tinyML® Talks

Enabling Ultra-low Power Machine Learning at the Edge

"Tutorial on micro-kernel based hardware acceleration"

Manu Rastogi

[Bay Area Group] - August 13, 2020

www.tinyML.org

Bay Area/Silicon Valley tinyML Meetup

Organizing Committee

Manu Rastogi

Amey Naik

Mahmut Sinangil

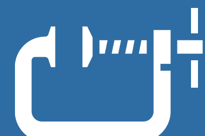
Yu Pu

If you are interested in volunteering, email meetups@tinyML.org

tinyML Talks Sponsors

Additional Sponsorships available – contact Bette@tinyML.org for info

WE USE AI TO MAKE OTHER AI FASTER, SMALLER AND MORE POWER EFFICIENT



Automatically compress SOTA models like MobileNet to <200KB with **little to no drop in accuracy** for inference on resource-limited MCUs

Reduce model optimization trial & error from weeks to days using Deeplite's **design space exploration**

Deploy more models to your device without sacrificing performance or battery life with our **easy-to-use software**

VISIT **BIT.LY/DEEPLITE** FOR MORE INFO

TinyML for all developers

C++ library

Arduino library

WebAssembly

Dataset

Acquire valuable training data securely

Enrich data and train ML algorithms

Edge Device

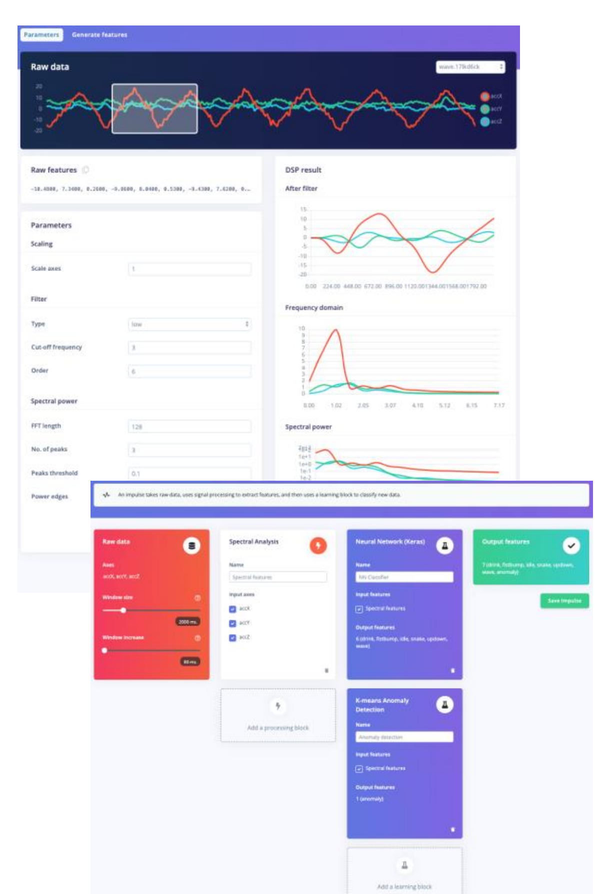
Real sensors in real time
Open source SDK

Impulse

Embedded and edge compute deployment options

Test impulse with real-time device data flows

Test



Copyright © EdgeImpulse Inc.

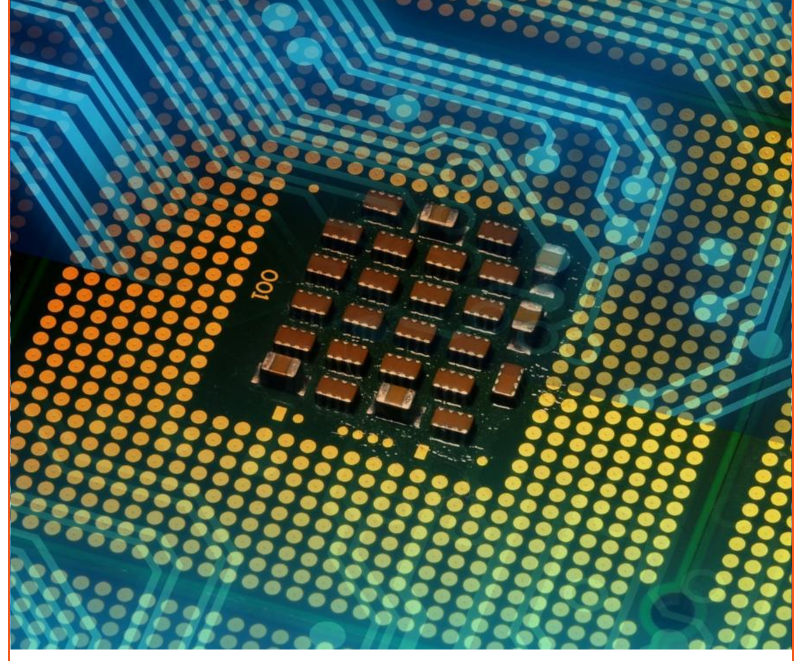
Get your free account at http://edgeimpulse.com

Maxim Integrated: Enabling Edge Intelligence

Sensors and Signal Conditioning

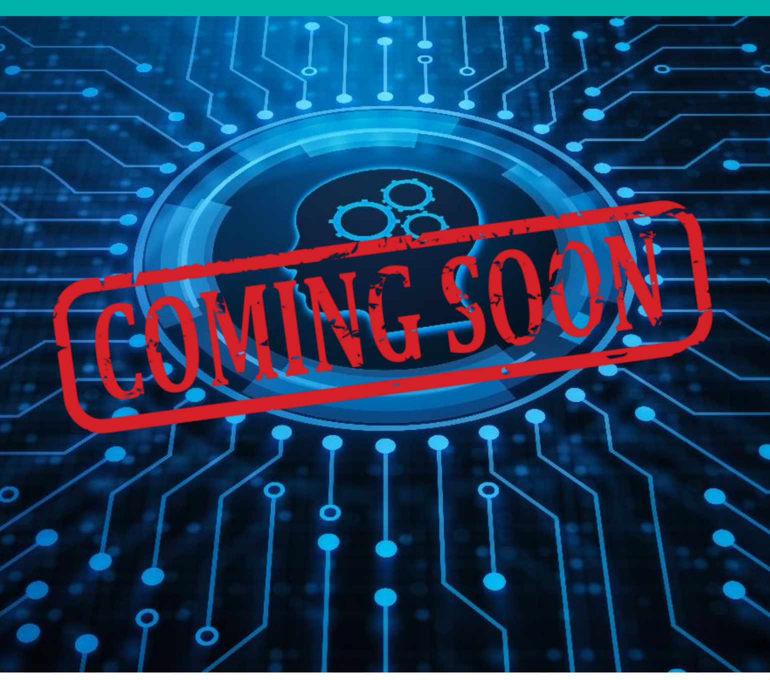
Health sensors measure PPG and ECG signals critical to understanding vital signs. Signal chain products enable measuring even the most sensitive signals.

Low Power Cortex M4 Micros



The biggest (3MB flash and 1MB SRAM) and the smallest (256KB flash and 96KB SRAM) Cortex M4 microcontrollers enable algorithms and neural networks to run at wearable power levels

Advanced AI Acceleration



AI inferences at a cost and power point that makes sense for the edge. Computation capability to give vision to the IoT, without the power cables. *Coming soon!*

Qeexo AutoML for Embedded AI

Automated Machine Learning Platform that builds tinyML solutions for the Edge using sensor data

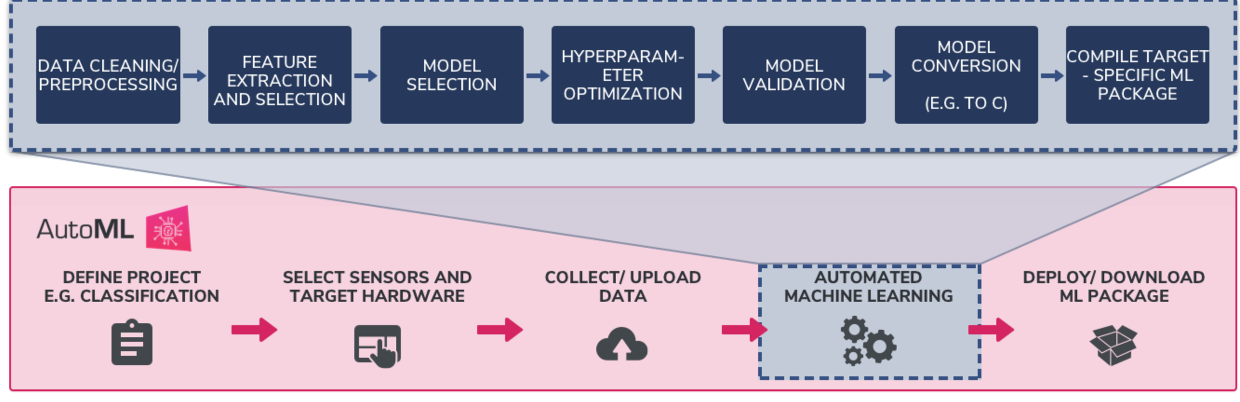
Key Features

- Wide range of ML methods: GBM, XGBoost, Random Forest, Logistic Regression, Decision Tree, SVM, CNN, RNN, CRNN, ANN, Local Outlier Factor, and Isolation Forest
- Easy-to-use interface for labeling, recording, validating, and visualizing time-series sensor data
- On-device inference optimized for low latency, low power consumption, and a small memory footprint
- Supports Arm® Cortex™- M0 to M4 class MCUs
- Automates complex and labor-intensive processes of a typical ML workflow – no coding or ML expertise required!

Target Markets/Applications

- Industrial Predictive Maintenance
- Smart Home
- Wearables
- Automotive
- Mobile
- IoT

QEEXO AUTOML: END-TO-END MACHINE LEARNING PLATFORM

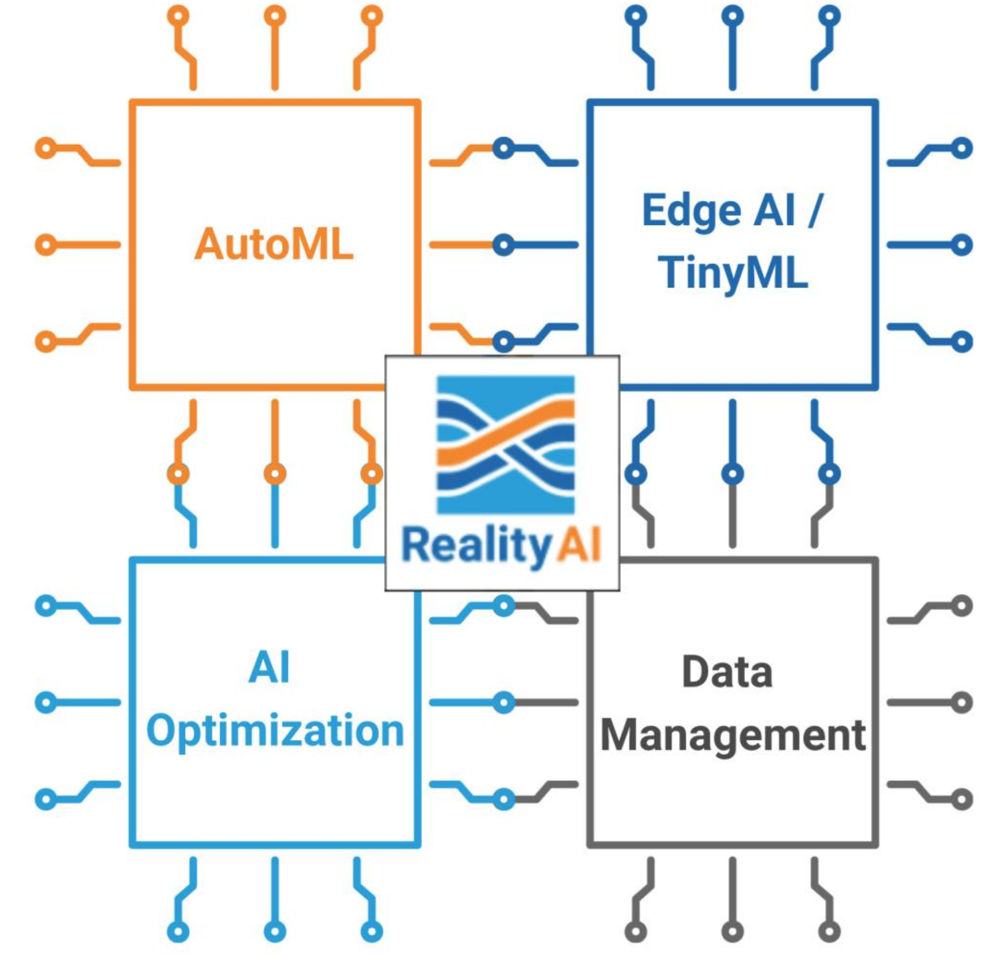


For a limited time, sign up to use Qeexo AutoML at automl.qeexo.com for FREE to bring intelligence to your devices!

Next-Generation AI Tools for Product Development

- Extensive, highly-optimized feature spaces

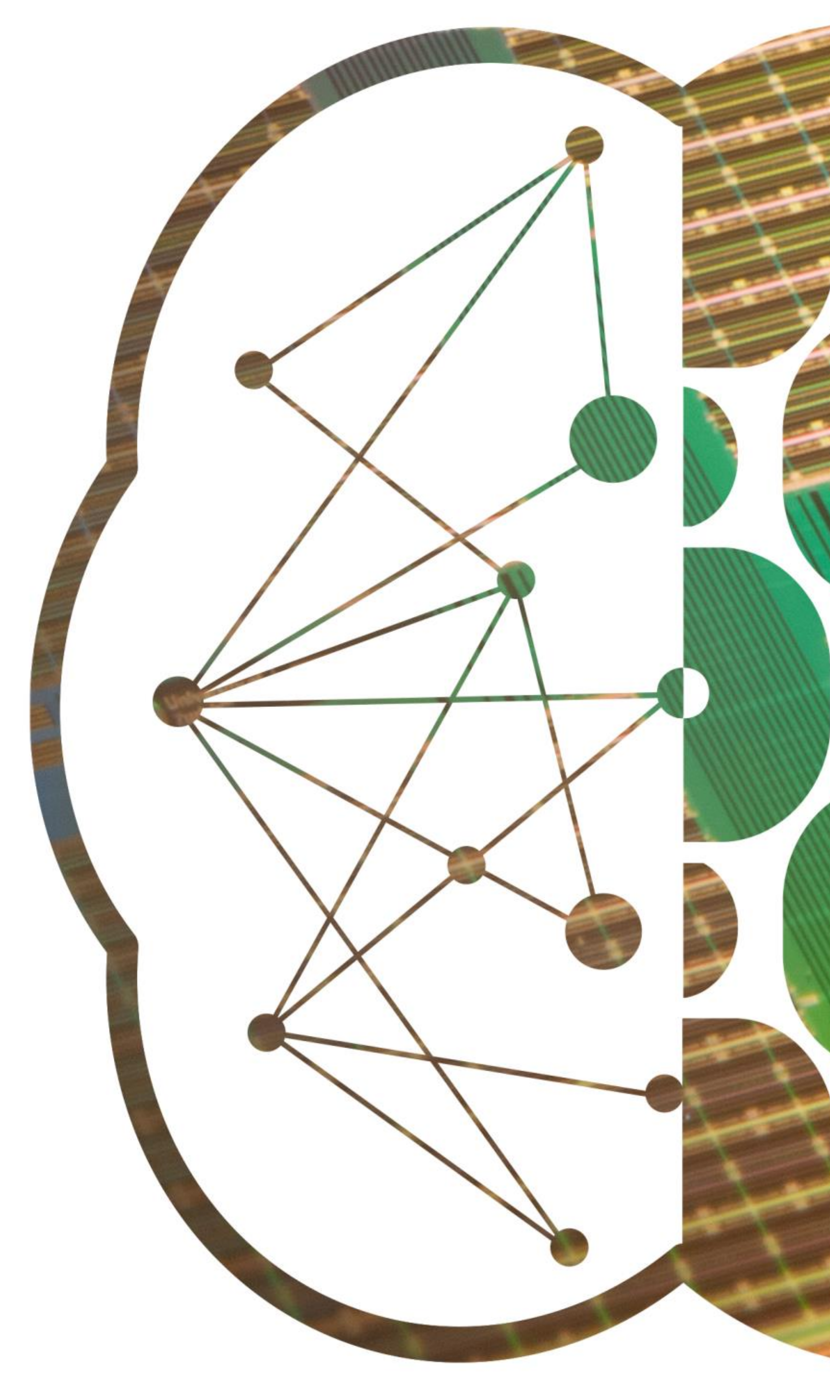
- Super-compact code for MCUs and Gateways
- Sensor selection and placement analysis
- AI-driven component specs
- Automated data quality checks
- Data collection, augmentation & labeling services
- No open source - clean licensing



Get started w/ a special tinyML Talks offer for corporate customers: https://reality.ai/get-started

SynSense (formerly known as aiCTX) builds **ultra-low-power** (sub-mW) **sensing and inference** hardware for **embedded, mobile and edge** devices. We design systems for **real-time always-on smart sensing**, for audio, vision, bio-signals and more.

https://SynSense.ai



Next tinyML Talks

Date	Presenter	Topic / Title
Tuesday August 18	**Mark Stubbs** CTO and Co-Founder, Shoreline IoT Inc. **Urmish Thakker** Senior Research Engineer, Arm ML Research Group	Practical application of tinyML in battery powered anomaly sensors for predictive maintenance of industrial assets Pushing the limits of RNN Compression using Kronecker Products

Webcast start time is 8 am Pacific time

Each presentation is approximately 30 minutes in length

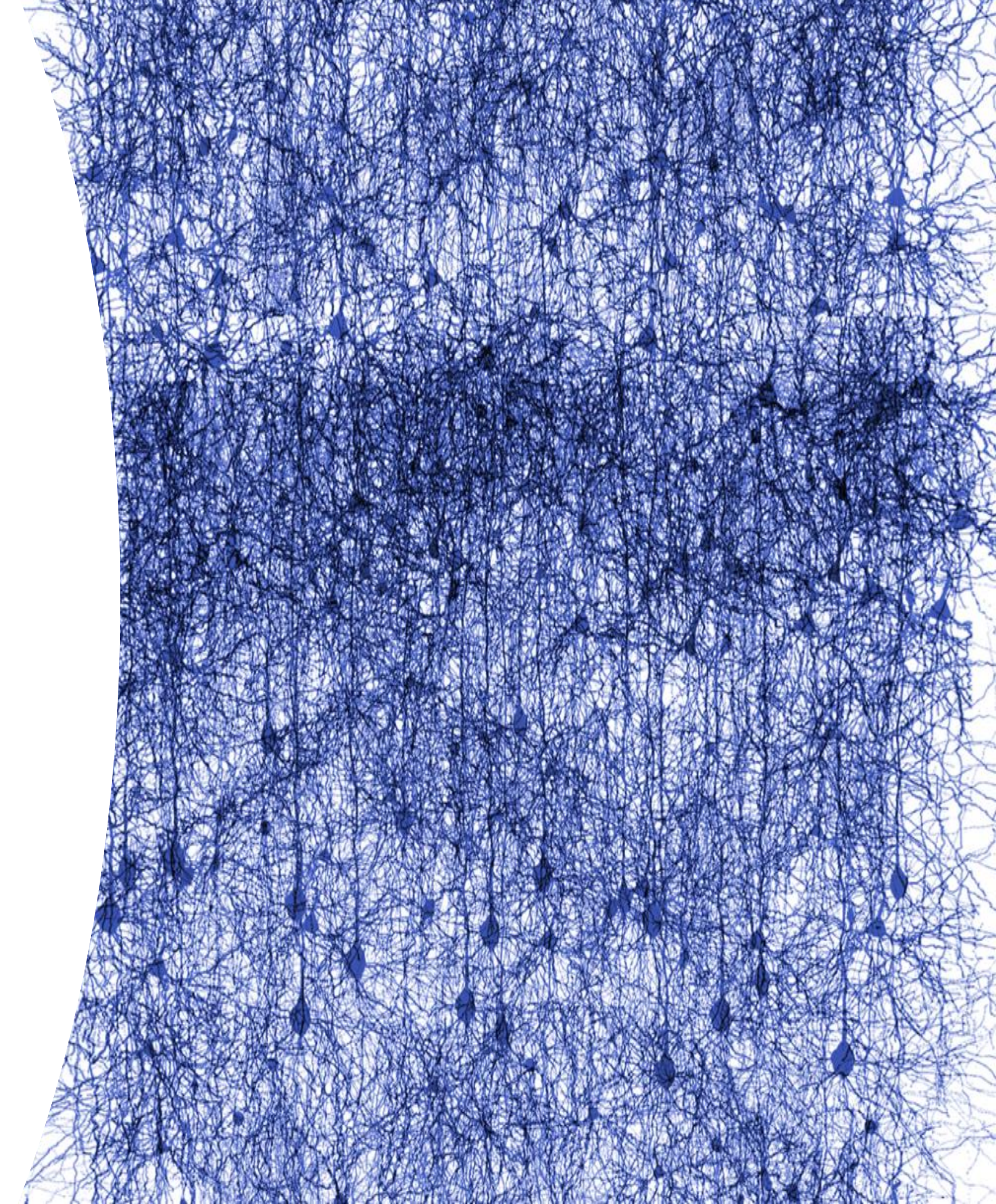
Please contact talks@tinyml.org if you are interested in presenting

Manu Rastogi

Manu Rastogi received his B.Tech from India and his MS and Ph.D. from the University of Florida in 2012. Since graduation, he has worked at Qualcomm Research and HP Labs. As a member of the Qualcomm research team, he worked on the Qualcomm Zeroth processor in various capacities and later on the Qualcomm deep learning engine. His roles at Qualcomm varied from developing signal processing algorithms, model development, and deep learning model optimizations. At HP he led the efforts around machine learning at the edge and self-supervised learning methods using mutual information for speaker identification.

Tutorial: Micro-kernels for hardware acceleration

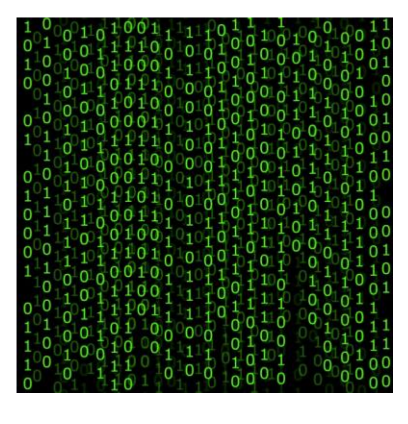
Manu Rastogi



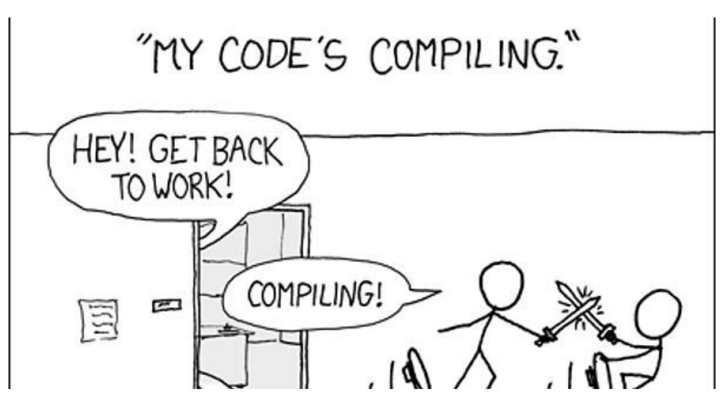
Outline

Motivation

CNN on a simple processor



Matrix Multiply on a simple processor



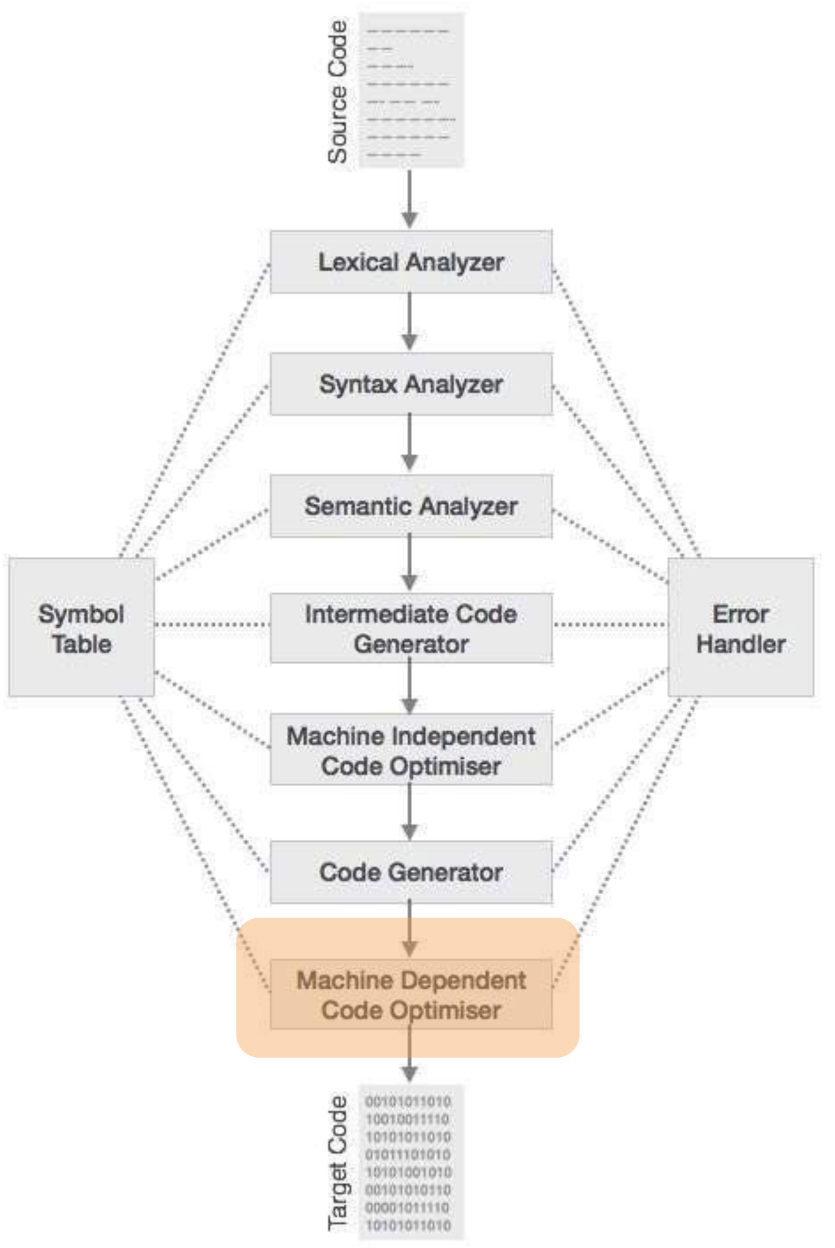
ML based approaches to compilers

Loop transformations

Discussion

Motivation

- What is a micro kernel?
  - Very simply is the assembly code that runs on the processor.
- Why should anyone care?
  - Micro-kernel or micro-code is responsible for orchestrating computation and data movements.
  - Performance critical applications are still tuned, tweaked or written manually in micro-code.
  - A good understanding of micro-code basics can help choose the right hardware product and/or create more efficient algorithms.



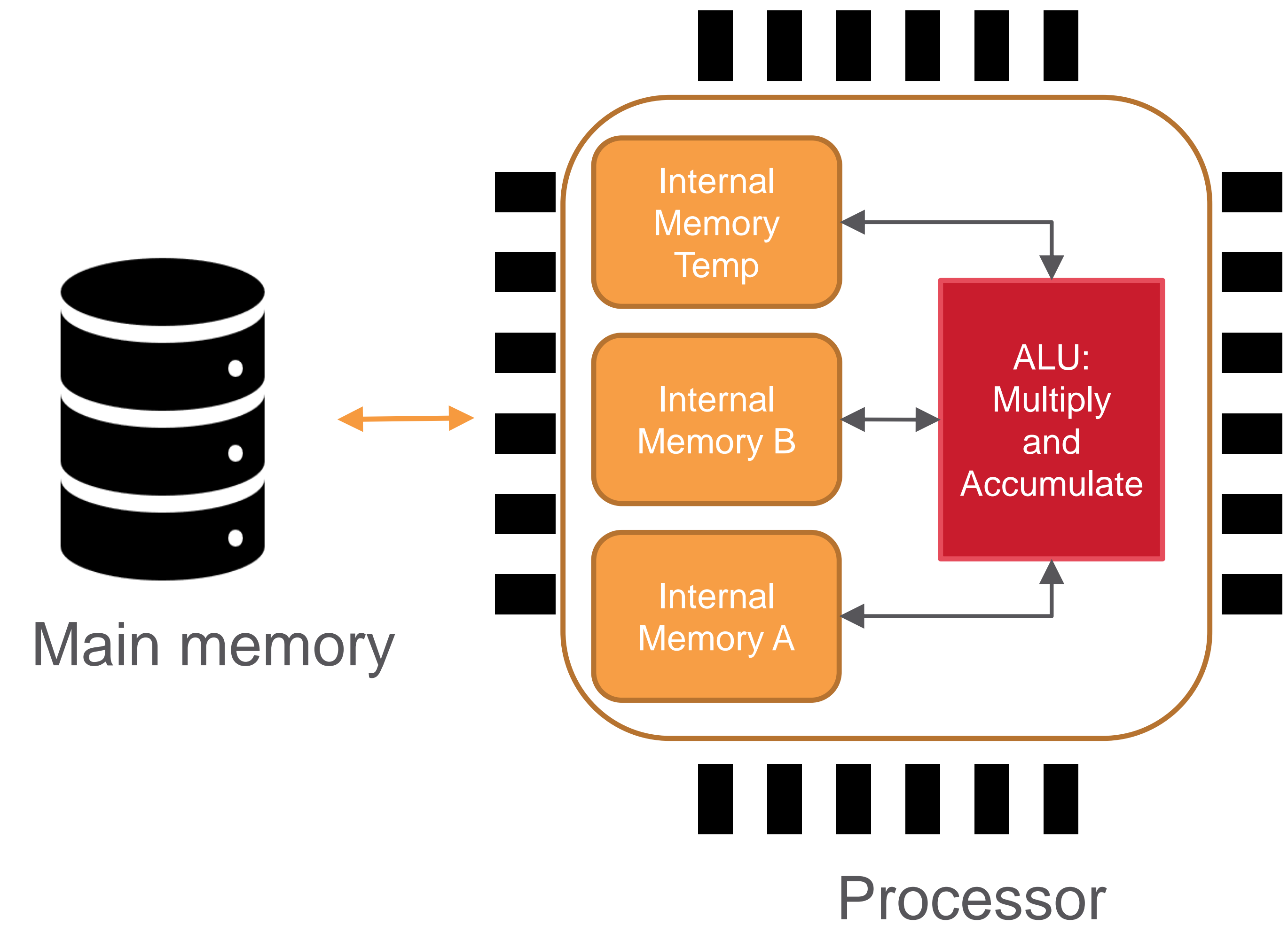
A high-level view of the compilation process.

Source: https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm

A Simple Processor

- A simple processor:
  - Three on-chip memories
  - A main memory
  - A MAC unit

- Some costs:
  - Can be Energy, Power, Delay or EDP etc.
  - *C_mem :* Cost of moving 1 number (16 bits) from main memory to Processor
  - *C_mac :* Cost of multiply and add (or any arithmetic operation).
  - *C_int:* Cost of moving data between ALU and internal memory (or for any data movement in the processor)

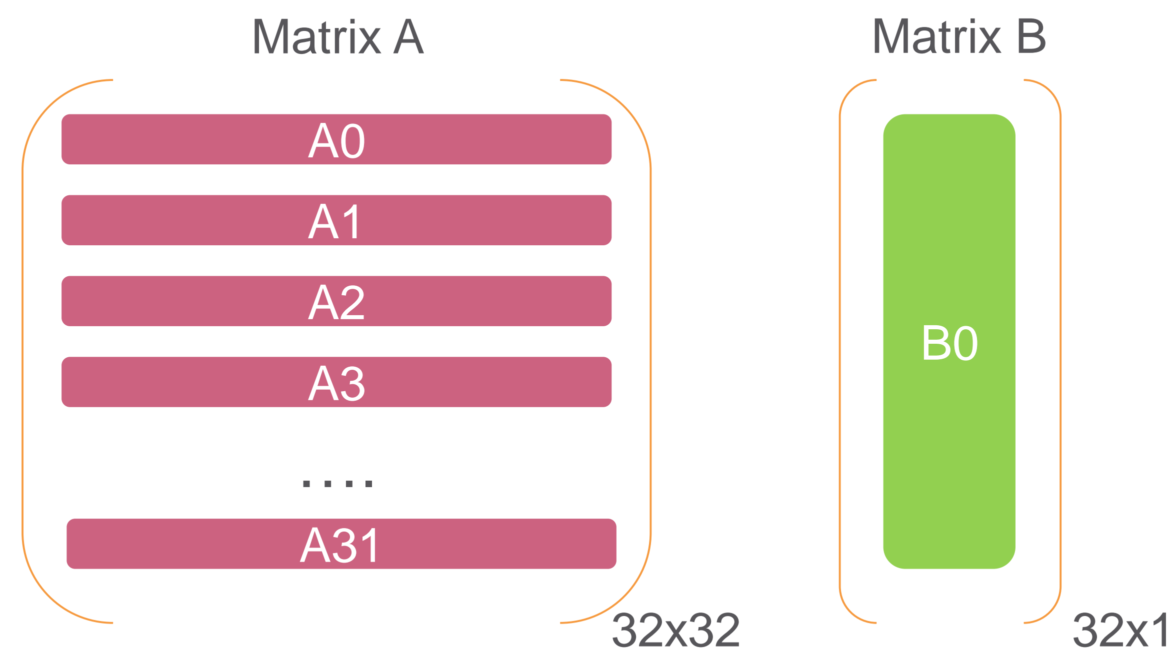


Matrix Multiply

- Multiplying two matrices:

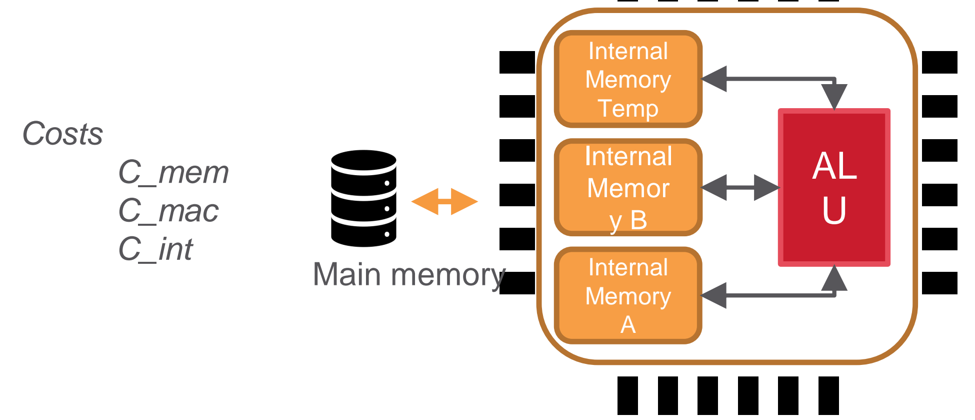
- A of size [32x32]
- B of size [32x1]

- Total MACs: 32x32

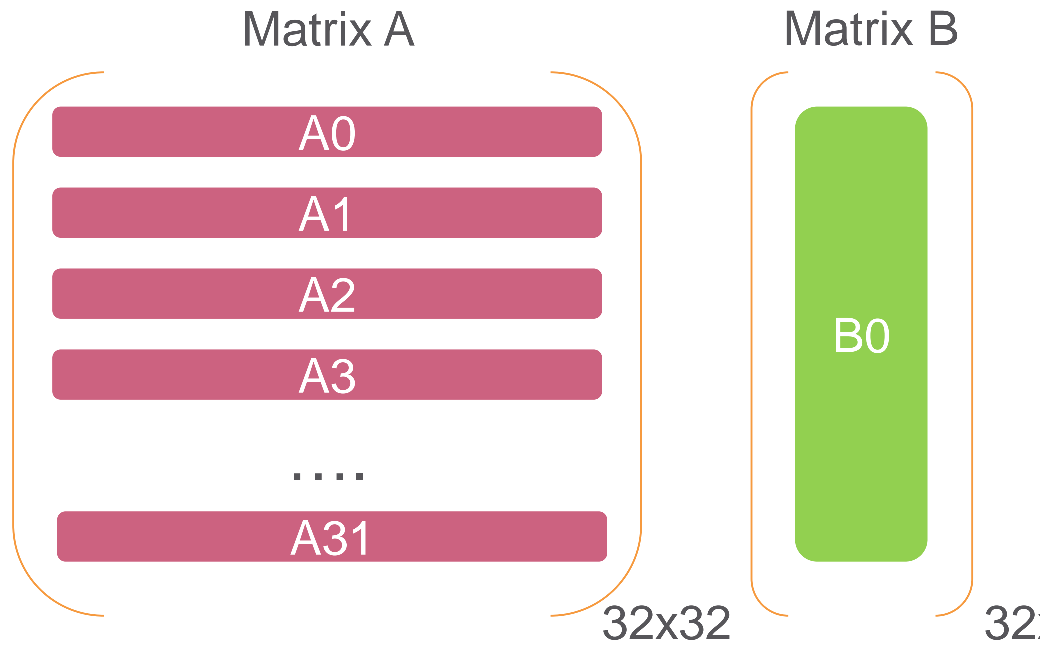


Matrix Multiply on our Processor

- Internal Memories are of size:
  - Mem A : [32x1] Mem B : [32x1] Temp [32x1]



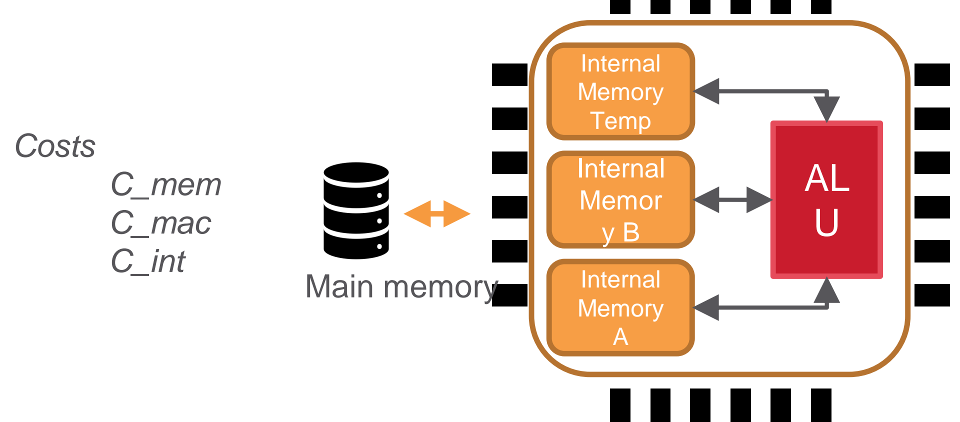
- Steps are:
  - Transfer row of MatA and MatB to memA, memB
  - Multiply and accumulate: A0 x B0
  - Store result in temp memory
  - Repeat for A1,…A31
  - Store back to main memory

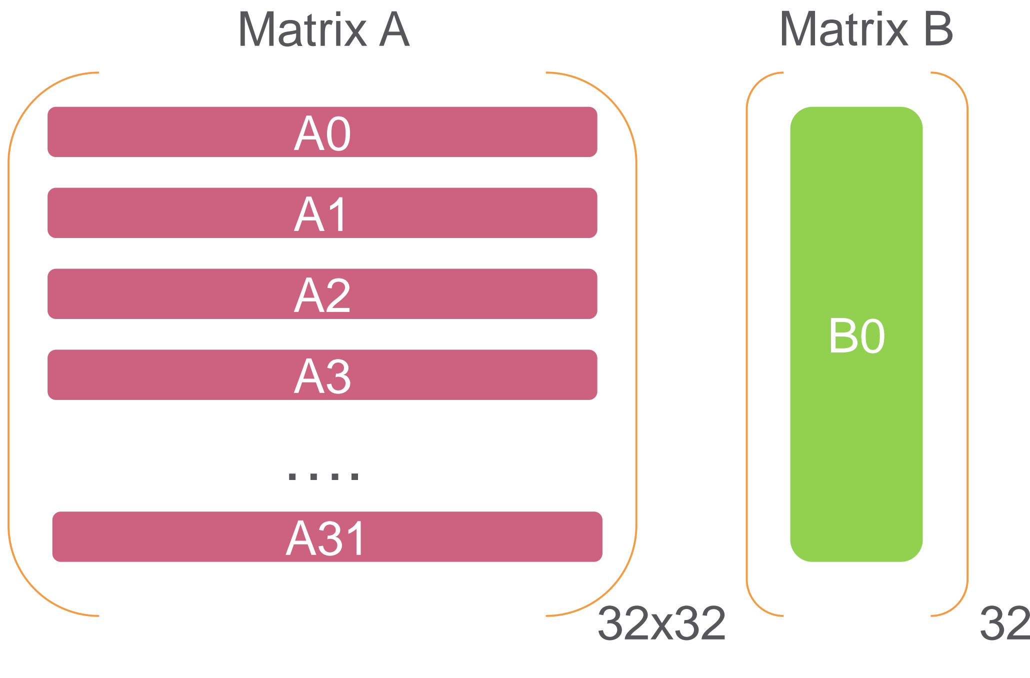


Matrix Multiply on our Processor

- Internal Memories are of size:
  - Mem A : [32x1] Mem B : [32x1] Temp [32x1]
- Cost:
  - Transfer : Cost of 1 row
    - Single row of MatA is 32*Cmem + One time transfer of MatB 32*Cmem
  - Compute cost of a single row:
    - 32*Cmac
  - Store in memory 1 result:
    - 1*Cmem
  - Total cost:
    - Transfer: 32 rows of A * 32 nums/row + 32 rows of B * 1 num/row
    - Compute: 32 rows of A * 32 MAC/row
    - Write Back result: 32 nums
    - 32*32*Cmem+32*Cmem + 32*32*Cmac + 32*Cmem

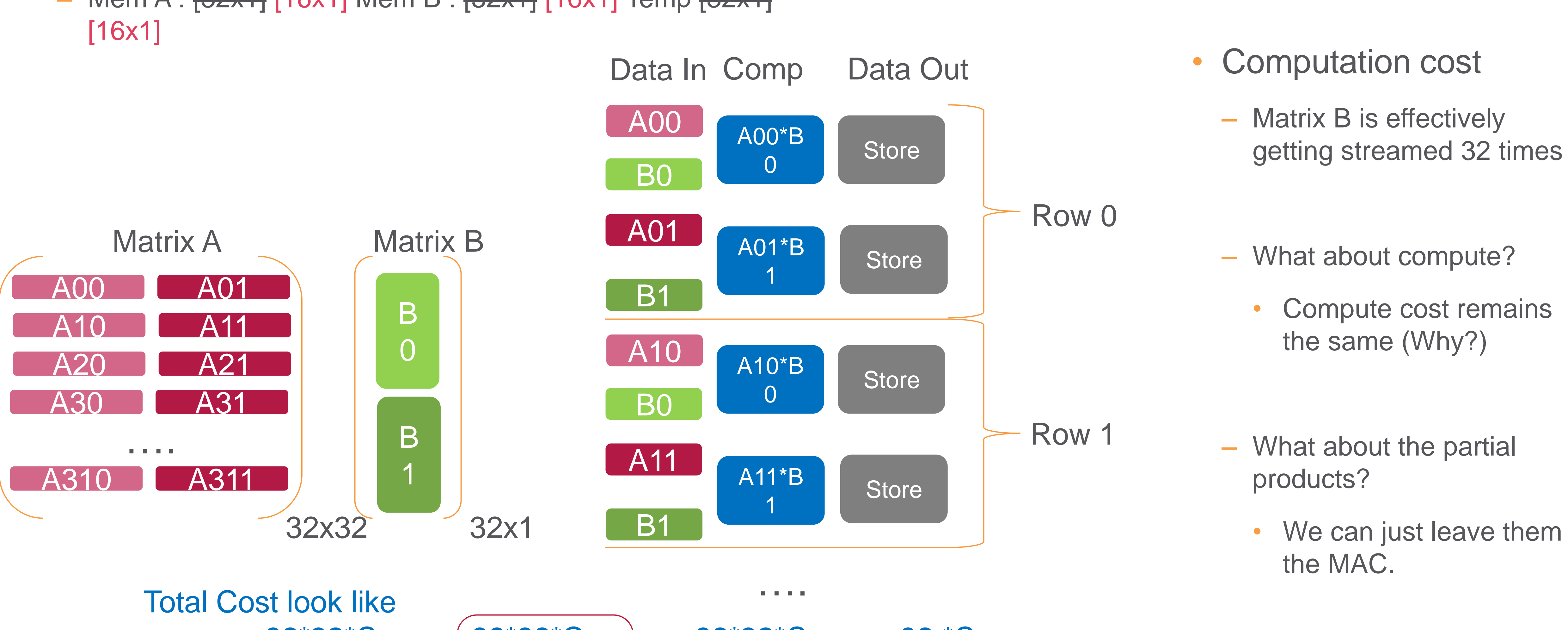
      A transfer   B transfer   Compute   Result transfer





Matrix Multiply with Reduced Memory : Streaming # 1

- Internal Memories are of size:
  - Mem A : ~~[32x1]~~ [16x1] Mem B : ~~[32x1]~~ [16x1] Temp ~~[32x1]~~ [16x1]



Total Cost look like

32*32*Cmem+ 32*32*Cmem + 32*32*Cmac + 32 *Cmem

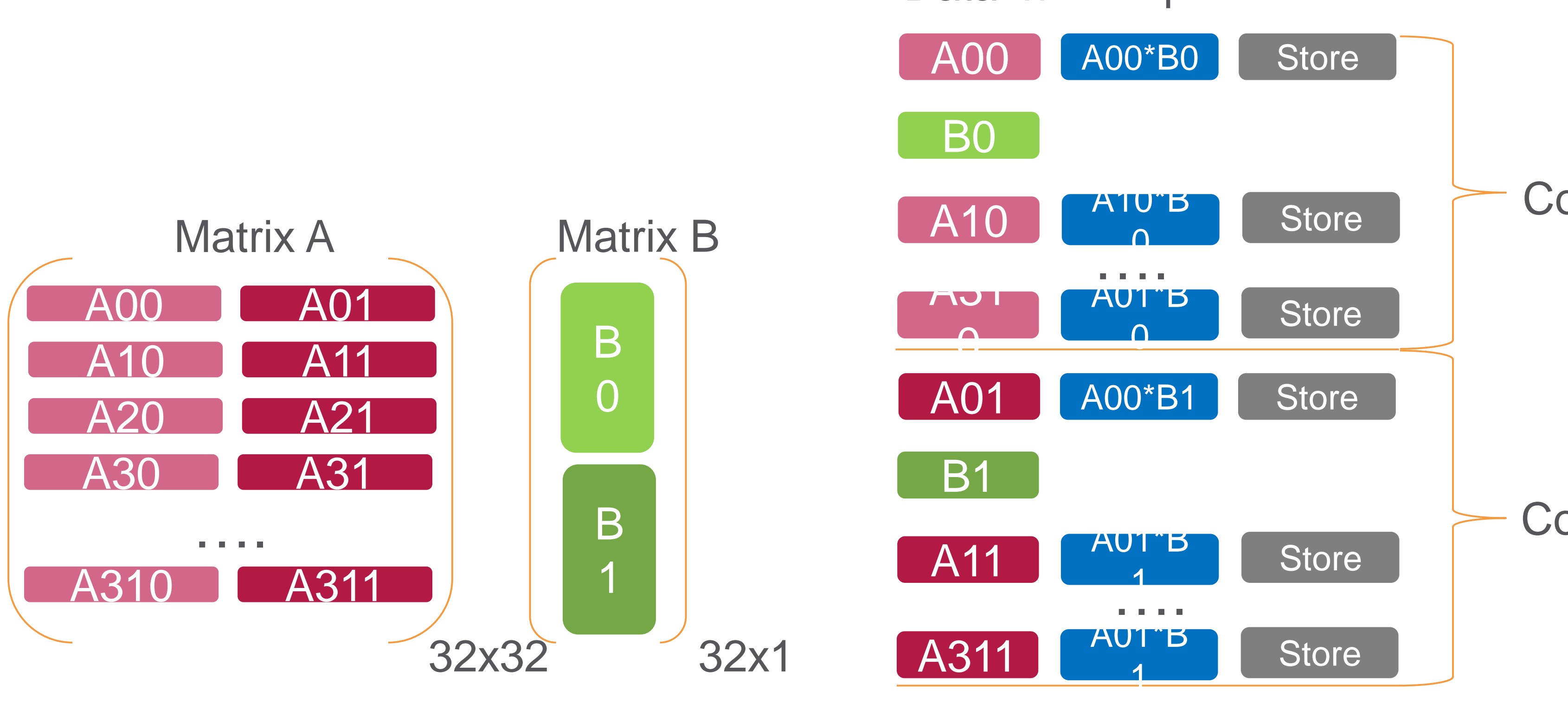
Comparing to prev:

32*32*Cmem+ 32*Cmem + 32*32*Cmac + 32*Cmem

Increased cost of transferring the B matrix

Matrix Multiply with Reduced Memory : Streaming # 2

- Internal Memories are of size:
  - Mem A : ~~[32x1]~~ [16x1] Mem B : ~~[32x1]~~ [16x1] Temp ~~[32x1]~~ [16x1]



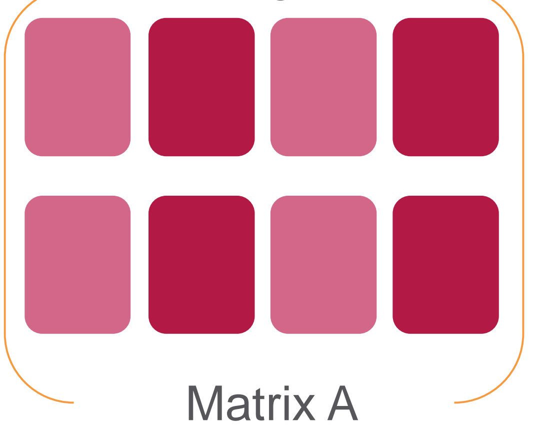
This approach of streaming partial data is also known as **memory tiling**

- Computation cost
  - Matrix B is streamed once.
  - What about compute?
    - Compute cost remains the same. Or does it?
    - Instruction Set Architecture (ISA) dependent
  - There is a hidden cost of moving partials to main memory and back.
  - Another second order effect.
    - We save energy from the movement of data.
    - However, the processor will be idle waiting for next 'A' tile to come.

Cost vs. On chip Memory

- Key takeaways:
  - Cost when memory sizes are:
  - [32x1] : 32*32*Cmem + 32*Cmem + 32*32*Cmac + 32*Cmem
  - [16x1] : 32*32*Cmem + 32*32*Cmem + 32*32*Cmac + 32 *Cmem
  - [16x1] : 32*32*Cmem + 32*Cmem + 32*32*Cmac + 32*Cmem + 32*Cmem + 32*Cmac

  - If we reduce the memory size further to say [4x1] the partial terms become more dominant and it may be beneficial to mix column wise and row wise tiling.



Memory Tiling as a loop transformation

```
# Matrix Multiply
n = 32;
for row_idx in range(0,n):
   for col_idx in range(0,n):
      c[col_idx] += a[row_idx][col_idx]* b[col_idx]
```

```
# Tiled Matrix Multiply
n = 32
tiles = 2
col_per_tile = n/tiles
for tile_num in range(0,tiles):
    for row_idx in range (0,n):
        start_col = cols_per_tile *tile_num
        end_col    = start_col + cols_per_tile
        for col_idx in range(start_col, end_col):
           C[col_idx] += A[row_idx][col_idx] * B [col_idx]
```

- Tiling will result in more complicated control logic.
- Memory access become tricker
  - No longer vanilla row-major or column major
  - Dedicated memory controller
  - Higher silicon cost (and energy)

A few more transformations

```
# Simple Multiplication
N = 1000
for i in range(0,N):
  C[i] = A[i] * B[i]
```

```
# Vector Multiplier
# Single Instruction Multiple Data (SIMD)
N = 1000
VEC_Len = 100
for i in range(0,N/100):
   C[i:VEC_LEN] = A[i:VEC_LEN] * B[i:VEC_LEN]
```

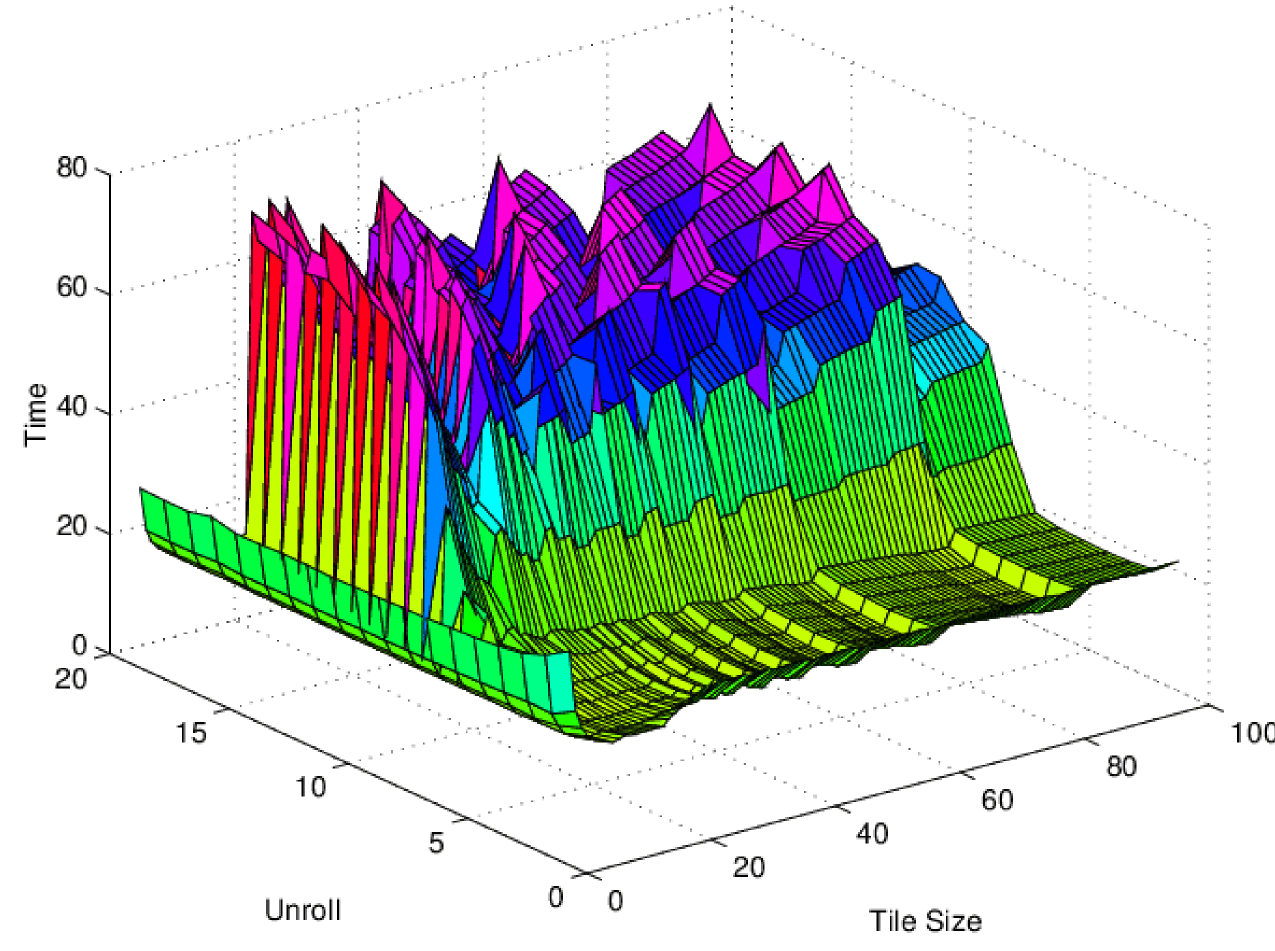
```
# Loop Unroling
N = 1000
for i in range(0,4,500):
   C[i]   = A[i] * B[i]
   C[i+1] = A[i+1] * B[i+1]
   C[i+2] = A[i+2] * B[i+2]
   C[i+3] = A[i+3] * B[i+3]
```

```
# Multi Processor System
N = 1000
#Data dependencies make it a challenge
# Processor 0
for i in range(0,500):
   C[i] = A[i] * B[i]

# Processor 1
for i in range(500,1000):
   C[i] = A[i] * B[i]
```

Performance Loop Unroll vs. Tiling

- Mathematical constructs and heuristics are used for applying these.
- Choosing transformations is not trivial.
- Widely researched, studied and used by the compiler community.
- Optimizations are typically stacked.
- GCC for example can make multiple passes to figure out the sequence.



https://en.wikipedia.org/wiki/Loop_optimization

Knijnenburg, P., Kisuki, T., & O’Boyle, M. (2002). Iterative Compilation. *Embedded Processor Design Challenges*, 171–187. https://doi.org/10.1007/3-540-45874-3_10

Single Layer CNN

Input [5x5]

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

Kernel [3x3]

1	2	-1
1	0	0
0	1	0

Stride =1

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

Output

0	4	1
2	3	-6
6	-4	-2

Single Layer CNN

- Assume that our processor has the following memory constraints
  - A mem = [3x3]
  - B mem = [3x3] (The entire kernel fits)
  - C mem = [1x1]

Stride =1

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

1	0	2	-1	-1
0	1	-1	-2	0
1	1	-2	-1	0
0	-2	1	1	0
-2	1	0	1	0

Single Layer CNN

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

- Naïve solution is reload the matrix under computation (the red part)
- This would be wasteful since there is an overlap.
- Makes sense to load only the new stuff.

Single Layer CNN

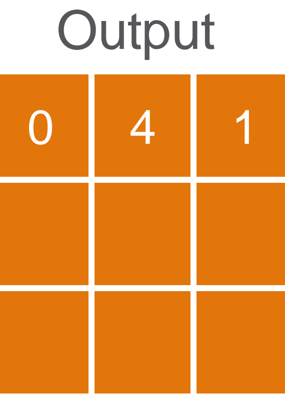
New Data

Existing Data

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4



Single Layer CNN

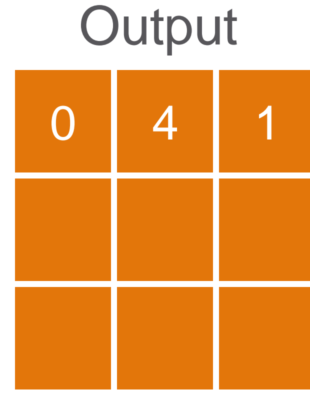
New Data   Existing Data

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

What output should be calculated next?



Single Layer CNN

New Data

Existing Data

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

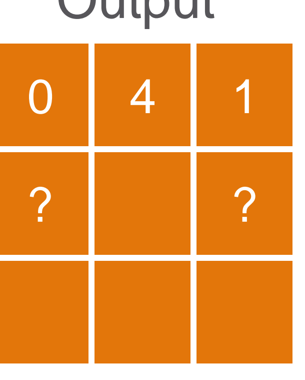
A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

OR

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4



Single Layer CNN

New Data

Existing Data

A00	A01	A02	A03	A04
A10	A11	A12	A13	A14
A20	A21	A22	A23	A24
A30	A31	A32	A33	A34
A40	A41	A42	A43	A44

A00	A01	A02	A03	A04
A10	A11	A12	A13	A14
A20	A21	A22	A23	A24
A30	A31	A32	A33	A34
A40	A41	A42	A43	A44

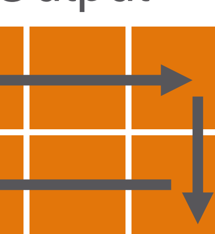
A00	A01	A02	A03	A04
A10	A11	A12	A13	A14
A20	A21	A22	A23	A24
A30	A31	A32	A33	A34
A40	A41	A42	A43	A44

A00	A01	A02	A03	A04
A10	A11	A12	A13	A14
A20	A21	A22	A23	A24
A30	A31	A32	A33	A34
A40	A41	A42	A43	A44

A00	A01	A02	A03	A04
A10	A11	A12	A13	A14
A20	A21	A22	A23	A24
A30	A31	A32	A33	A34
A40	A41	A42	A43	A44

A00	A01	A02	A03	A04
A10	A11	A12	A13	A14
A20	A21	A22	A23	A24
A30	A31	A32	A33	A34
A40	A41	A42	A43	A44

Output



Single Layer CNN

New Data   Existing Data

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

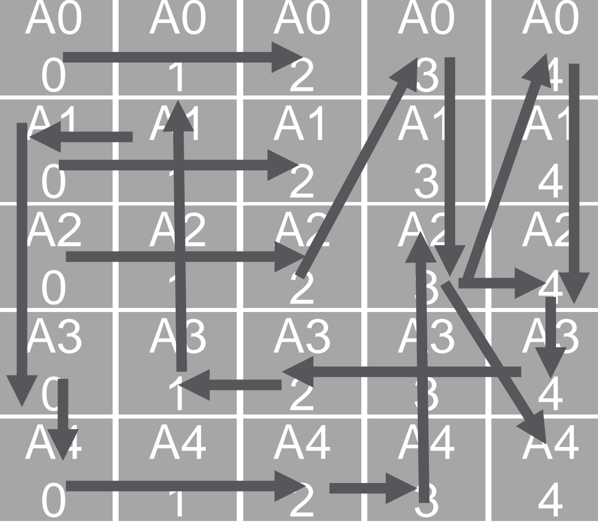
A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

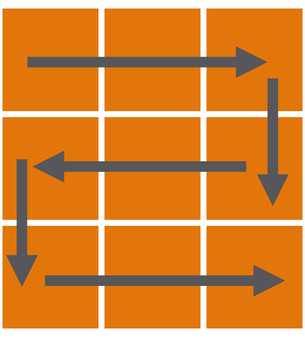
A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

A0 0	A0 1	A0 2	A0 3	A0 4
A1 0	A1 1	A1 2	A1 3	A1 4
A2 0	A2 1	A2 2	A2 3	A2 4
A3 0	A3 1	A3 2	A3 3	A3 4
A4 0	A4 1	A4 2	A4 3	A4 4

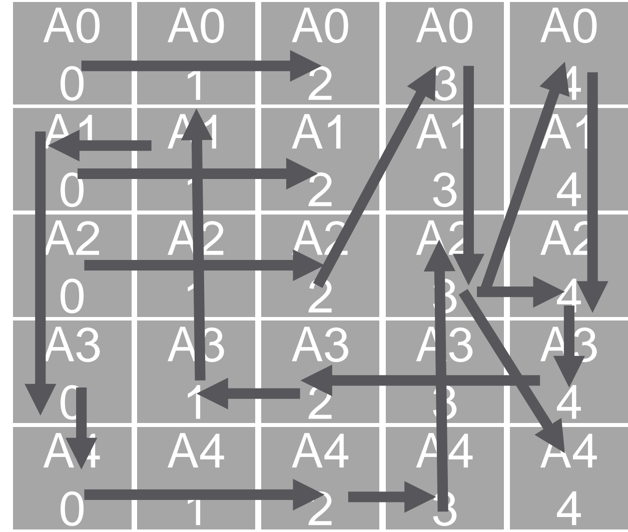
Memory Access Pattern

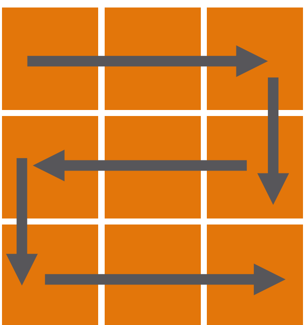


Output



CNN

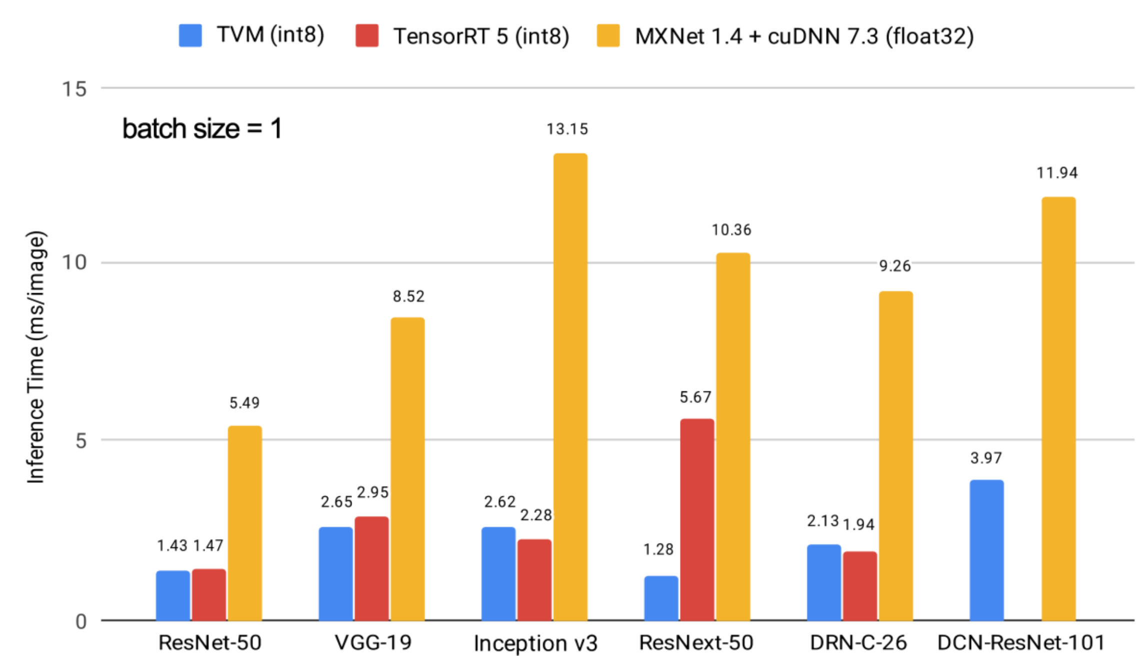


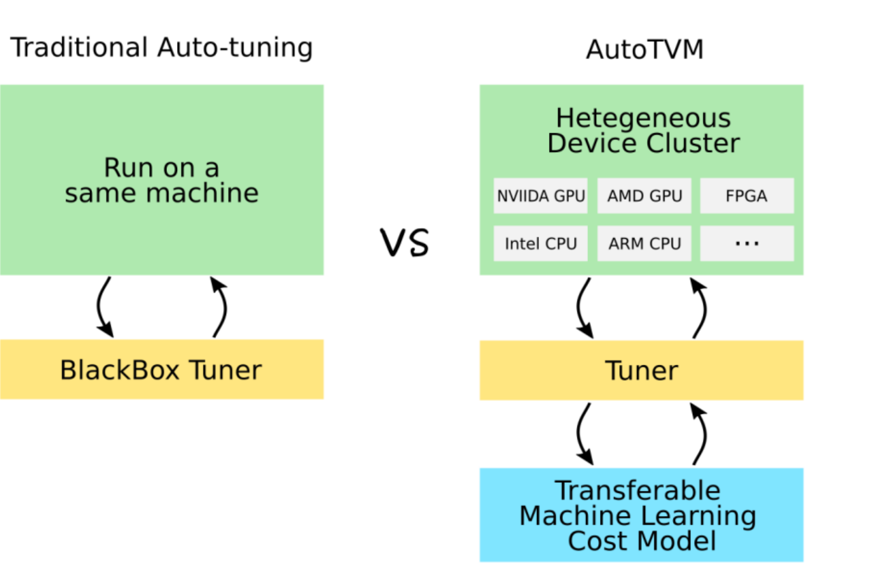


- Realistic scenario
  - Memory access pattern can be very very complicated.
  - Depending on the compute constraints and data bus constraints might be beneficial to repeat data, pad data or other tricks.
  - Partial products need to be accounted for and stored correctly.
    - Can result in significant cost.
  - Fueling research in both compression and quantization
  - Non-Trivial to evaluate ”effective” cost specially when multiple loop optimizations can be performed.

Machine Learning and Compilers

- TVM
  – https://tvm.apache.org/docs/tutorials/
- FB Glow
  – https://ai.facebook.com/tools/glow/
- Survey of ML techniques
  – Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., & Silvano, C. (2018). A Survey on Compiler Autotuning using Machine Learning. Retrieved from http://arxiv.org/abs/1801.04405
- Other compiler frameworks with interesting optimizations
  – LLVM, Halide, Pencil, Poly, Pluto, JIT, Chill
- GEMM Kernels
  – https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf





Key takeaways

- Memory affects the cost of data movement more than it affects computation in mid-memory range for very low memory on-chip footprints computation cost can become prohibitive.

- Quantization and compression are essential because more data can be moved in between memory and processor on same the same databus.
  - The caveat is that the compute engine should be able to take advantage.
  - A CNN accelerator can’t take advantage of binary network
    - unless data bus can shuttle data in 1-bit quantum
    - The processor should support an XOR based multiplier.
  - Most compute-in-memory accelerators take advantage of binary networks because they can be stored on-chip.

- Automated solutions like TVM and Glow are essential because the exploration space in non-trivial.

- Creating a tinyML product you should have some understanding of:
  - Memory patterns supported
  - Compute support and flexibility (16bit, 8bit, 4bit).
  - Blackbox toolchain support and flexibility to tune kernels.

Copyright Notice

This presentation in this publication was presented as a tinyML® Talks webcast. The content reflects the opinion of the author(s) and their respective companies. The inclusion of presentations in this publication does not constitute an endorsement by tinyML Foundation or the sponsors.

There is no copyright protection claimed by this publication. However, each presentation is the work of the authors and their respective companies and may contain copyrighted material. As such, it is strongly encouraged that any use reflect proper acknowledgement to the appropriate source. Any questions regarding the use of any materials presented should be directed to the author(s) or their companies.

tinyML is a registered trademark of the tinyML Foundation.

www.tinyML.org