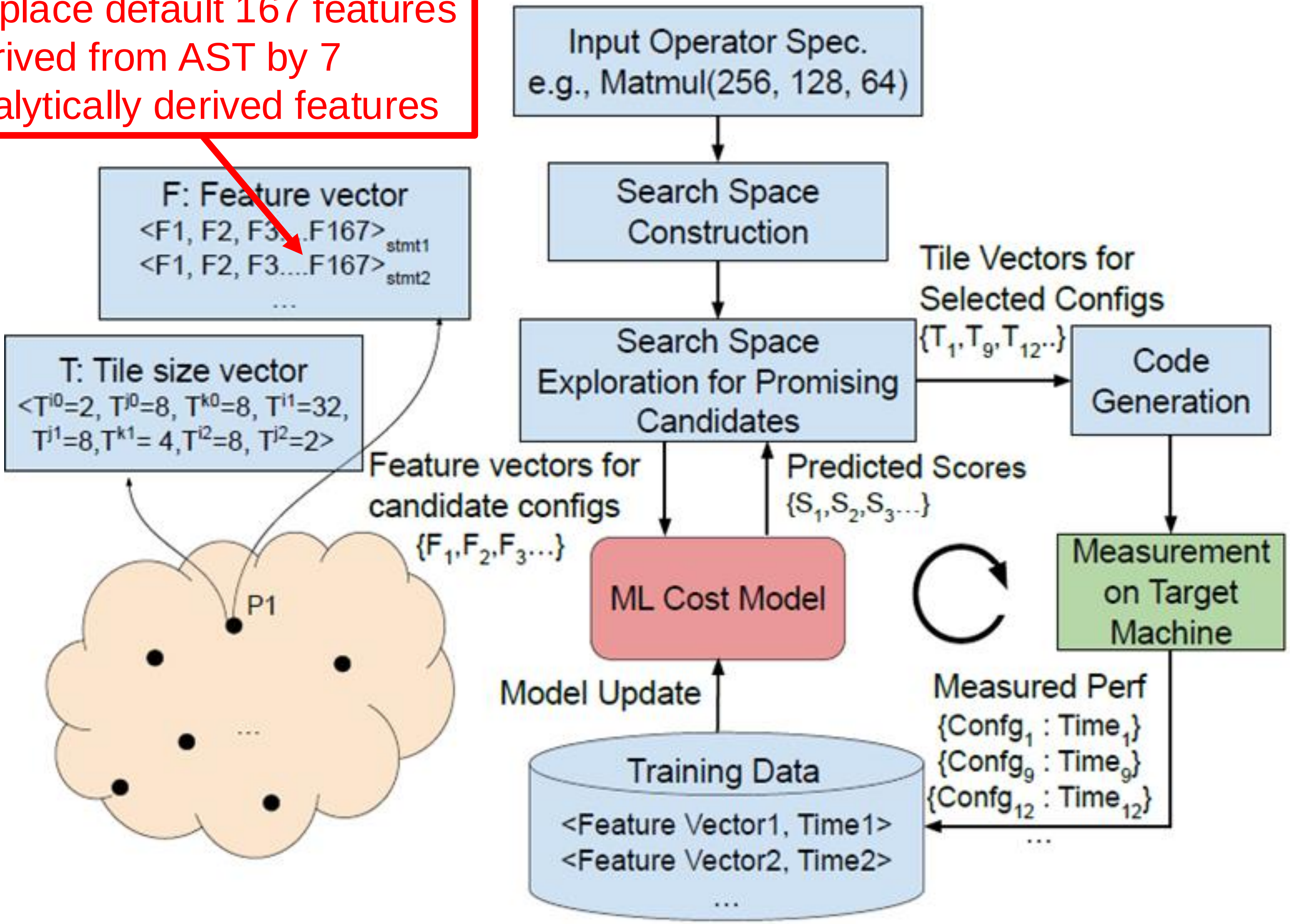
- How important is the accuracy of proxy ML model?
  - Can it be improved by using analytically derived features for concurrency, data movement vol…?
- How important is the search technique
  - Can an alternate search strategy improve auto-tuning time and/or performance?
    - Sampled gradient descent
    - Bayesian optimization



Li, C., Xu, Y., Saravani, S. M., & Sadayappan, P. (2024, May). Accelerated Auto-Tuning of GPU Kernels for Tensor Computations. In *Proc. ACM International Conference on Supercomputing* (ICS 2024).

Exploration 1: Use Analytical Modeling

Replace default 167 features derived from AST by 7 analytically derived features

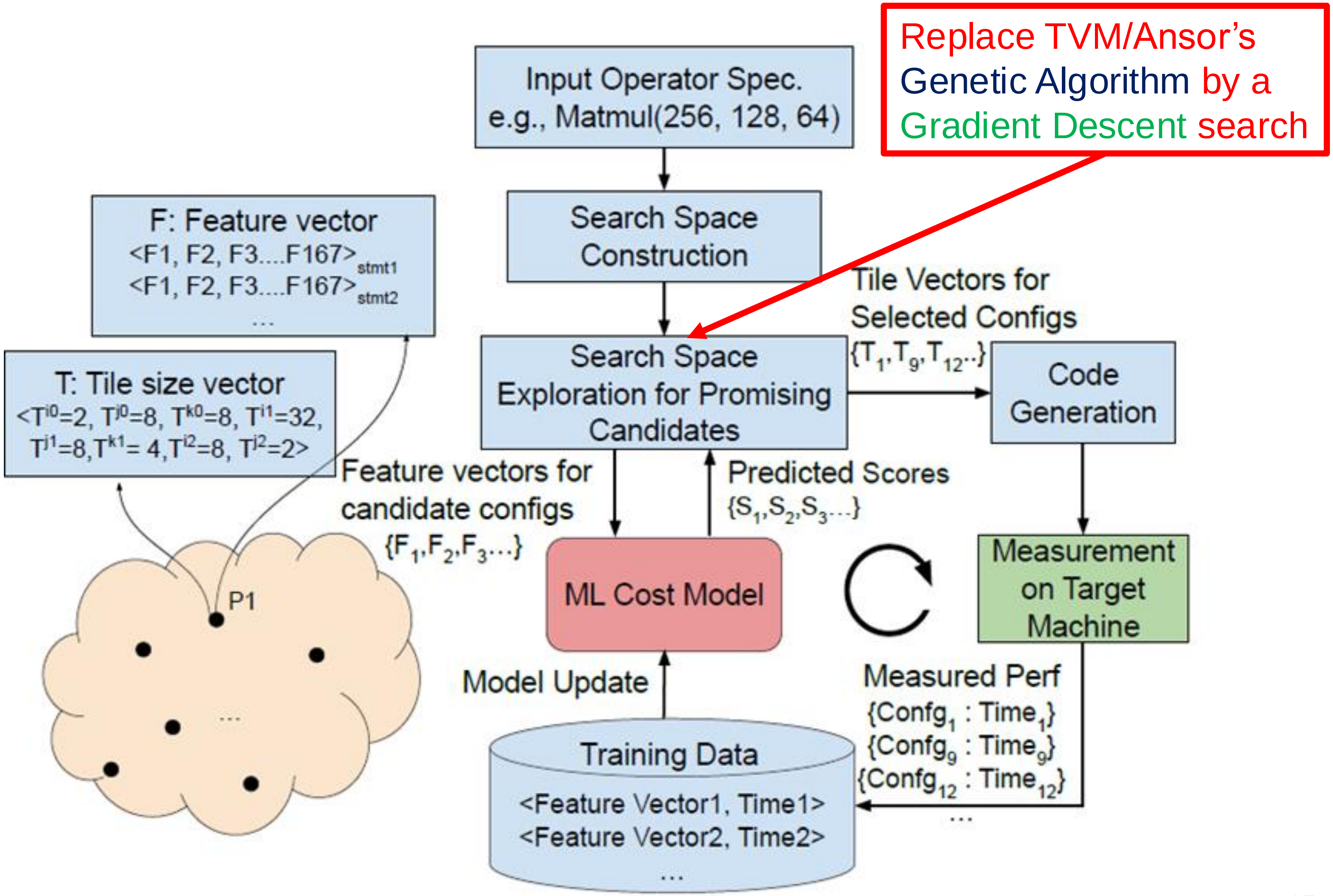


Exploration 1: Use Analytical Modeling

❿ Replace Ansor's 167 metrics from AST with 7 analytically derived metrics from tile sizes

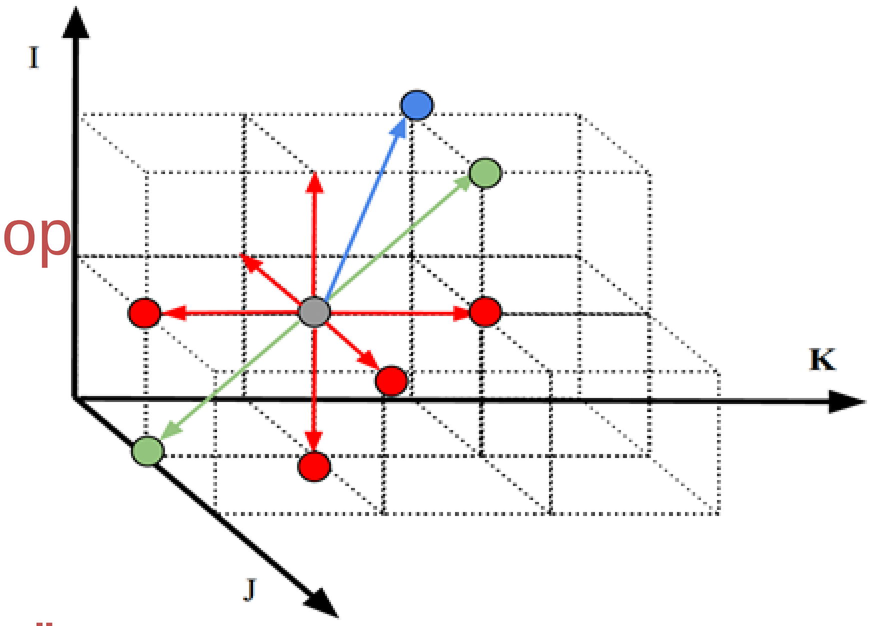
- Data movement metrics:
  1. Operation Intensity w.r.t. global => shared memory
  2. Operation Intensity w.r.t. shared memory => registers
  3. Operational Intensity w.r.t. registers => global memory
- Concurrency metrics:
  4. Instruction level parallelism (ILP)
  5. Warp level parallelism (WLP)
  6. Estimated occupancy
- Load balance metric:
  7. Wave efficiency

Exploration 2: Use Gradient-Descent Search



Exploration 2: Gradient Descent Search

- Replace Genetic Algorithm search by Gradient Descent search
  - Start at a random configuration
  - Use XGB proxy model to predict performance of all 1-hop and 2-hop neighbors in D-dimensional tile-space
  - Evaluate the top-k (k=2 used) by compile/execute/measure
  - If better, move; else "slide window" and evaluate the next-k
  - Abort thresholds to terminate search path and start at new random start



Experimental Evaluation

- Compared TVM/Ansor with Ansor-AF-DS
  - AF: Exploration 1 (Analytical Features for XGBoost model)
  - DS: Exploration 2 (Gradient-Descent Dynamic Search)
- Three runs for each benchmark on two platforms
  - Nvidia RTX 3090 and RTX 4090
  - Mean and variability computed
- Best achieved performance measured after:
  - 1 minute wall-time for auto-tuning
  - 2 minutes wal-ltime for auto-tuning
  - 1000 total trials for auto-tuning

Experimental Evaluation: Benchmarks

Matrix Multiplication
(Bert Base/Large)

Layer	M	N	K
M0	512	64	1024
M1	512	4096	1024
M2	512	64	768
M3	512	3072	768
M4	512	1024	4096
M5	512	768	3072

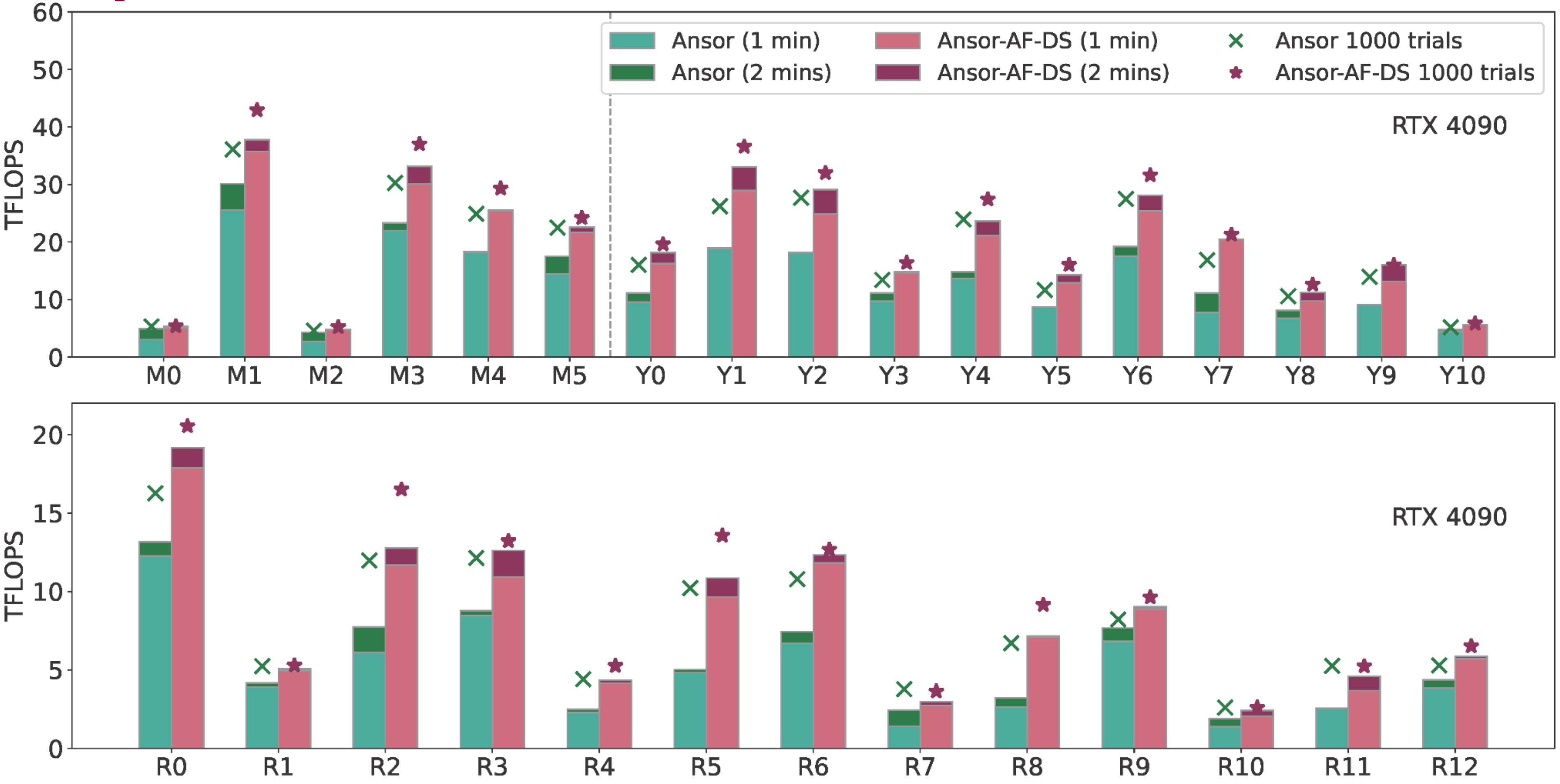
Layer	F	C	H/W	R/S
Y0	32	3	544	3
Y1	64	32	272	3
Y2	128	64	136	3
Y3	64	128	136	1
Y4	256	128	68	3
Y5	128	256	68	1
Y6	512	256	68	3
Y7	512	256	34	3
Y8	256	512	34	1
Y9	1024	512	17	3
Y10	512	1024	17	1

Yolo-9000 conv2d

ResNet-50 conv2d

Layer	F	C	H/W	R/S
R0*	64	3	224	7
R1	64	64	56	1
R2	64	64	56	3
R3	256	64	56	1
R4*	128	256	56	1
R5	128	128	28	3
R6	512	128	28	1
R7*	256	512	28	1
R8	256	256	14	3
R9	1024	256	14	1
R10*	512	1024	14	1
R11	512	512	7	3
R12	2048	512	7	1

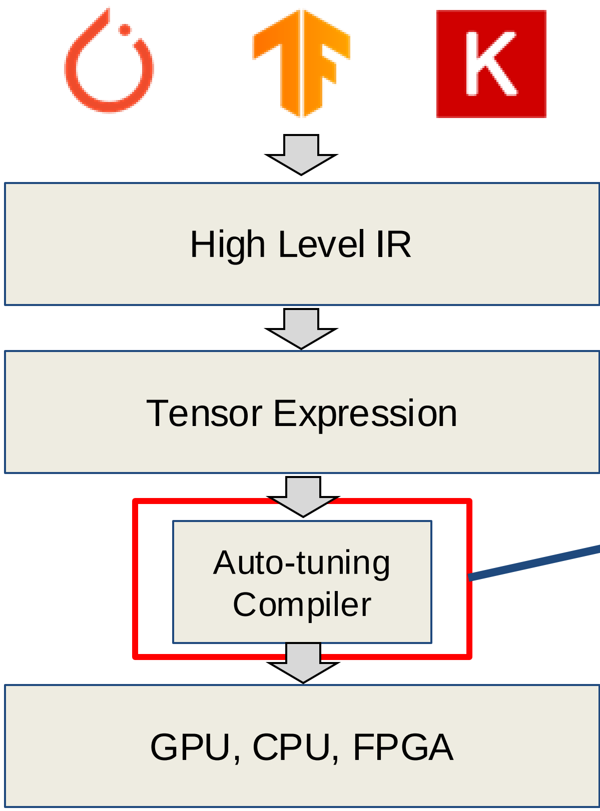
Experimental Evaluation: Kernel Performance



- Ansor-AF-DS: Comparable (within 5%) or better perf. in 2 minutes (dark maroon) than Ansor 1000 trials (green cross)
- For more than half the benchmarks, Ansor-AF-DS after 1 minute (light maroon bar) is better than Ansor-1000-trials.

Some Open Problems

TVM can only synthesize code for fixed constant values of N,M,K.
How to extend for parametric N,M,K?




```
matmul = te.compute(
        (N, M),
        lambda i, j: te.sum(A[i, k] * B[k, j]),)
```

Generate configuration space

Search for optimal configuration

TVM does not learn and improve from previous tuning runs.
How to progressively refine code schema or search strategy across runs?

Summary

- Significant interest in ML for compiler optimization
  - But most efforts are not (yet) targeting high-impact scenarios
- TVM/Ansor's ML-based framework synthesizes parallel high-performance codes for GPUs and CPUs
  - Higher performance than vendor libraries
  - But currently limited to fixed sized tensor operators
- Can the scope/impact of ML for compiler optimization be enlarged via powerful ML models/methodologies?
  - Reinforcement Learning?
  - Bayesian Optimization?
  - LLMs?