High Performance Matrix Multiplication

CS 217 Stanford

Ardavan Pedram

Basic Linear Algebra Subprograms (BLAS)

- The building blocks of most of scientific computing
- Application codes either call these routines directly, indirectly through other libraries that themselves call the BLAS, or via environments like Matlab
  Three Levels:
  - Level 1: vector-vector operations $z+ = \alpha x, \quad z = x^T y, \quad z = |x|_2$
  - Level 2: matrix-vector operations $y = \alpha Ax + \beta y, \quad Solve\ Lx = b$
  - Level 3: matrix-matrix operations $C = \alpha AB + \beta C, \quad Solve\ LX = B$

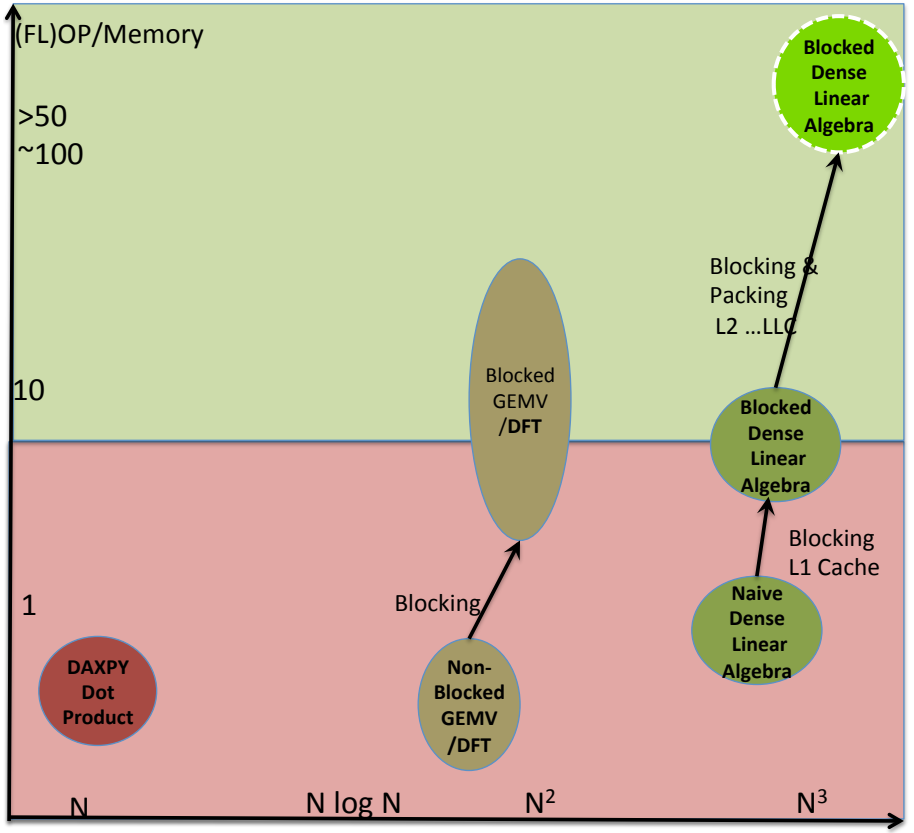
Basic Linear Algebra Subprograms (BLAS)

- Implementing level-3 BLAS in terms of GEneral Matrix-Matrix multiplication (GEMM) kernels.
  - B. Kagstrom, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. ACM Trans. Math. Soft., 24(3):268–302, 1998.
    Only a small set of kernels needed to be highly optimized.

- IBM's Algorithms and Architectures approach as part of the development of the Power processor:
  - R. C. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism on Power2 to design high-performance numerical algorithms. IBM Journal of Research and Development, 1994.
    If architectures, algorithms, and compilers were developed in tandem, then BLAS can be implemented in Fortran.

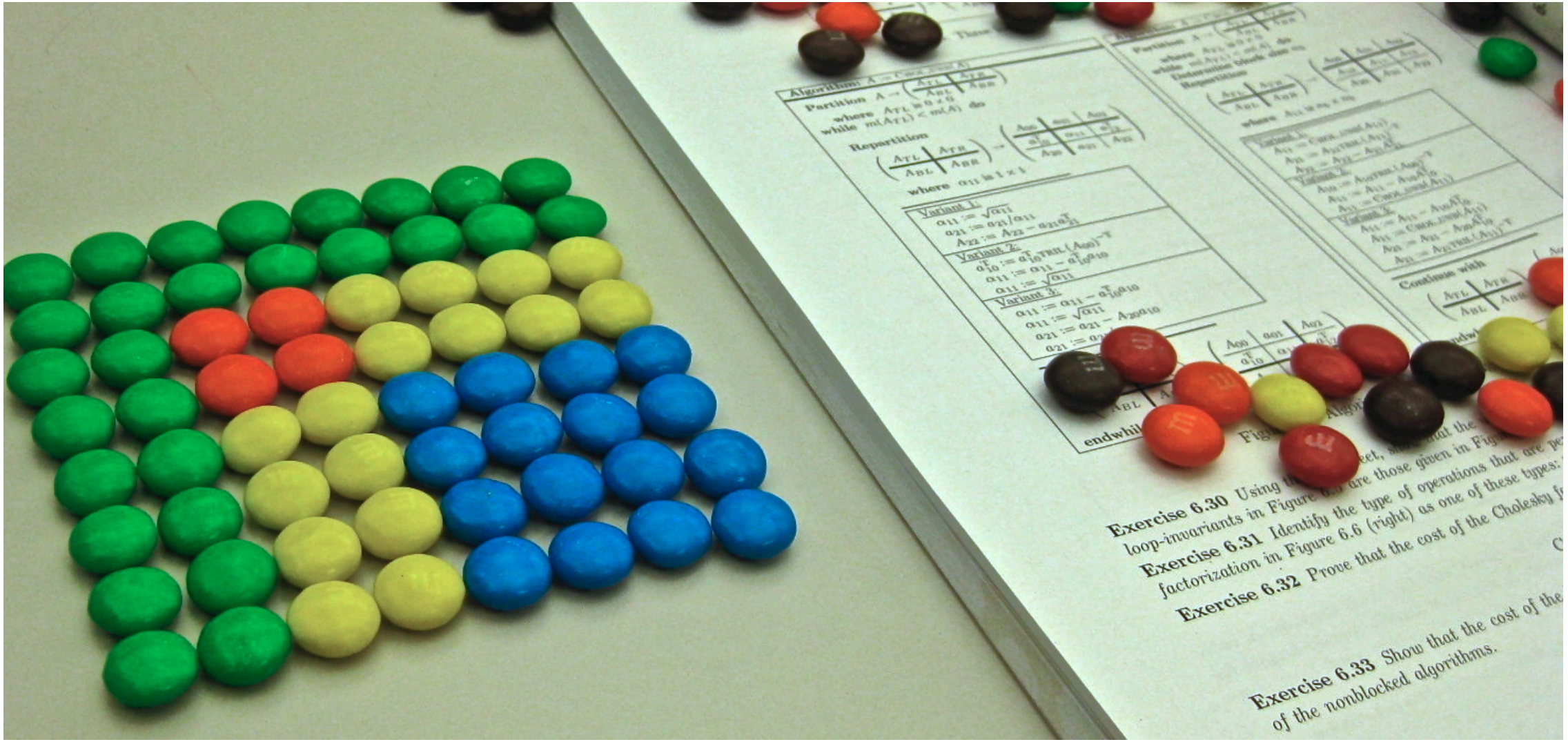
BLAS Complexity

BLAS level	Ex.	# mem refs	# flops	Flop/ memory
1	"Axpy" Dot prod	3n	$2n^1$	2/3
2	"GEMV" Matrix-vector mult	n^2	$2n^2$	2
3	"GEMM" Matrix-matrix mult	$4n^2$	$2n^3$	n/2



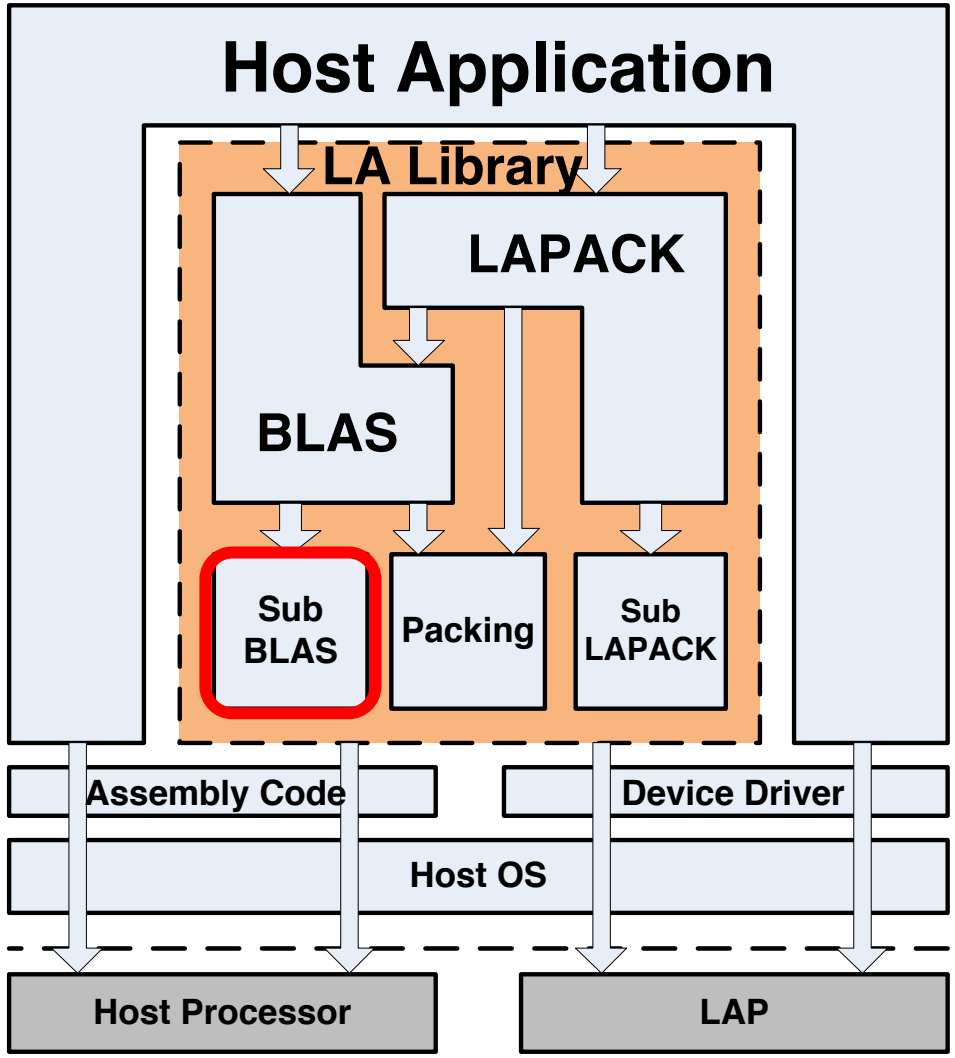
Nominal Complexity or the big O

Why Linear Algebra?



Why GEMM?

- Linear Algebra Package (LAPACK) level
  - Matrix factorizations
- Basic Linear Algebra Subroutines (BLAS)
  - Matrix-matrix and matrix-vector operations
- Inner kernels
  - Hand-optimized
- Accelerators
  - optimize GEMM
- **GEMM is crucial for many critical applications**

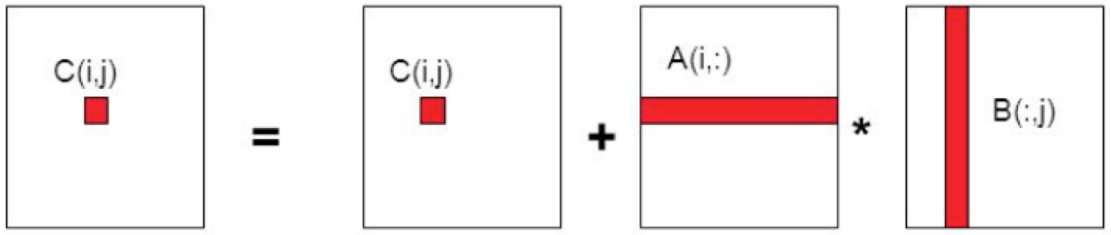


GEMM is 3 Nested Loops

```
{implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory

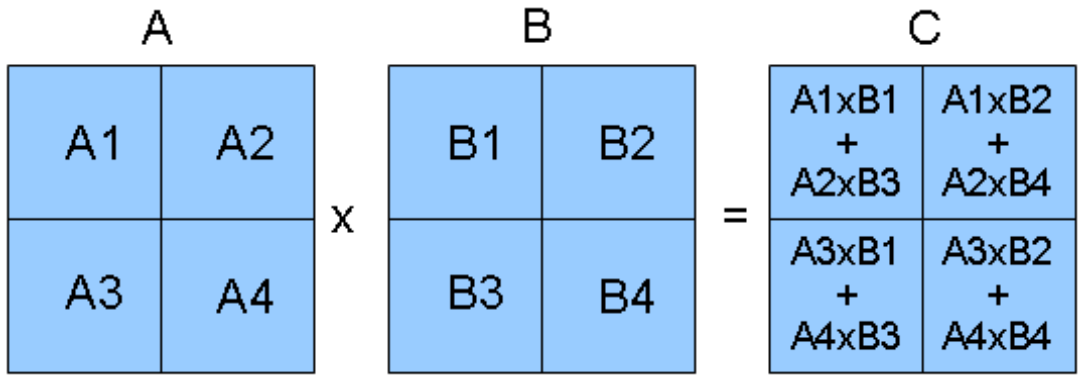
q potentially as large as $2*n^3 / 3*n^2 = O(n)$



- What is the performance of this code?
- Where is the cause of problem with this code's?
- How can we improve it ?
- What is an easy way to increase the performance with minimum code change?

Cache Oblivious/ Recursive GEMM

- Divide & conquer
- How does it work with non-squared matrices?
- Why Does it improve performance?



$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

Case 1

$$(A_1, A_2)\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (A_1 B + A_2 B)$$

Case 2

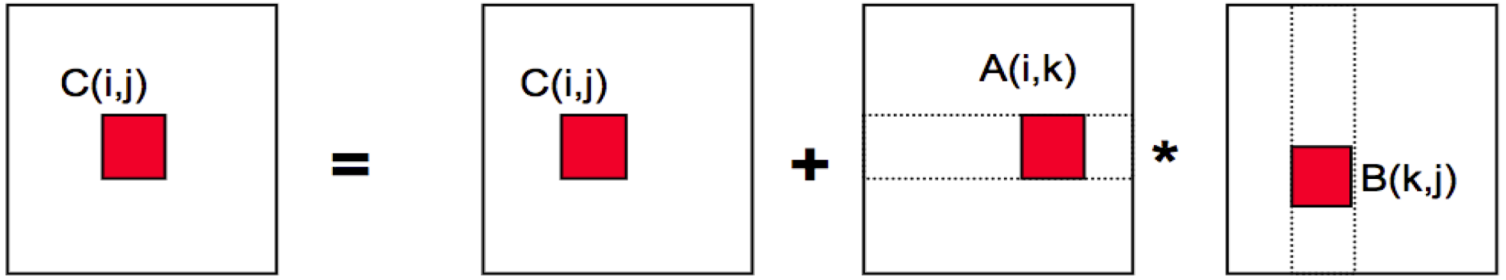
$$A(B_1, B_2) = (A\,B_1, A\,B_2)$$

Case 3

Blocked GEMM

Consider A,B,C to be n-by-n matrix viewed as N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```



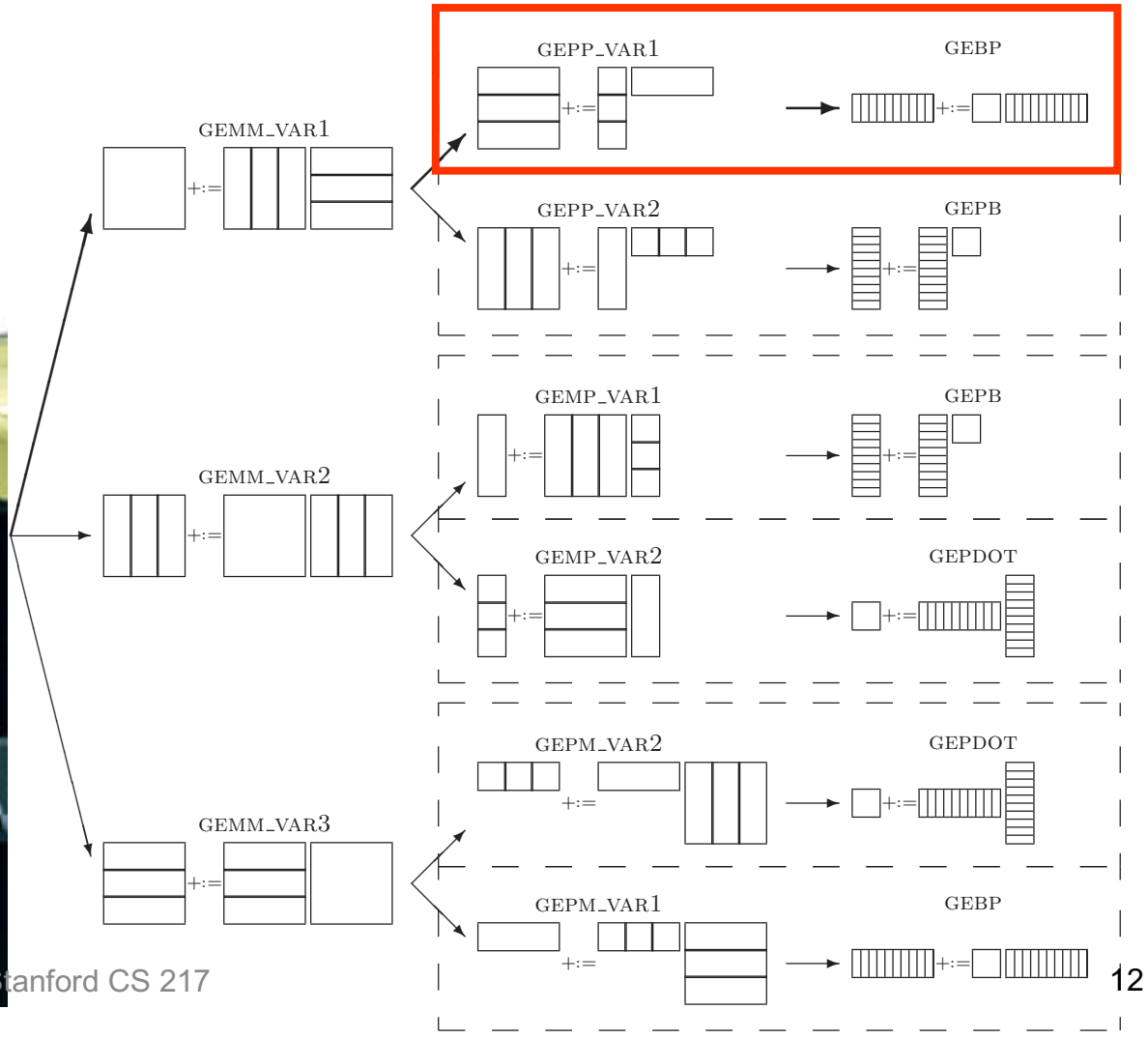
Source James Demmel


DUH
AND/OR
HELLO!

Blocking: A Tradition

General Matrix-Matrix Multiplication (GEMM)

- Blocked algorithm variants
  - Fastest general-purpose implementation [GotoBLAS]



GEMM Kernel

- Rank-1 update
  - Update matrix by adding outer product of two vectors to it

$$\begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,3} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} & \cdots & \gamma_{3,3} \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{0,0} \\ \vdots \\ \alpha_{3,0} \end{pmatrix} \begin{pmatrix} \beta_{0,0} & \cdots & \beta_{0,3} \end{pmatrix}$$

- Matrix multiplication as a series of rank-1 updates

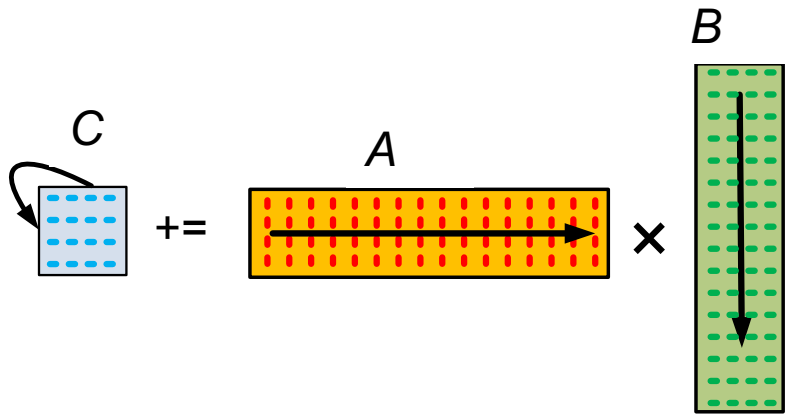
Let *C, A*, and *B* be 4x4, 4xk_c, and k_cx4 matrices.

C+=AB can be computed as:

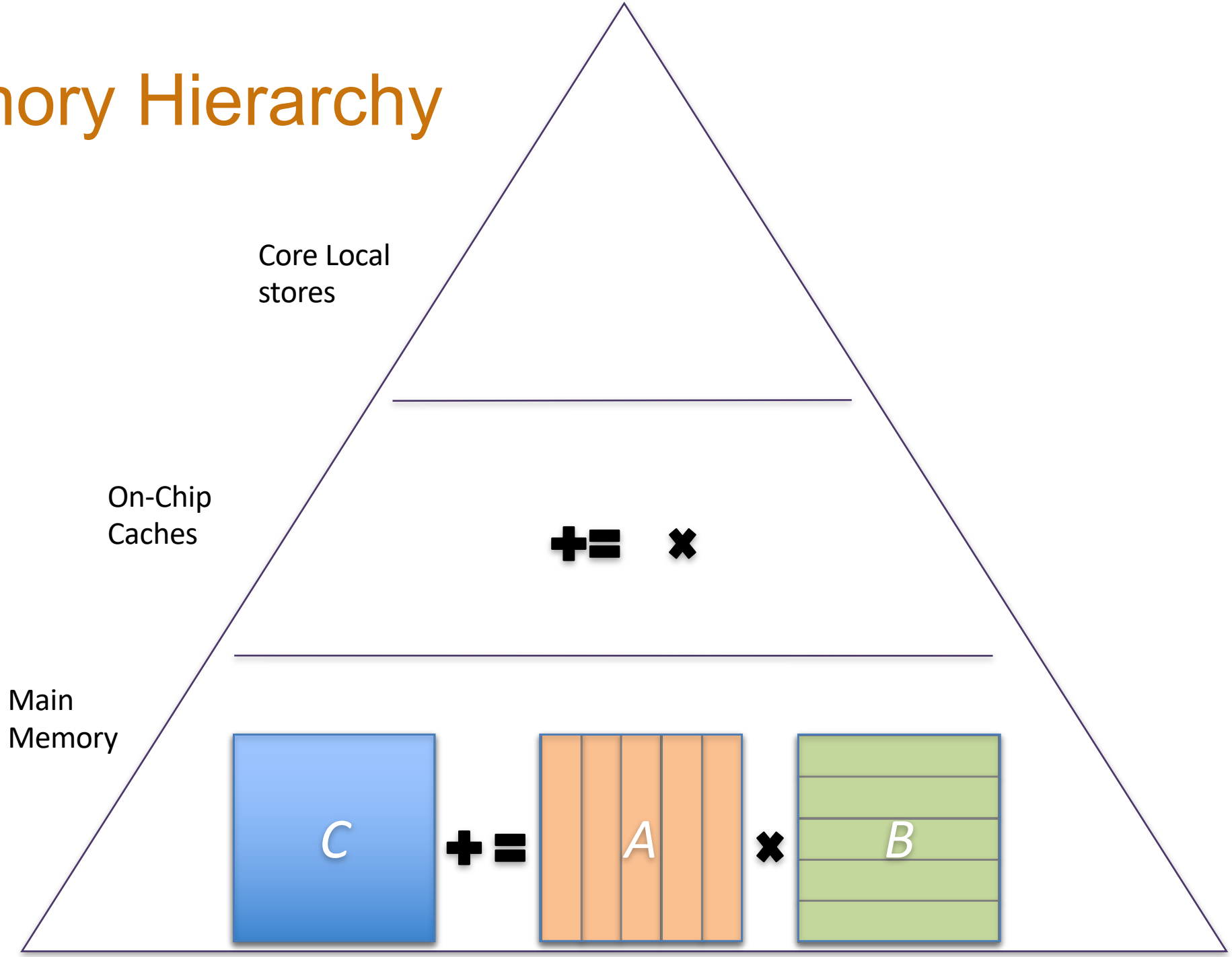
for *i=0* to k_c*-1*

$$\begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,3} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} & \cdots & \gamma_{3,3} \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{0,i} \\ \vdots \\ \alpha_{3,i} \end{pmatrix} \begin{pmatrix} \beta_{i,0} & \cdots & \beta_{i,3} \end{pmatrix}$$

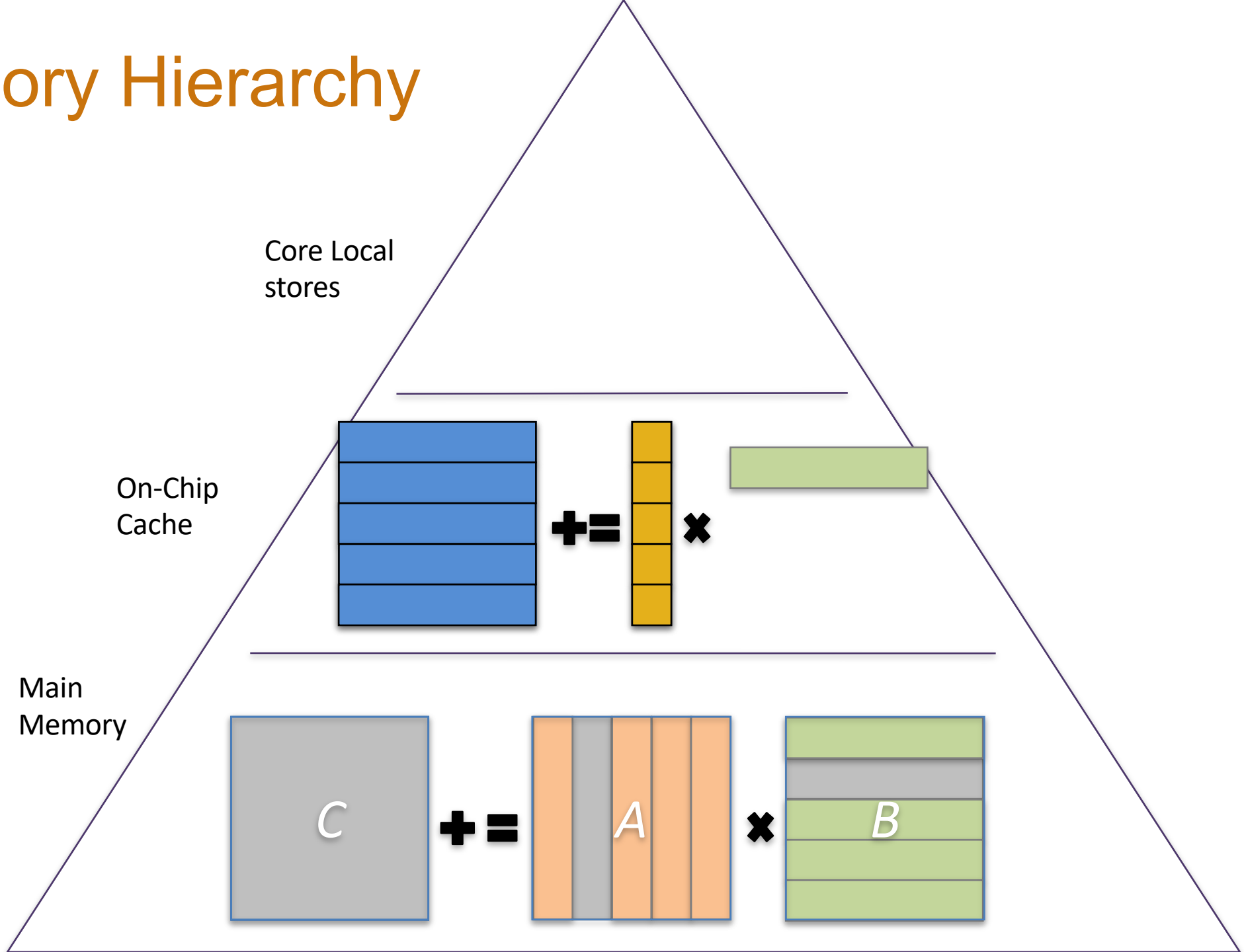
end for



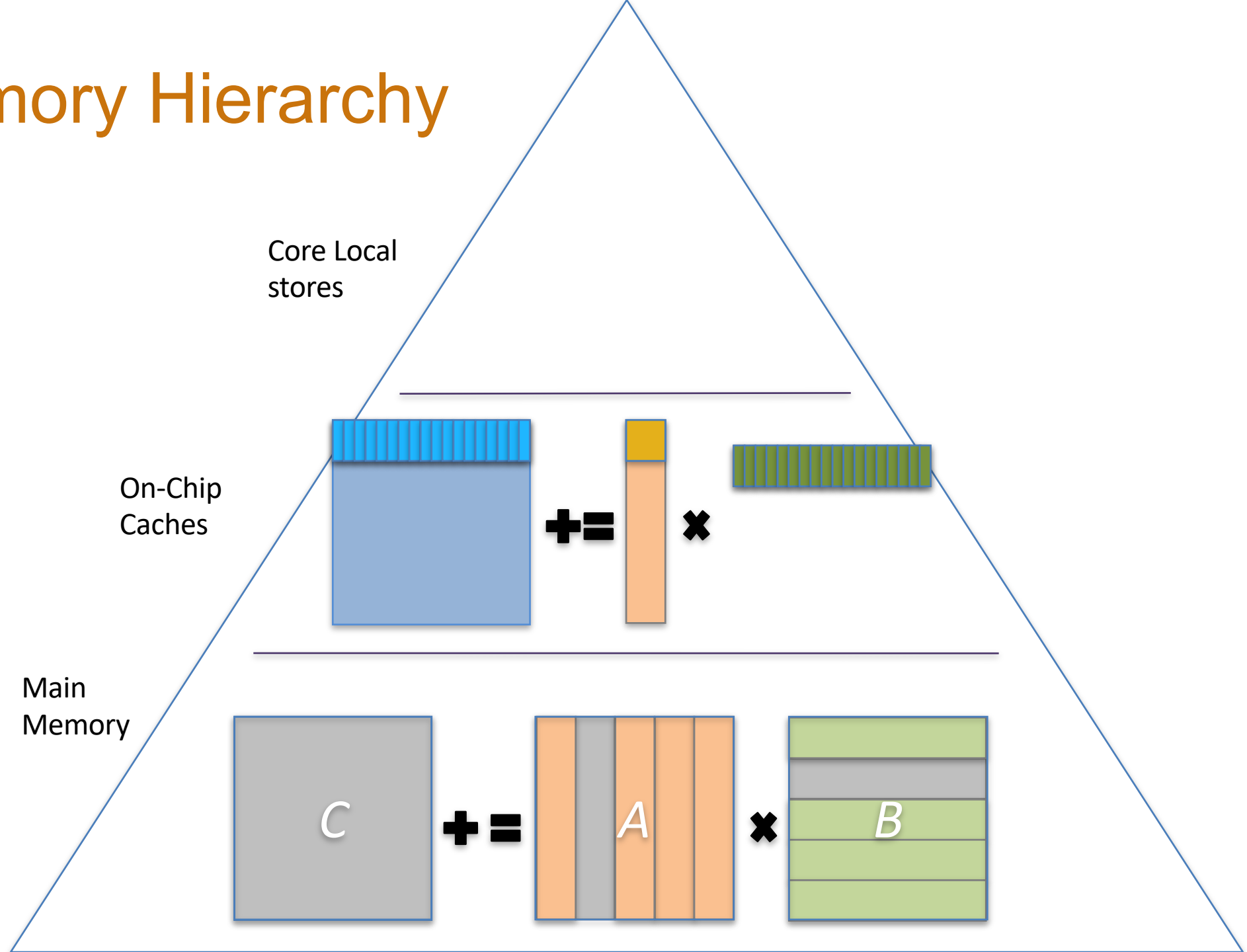
Memory Hierarchy



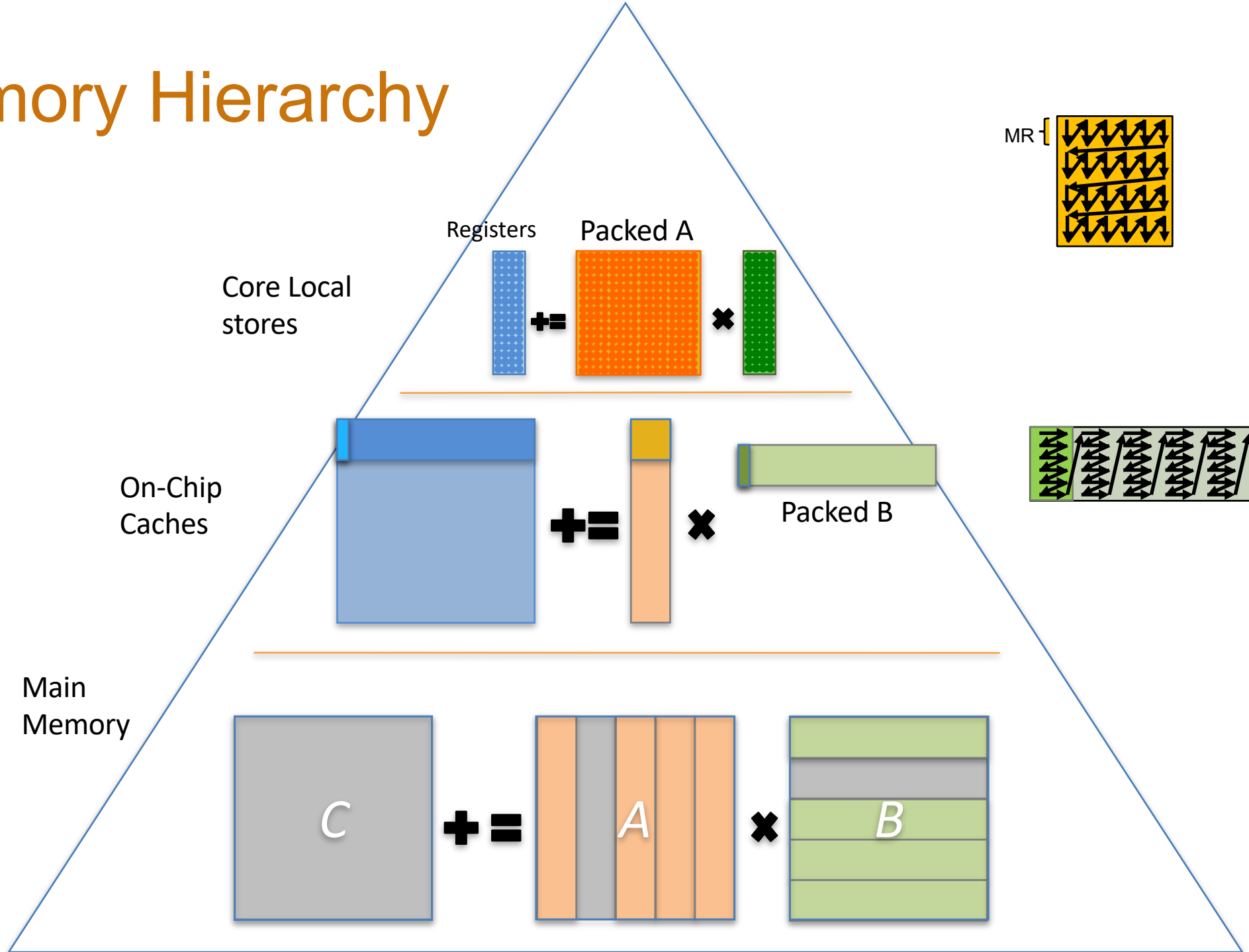
Memory Hierarchy



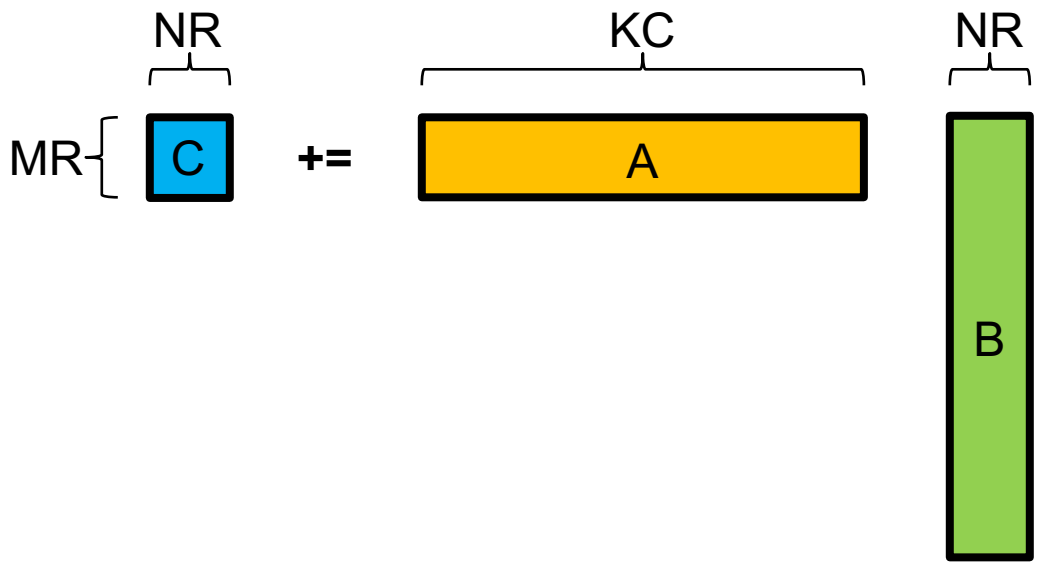
Memory Hierarchy



Memory Hierarchy

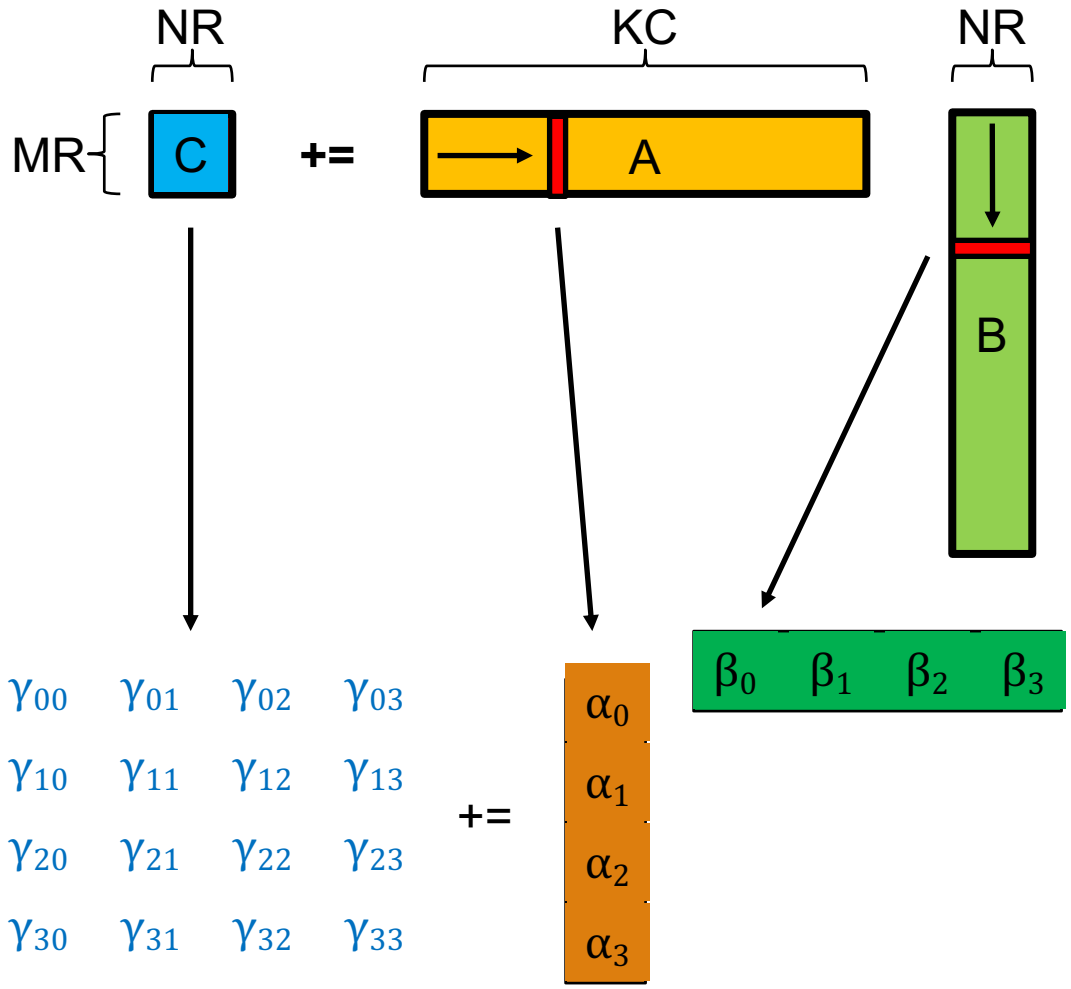


The GEMM micro-kernel




```
for ( 0 to NC: NR )
  for ( 0 to MC: MR )
    for ( 0 to KC: 1 )
      // block-dot product
    endfor
  endfor
endfor
```

The GEMM micro-kernel

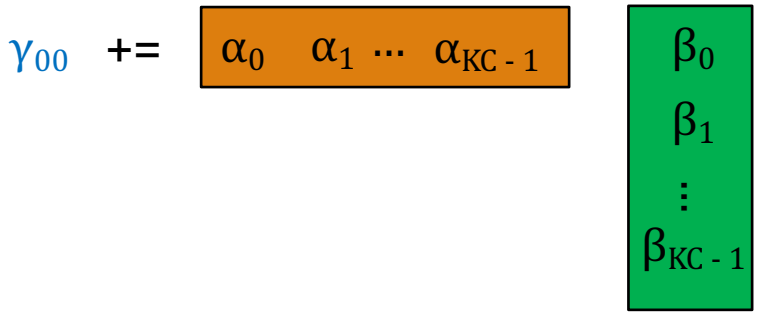



```
for ( 0 to NC: NR )
   for ( 0 to MC: MR )
      for ( 0 to KC: 1 )
         // block-dot product
      endfor
   endfor
endfor
```

- Typical micro-kernel loop iteration ("block-dot product")
  - Load column of packed A
  - Load row of packed B
  - Compute outer product
  - Update C (kept in registers)

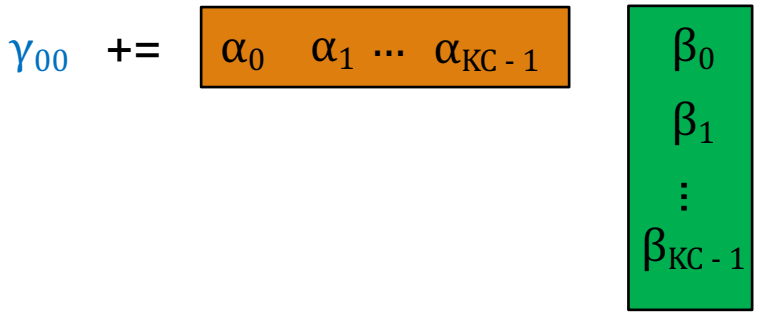
The GEMM micro-kernel

- Why MR x NR block-dot product?
  - Why not a simple scalar dot product?



The GEMM micro-kernel

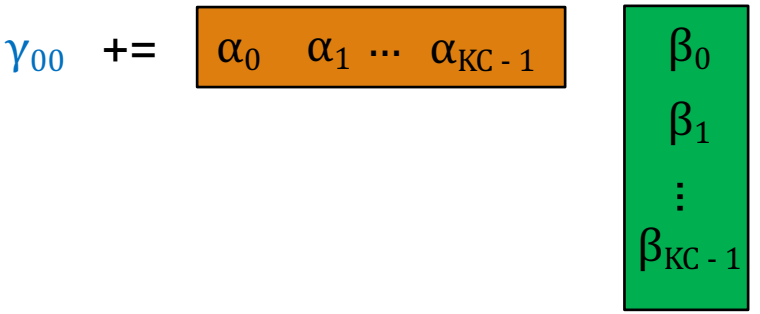
- Why MR x NR block-dot product?
  - Why not a simple scalar dot product?
  - Ratio of floating-point operations to memory operations



The GEMM micro-kernel

- Why MR x NR block-dot product?
  - Why not a simple scalar dot product?
  - Ratio of floating-point operations to memory operations

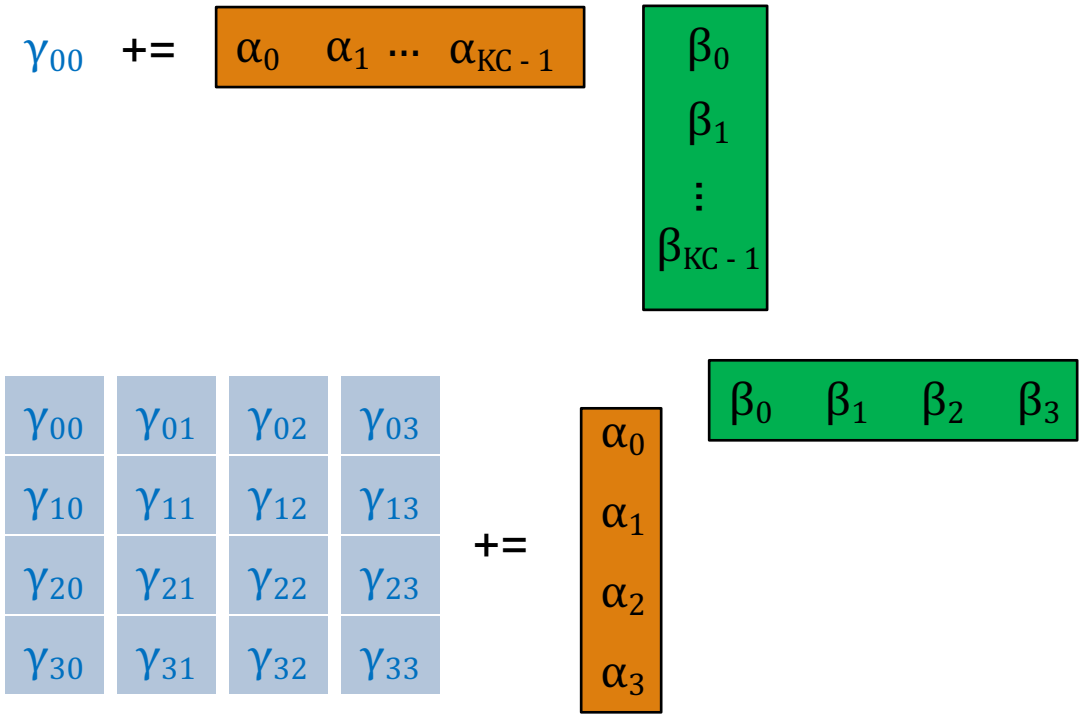
- $\frac{2\,K_C}{2K_C+2} \approx 1$



The GEMM micro-kernel

- Why MR x NR block-dot product?
  - Why not a simple scalar dot product?
  - Ratio of floating-point operations to memory operations

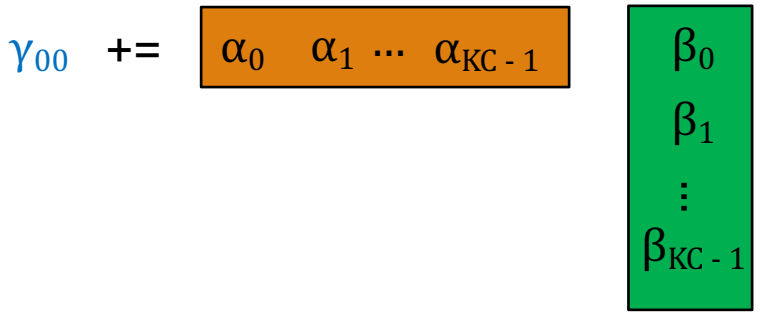
- $\frac{2\,K_C}{2K_C+2} \approx 1$



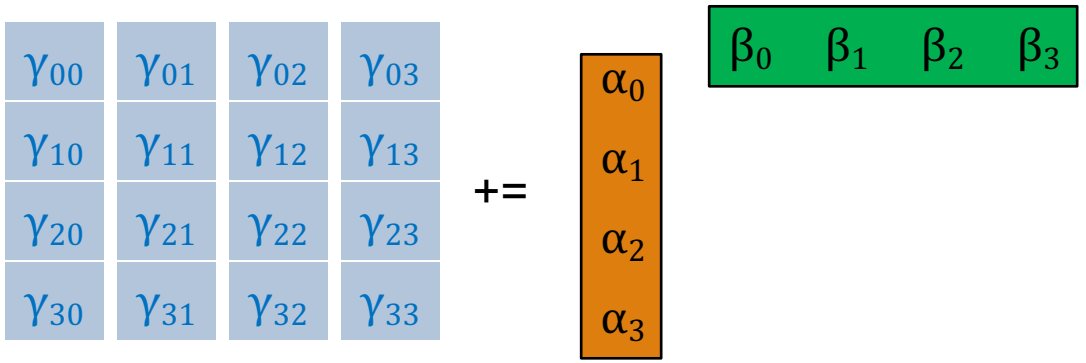
The GEMM micro-kernel

- Why MR x NR block-dot product?
  - Why not a simple scalar dot product?
  - Ratio of floating-point operations to memory operations

- $\frac{2\,K_C}{2K_C+2} \approx 1$



- $\frac{2M_R N_R K_C}{(M_R+N_R)K_C+2M_R N_R}$
- $\approx \frac{2M_R N_R}{(M_R+N_R)}$

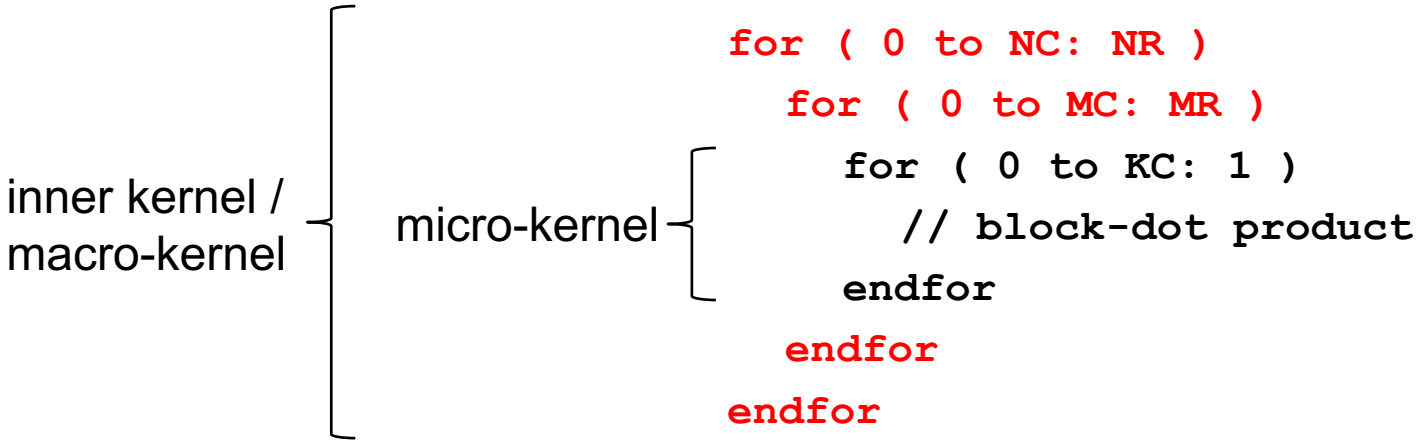


The GEMM micro-kernel

- That's all the developer has to write for gemm?
  - Not just gemm, almost *any* level-3 BLAS-like operation
- Wait, what? So whereas before I needed a few dozen inner kernels (each one ≈ 2000 lines), now I need?
  - One micro-kernel (≈ 700 lines) per datatype
- Remind me, how is this possible?

The GEMM micro-kernel

- BLIS exposes outer two loops of the inner kernel
  - Key observation: Virtually *all* of the differences between level-3 inner kernels reside in the outer two loops (e.g. loop bounds)
  - All inner kernels (which we call “macro-kernels”) are provided as part of the BLIS framework (C99)
  - Inner-most exposed loop simply calls micro-kernel

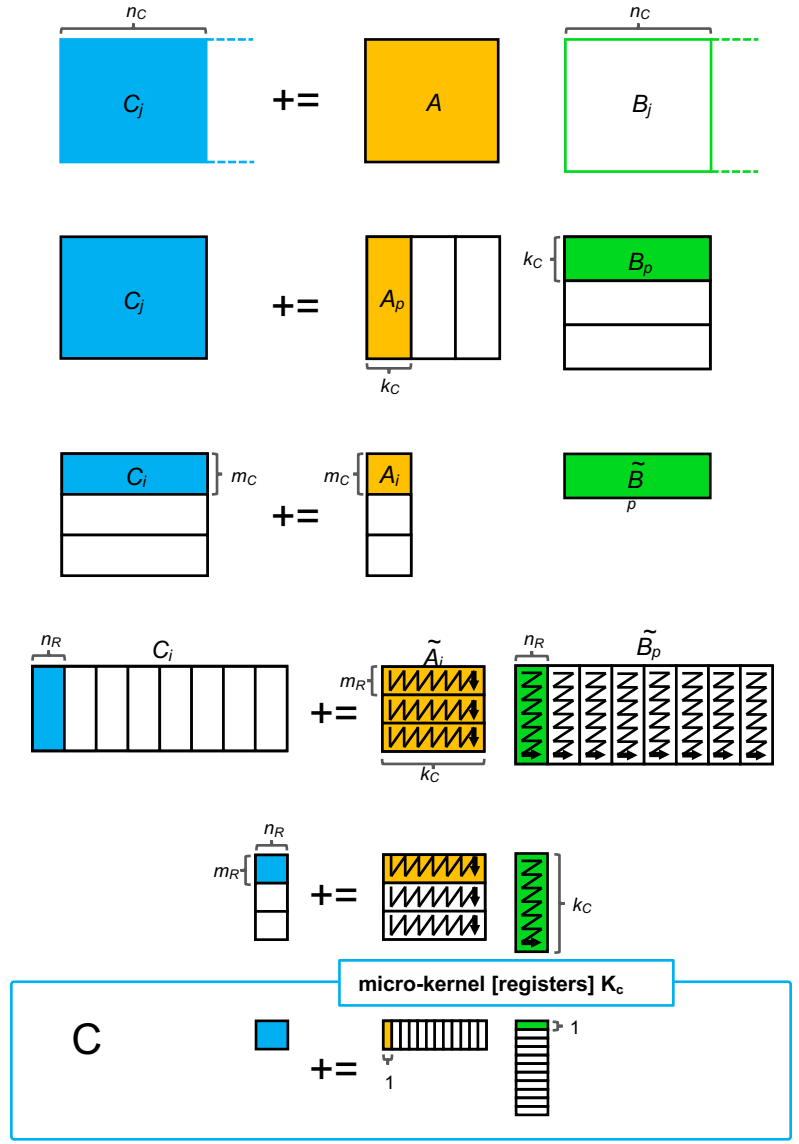


Recap of GEMM Loops

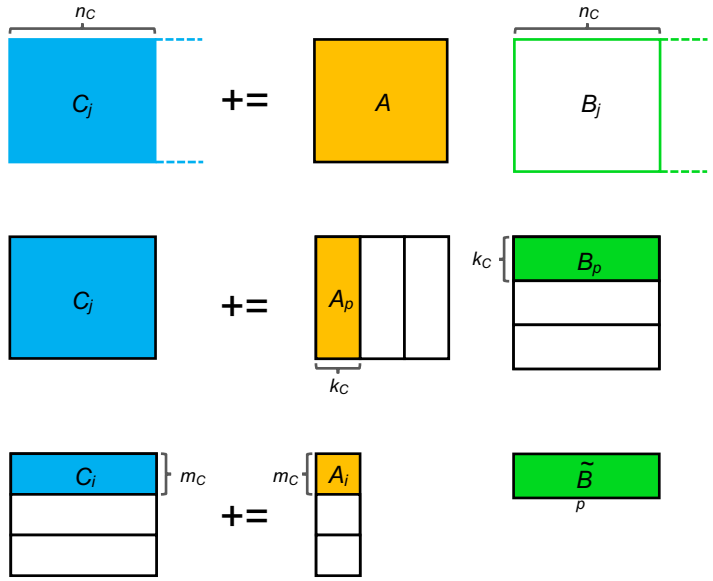
for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$

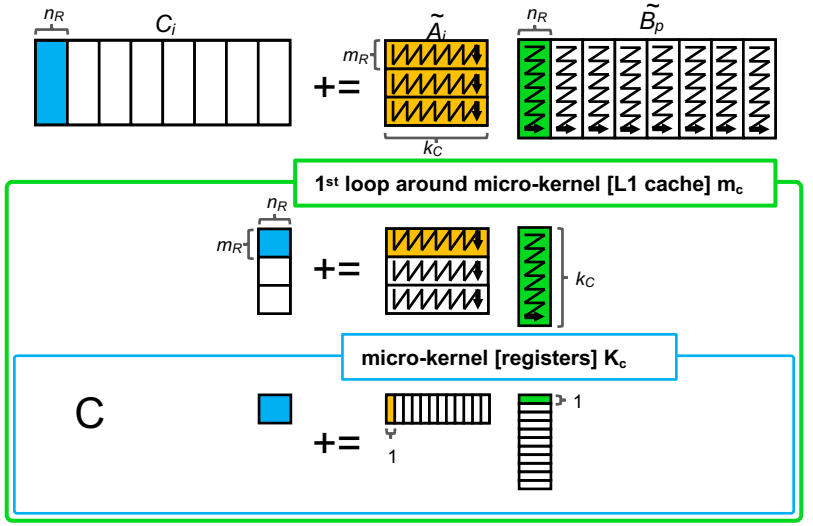
---

      for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
          for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
            $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
              $+= A_c(i_r : i_r + m_r - 1, p_r)$
              $\cdot \; B_c(p_r, j_r : j_r + n_r - 1)$



Recap of GEMM Loops



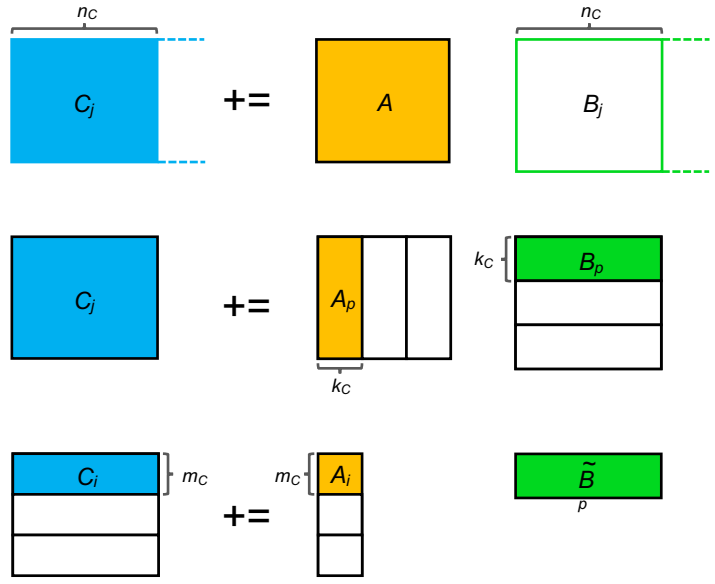


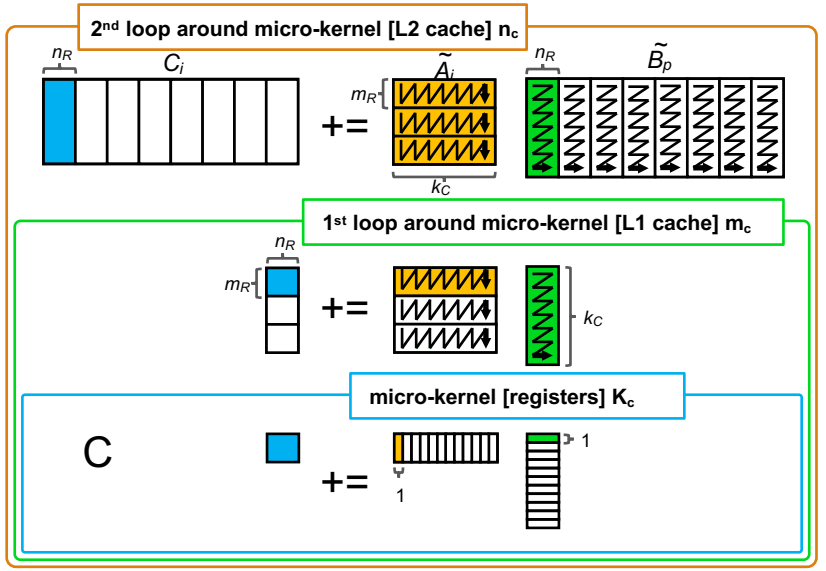
for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$

for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
  for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r

for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
  $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
  $+= A_c(i_r : i_r + m_r - 1, p_r)$
  $\cdot \; B_c(p_r, j_r : j_r + n_r - 1)$

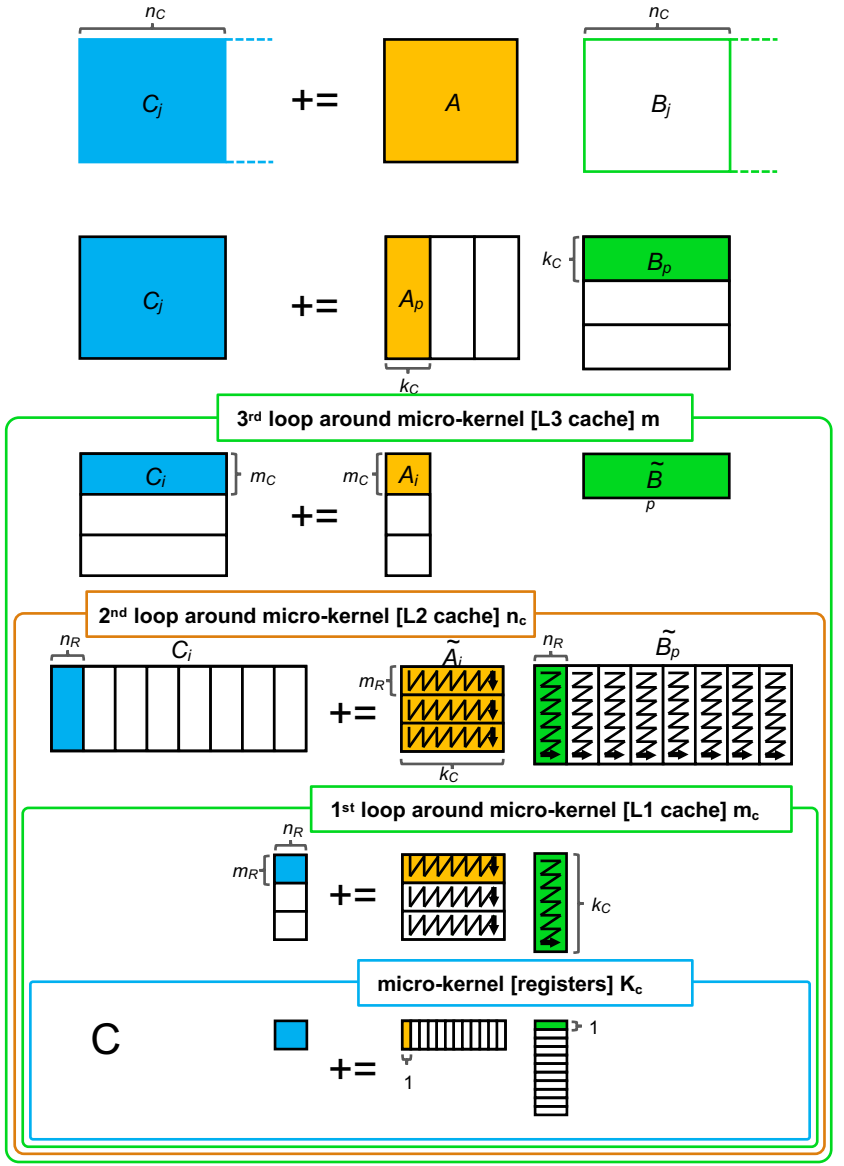
Recap of GEMM Loops





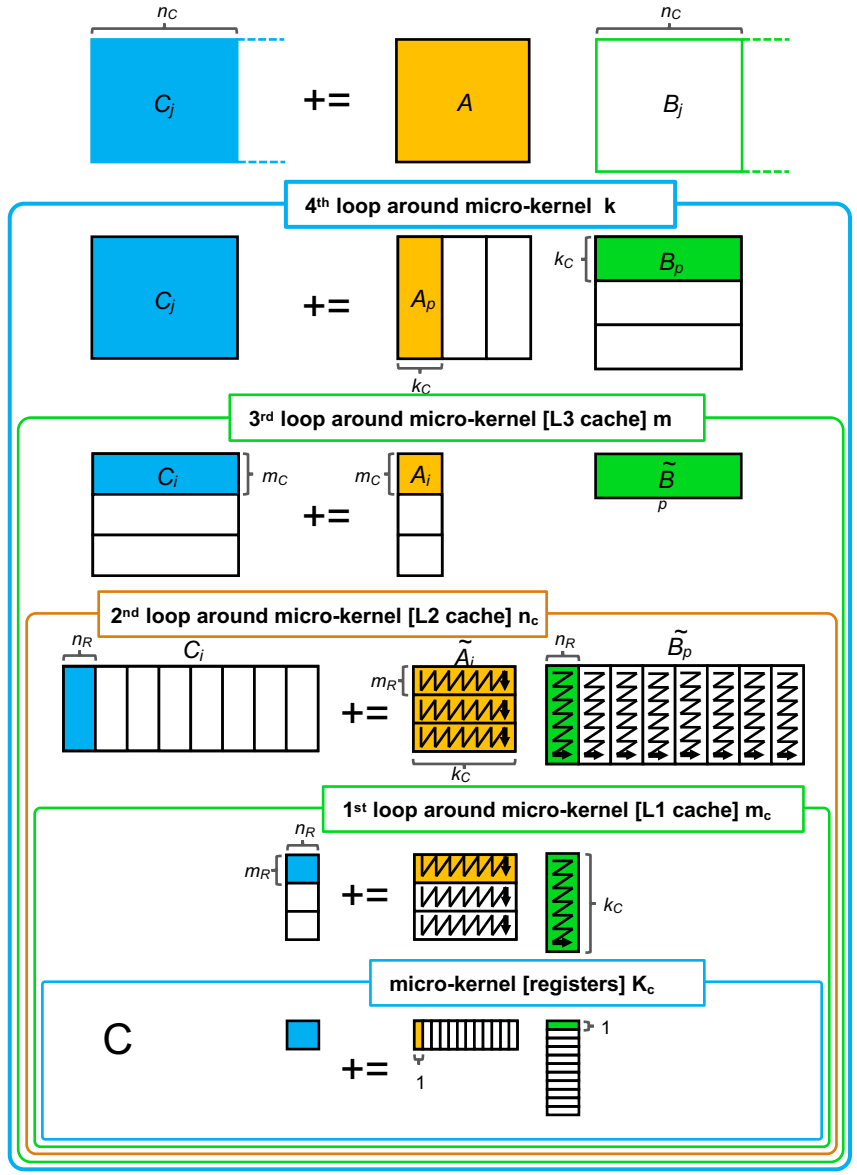
$$
\begin{array}{l}
\textbf{for } j_c = 0, \ldots, n-1 \textbf{ in steps of } n_c \\
\quad \textbf{for } p_c = 0, \ldots, k-1 \textbf{ in steps of } k_c \\
\quad\quad B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c \\
\quad\quad \textbf{for } i_c = 0, \ldots, m-1 \textbf{ in steps of } m_c \\
\quad\quad\quad A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c \\
\quad\quad\quad \textbf{for } j_r = 0, \ldots, n_c - 1 \textbf{ in steps of } n_r \\
\quad\quad\quad\quad \textbf{for } i_r = 0, \ldots, m_c - 1 \textbf{ in steps of } m_r \\
\quad\quad\quad\quad\quad \textbf{for } p_r = 0, \ldots, k_c - 1 \textbf{ in steps of } 1 \\
\quad\quad\quad\quad\quad\quad C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\
\quad\quad\quad\quad\quad\quad\quad \mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r) \\
\quad\quad\quad\quad\quad\quad\quad \cdot\ B_c(p_r, j_r : j_r + n_r - 1)
\end{array}
$$

Recap of GEMM Loops



for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$
      for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
          for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
            $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
              $+= A_c(i_r : i_r + m_r - 1, p_r)$
              $\cdot\ B_c(p_r, j_r : j_r + n_r - 1)$

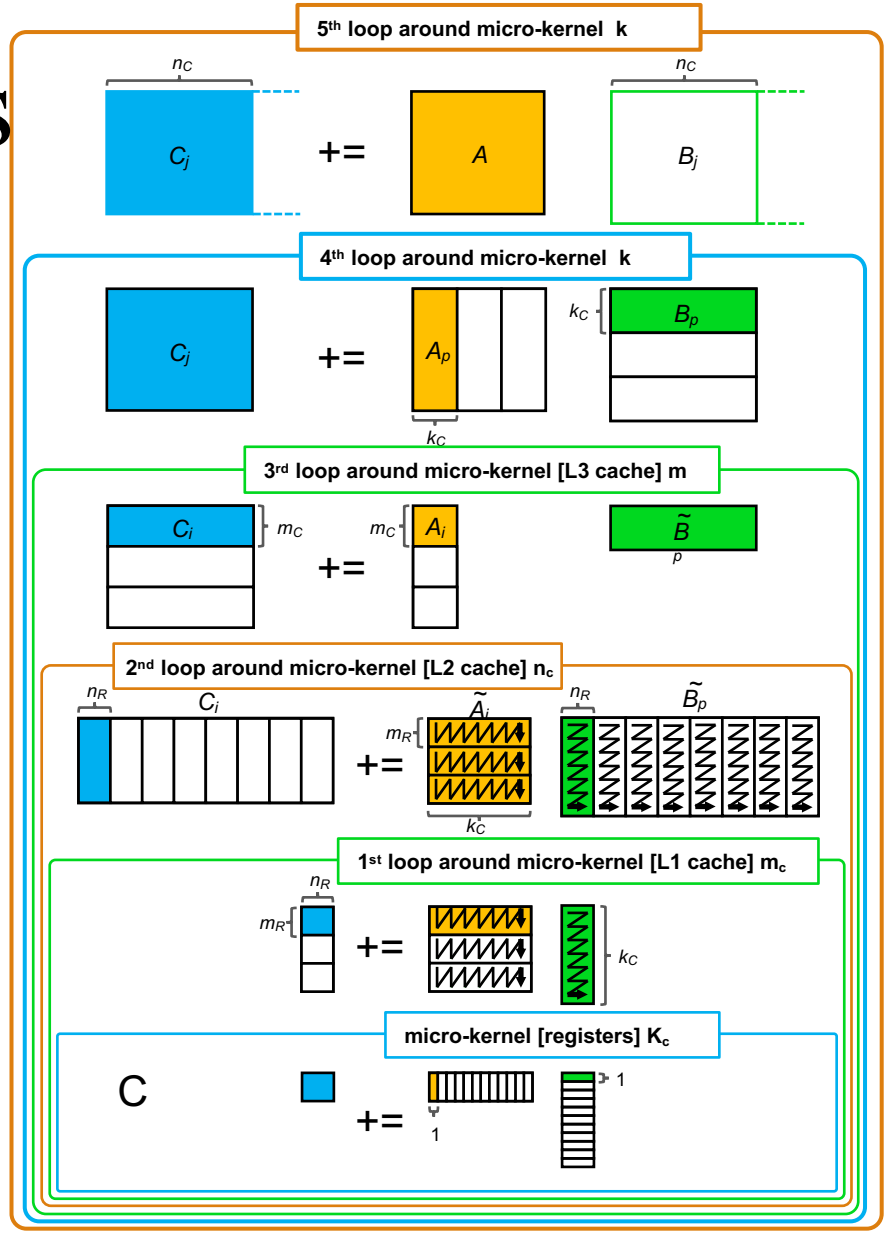
Recap of GEMM Loops



$$
\begin{array}{l}
\textbf{for } j_c = 0, \ldots, n-1 \textbf{ in steps of } n_c \\
\quad \textbf{for } p_c = 0, \ldots, k-1 \textbf{ in steps of } k_c \\
\quad\quad B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c \\
\quad\quad \textbf{for } i_c = 0, \ldots, m-1 \textbf{ in steps of } m_c \\
\quad\quad\quad A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c \\
\hline
\quad\quad\quad \textbf{for } j_r = 0, \ldots, n_c - 1 \textbf{ in steps of } n_r \\
\quad\quad\quad\quad \textbf{for } i_r = 0, \ldots, m_c - 1 \textbf{ in steps of } m_r \\
\hline
\quad\quad\quad\quad\quad \textbf{for } p_r = 0, \ldots, k_c - 1 \textbf{ in steps of } 1 \\
\quad\quad\quad\quad\quad\quad C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\
\quad\quad\quad\quad\quad\quad\quad += A_c(i_r : i_r + m_r - 1, p_r) \\
\quad\quad\quad\quad\quad\quad\quad \cdot\ B_c(p_r, j_r : j_r + n_r - 1)
\end{array}
$$

Recap of GEMM Loops

for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$
      for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
          for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
            $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
              $+= A_c(i_r : i_r + m_r - 1, p_r)$
              $\cdot\ B_c(p_r, j_r : j_r + n_r - 1)$



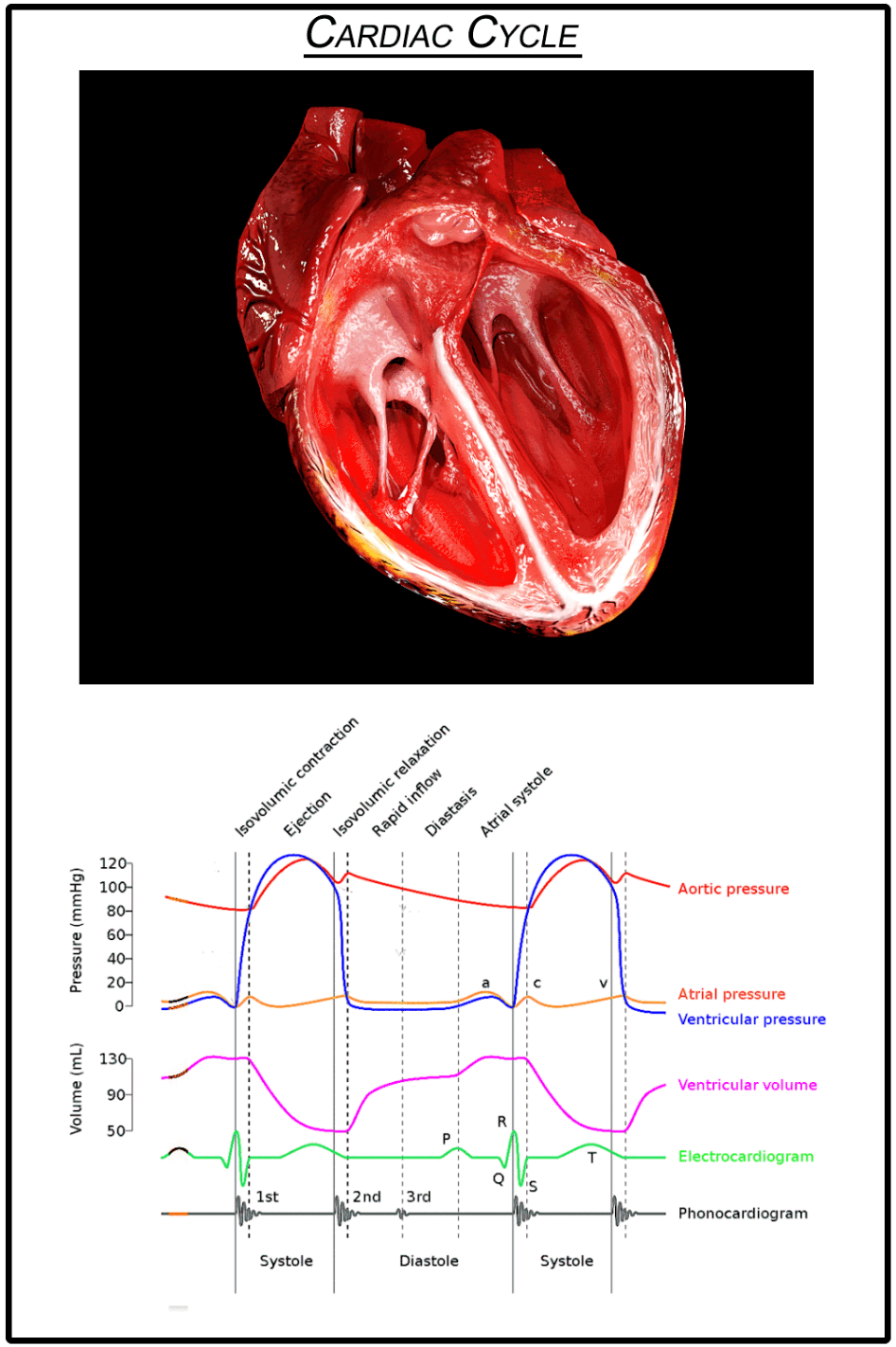
Design Hardware for GEMM

- Systolic Arrays
  - Systolic Arrays
  - GEMM Systolic solution

- Others
  - Other parallel GMEM algorithms

Systolic Arrays

Systolic Arrays offer a way to take certain exponential algorithms and use hardware to make them linear. They are **expensive and complex** but yield enormous throughput.



Definitions of Systolic Array

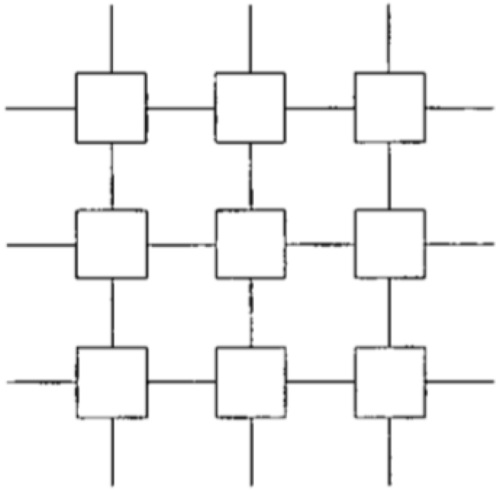
- "Imagine n simple processors arranged in a row or an array and connected in such a manner that each processor may exchange each processor may exchange information with **only its neighbors** to the right and left. The processors at either end of the row are used for input and output. Such a machine constitutes the simplest example of a systolic array."[1]

- [1] Bayoumi, Magdy. Ling, Nam. Specification and Verification of Systolic Arrays. World Scientific Publishing Co. Pte. Ltd. Singapore. 1999.
- [2] Brown, Andrew. VLSI Circuits and Systems in Silicon. McGraw-Hill Book Company. London. 1991.
- [3] Dewdney, A.K. The (New) Turing Omnibus. Henry Holt and Company. New York. 1993.

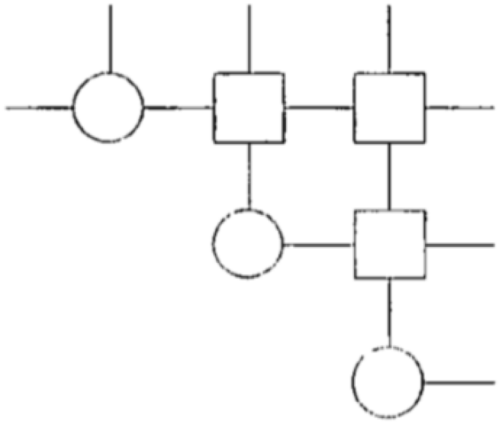
- "Systolic Arrays are **regular arrays** of simple finite state machines, where each finite state machine in the array is identical...A systolic algorithm relies on data from different directions arriving at cells in the array at regular intervals and being combined." [2]
- By **pipelining, processing may proceed concurrently** with input and output, and consequently overall execution time is minimized. Pipelining plus multiprocessing at each stage of a pipeline should lead to the best-possible performance."[3]

What do they look like?

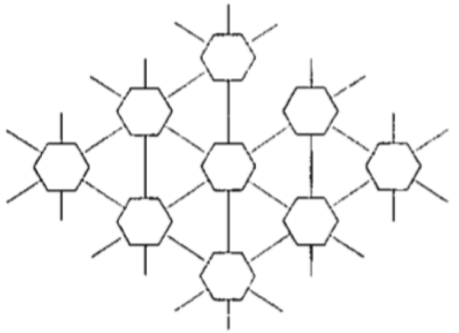
Linear Array



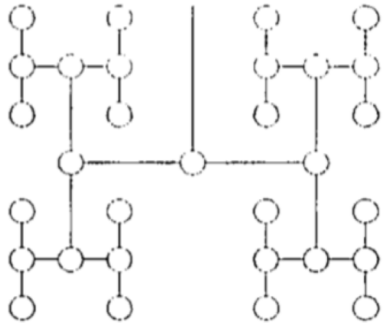
Orthogonal Array



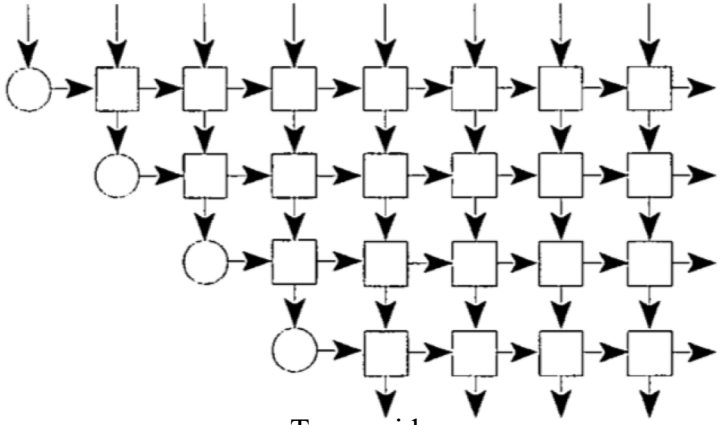
Triangular Array



Hexagonal Array



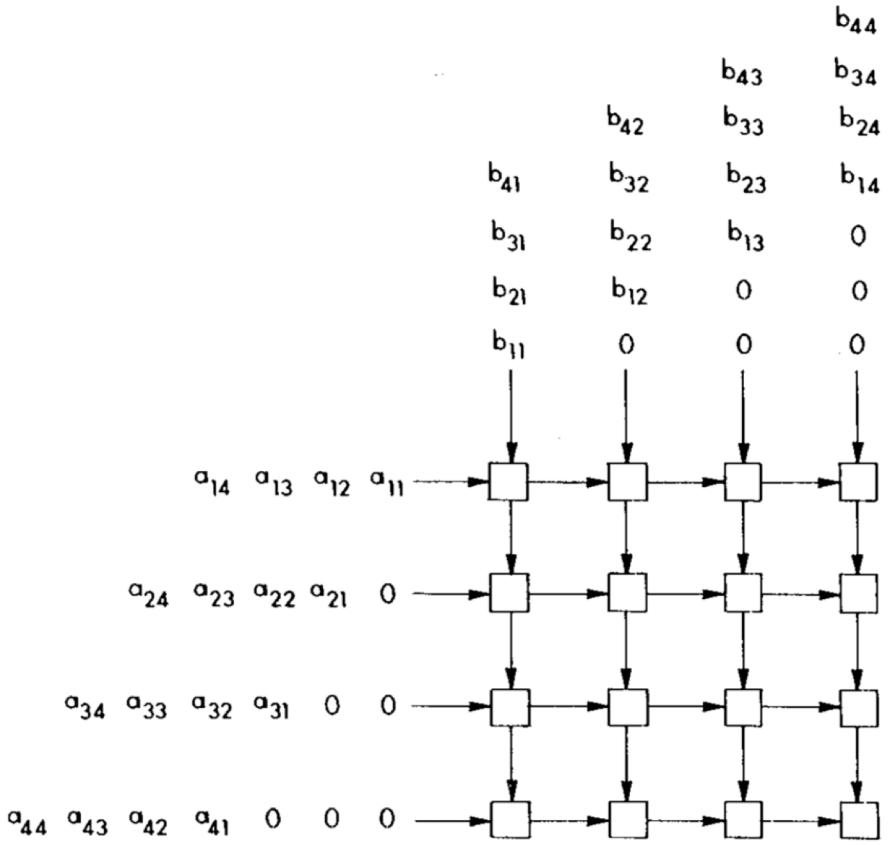
Binary Tree

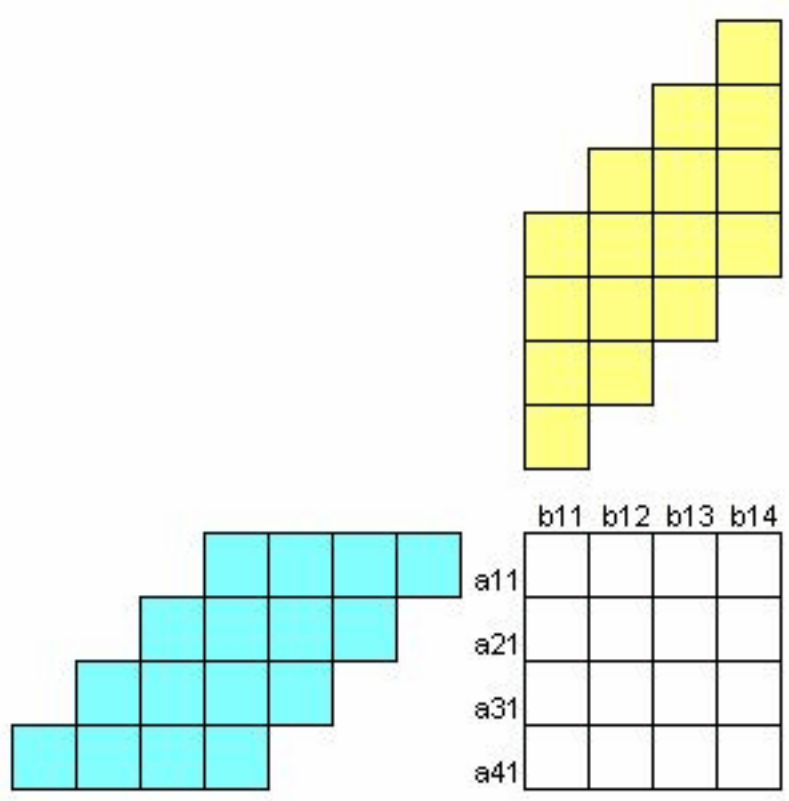


Trapezoid

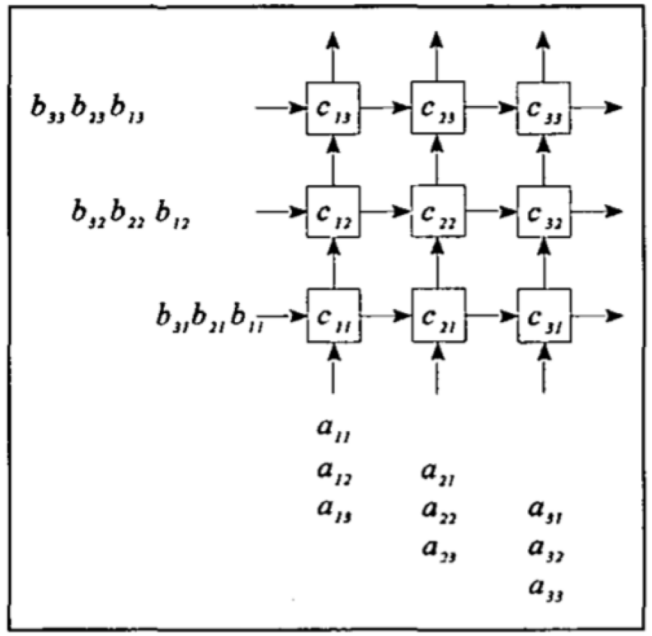
Source Chunru Wan

Systolic Matrix Multiplication

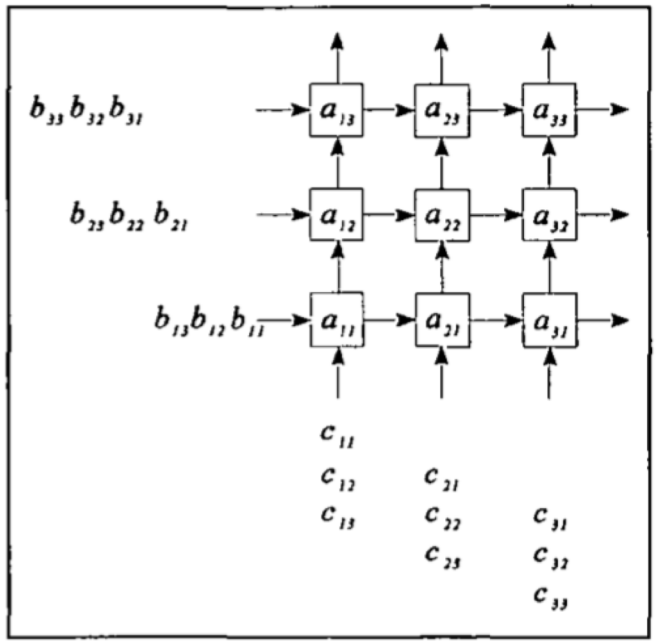




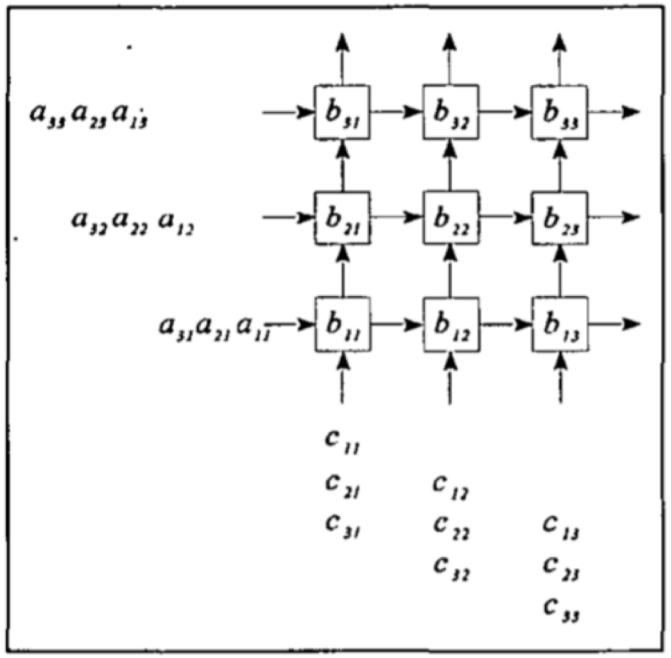
Are There Other Ways?



- Keep C resident
- Stream A,B



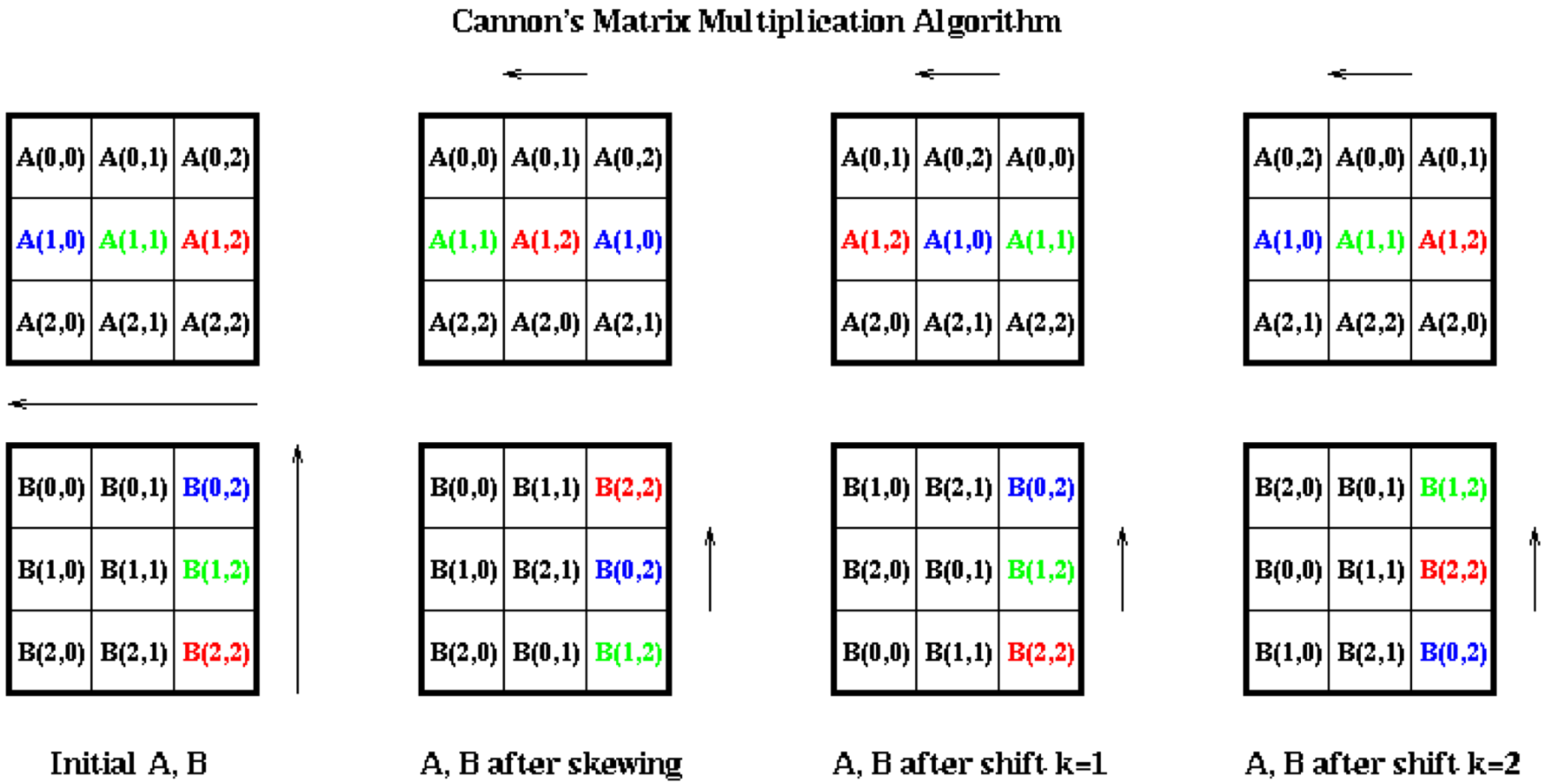
- Keep A resident
- Stream B,C



- Keep B resident
- Stream A,C

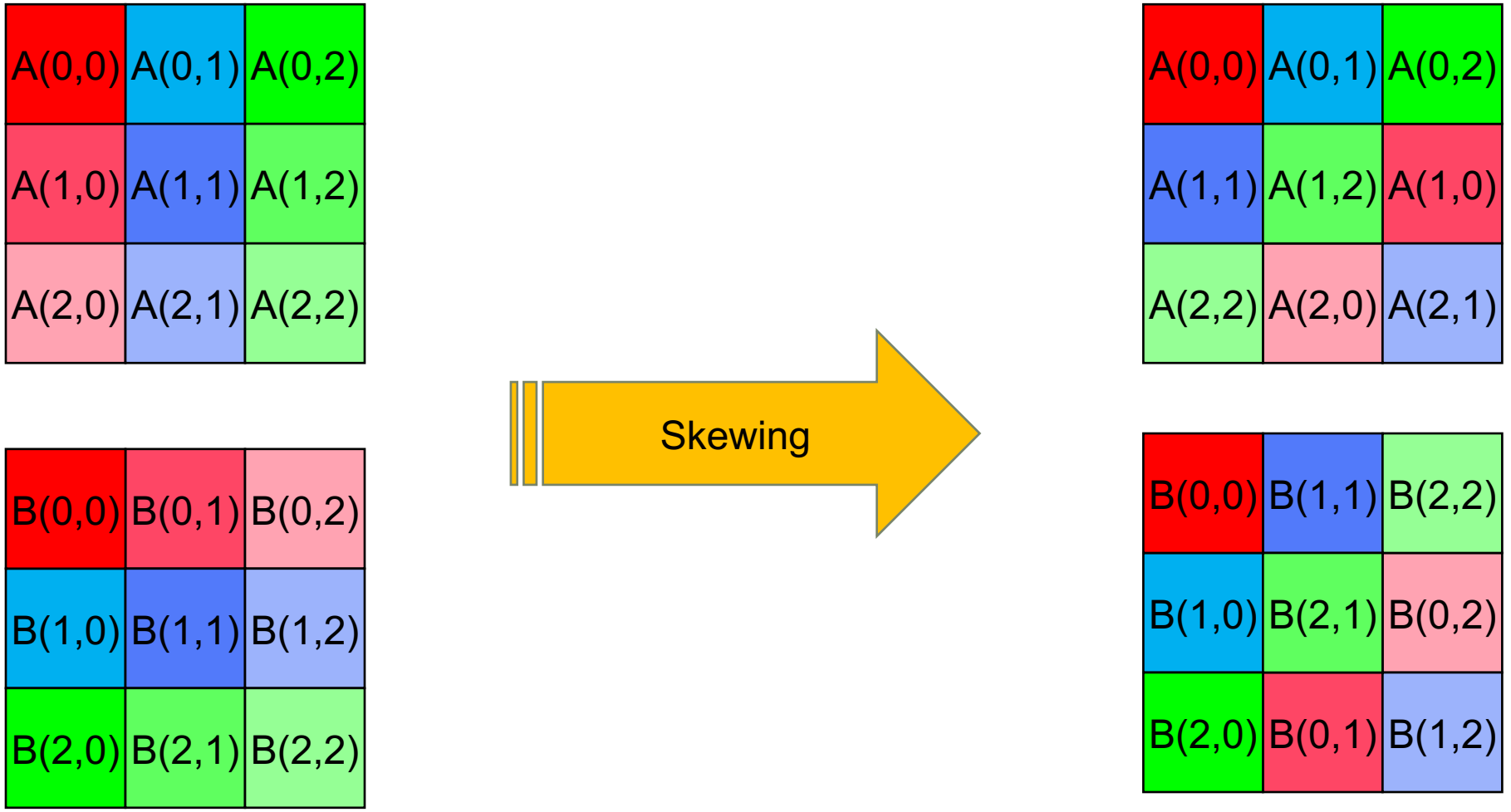
Source Chunru Wan

Cannon's Matrix Multiplication

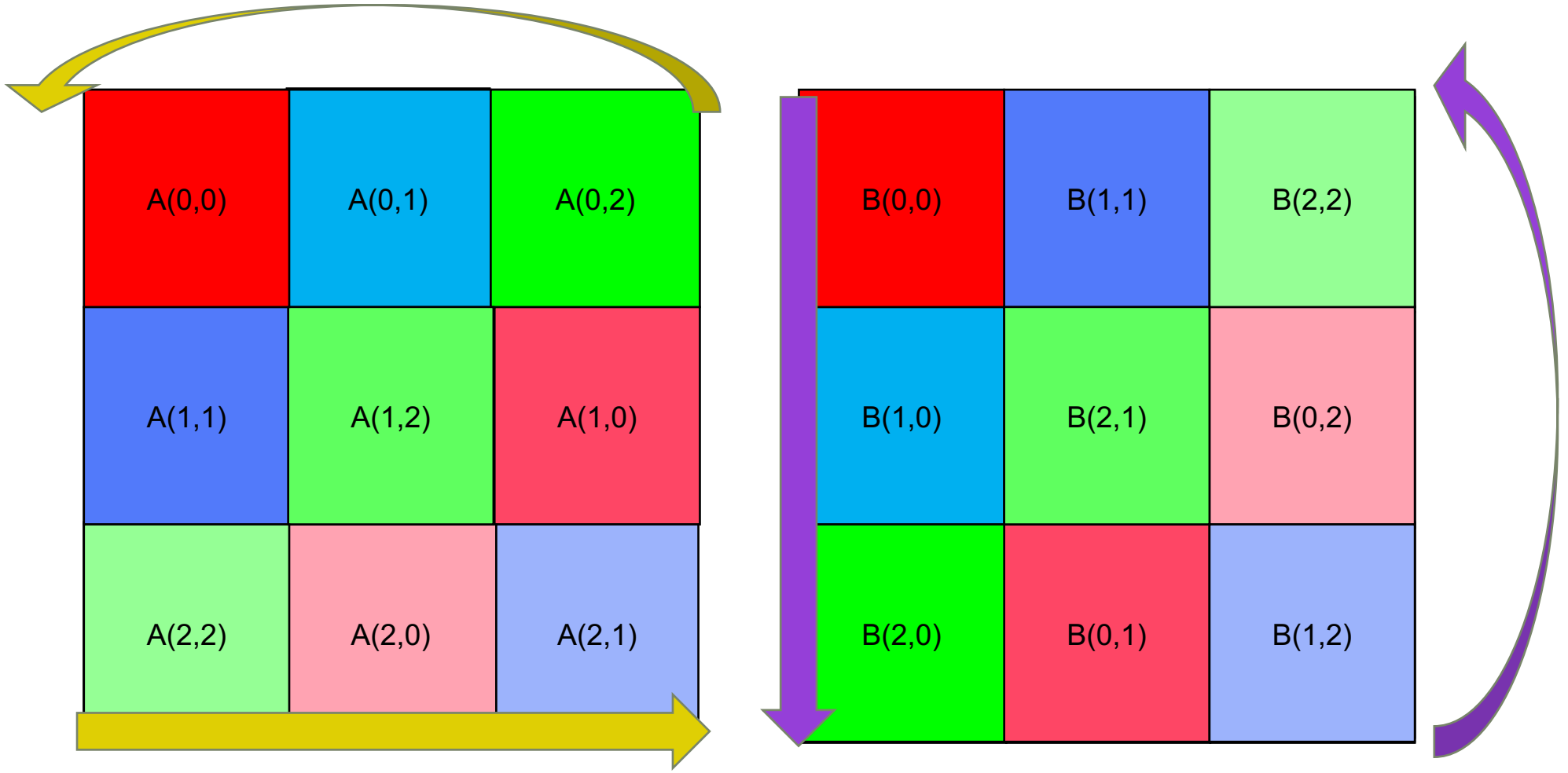


C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

Initial Step to Skew Matrices in Cannon



Skewing Steps in Cannon



All blocks of A must multiply all like-colored blocks of B

Systolic GEMM Google TPU1

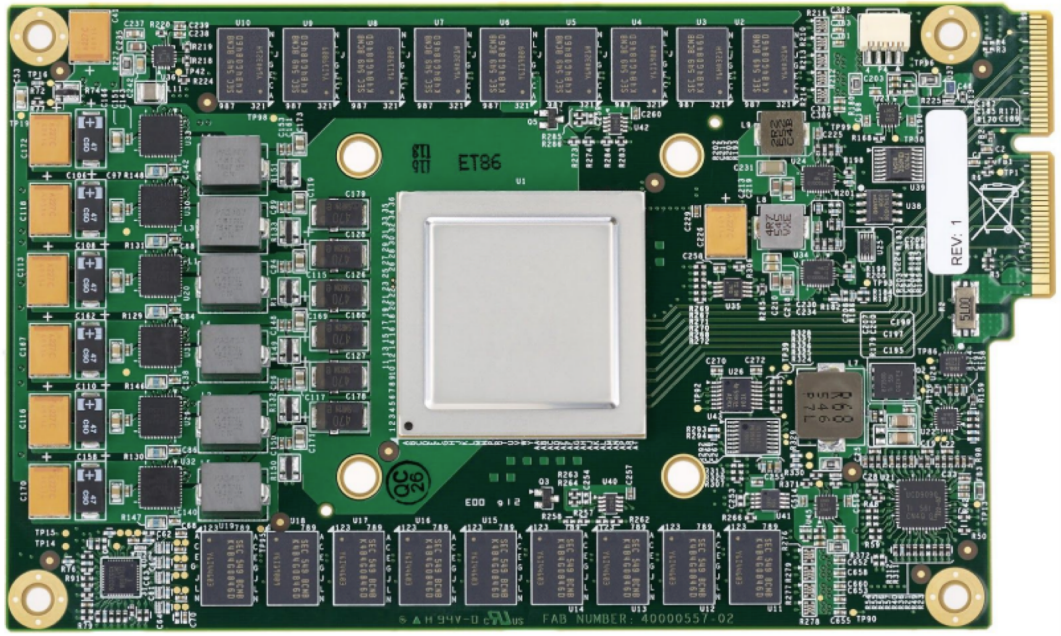


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

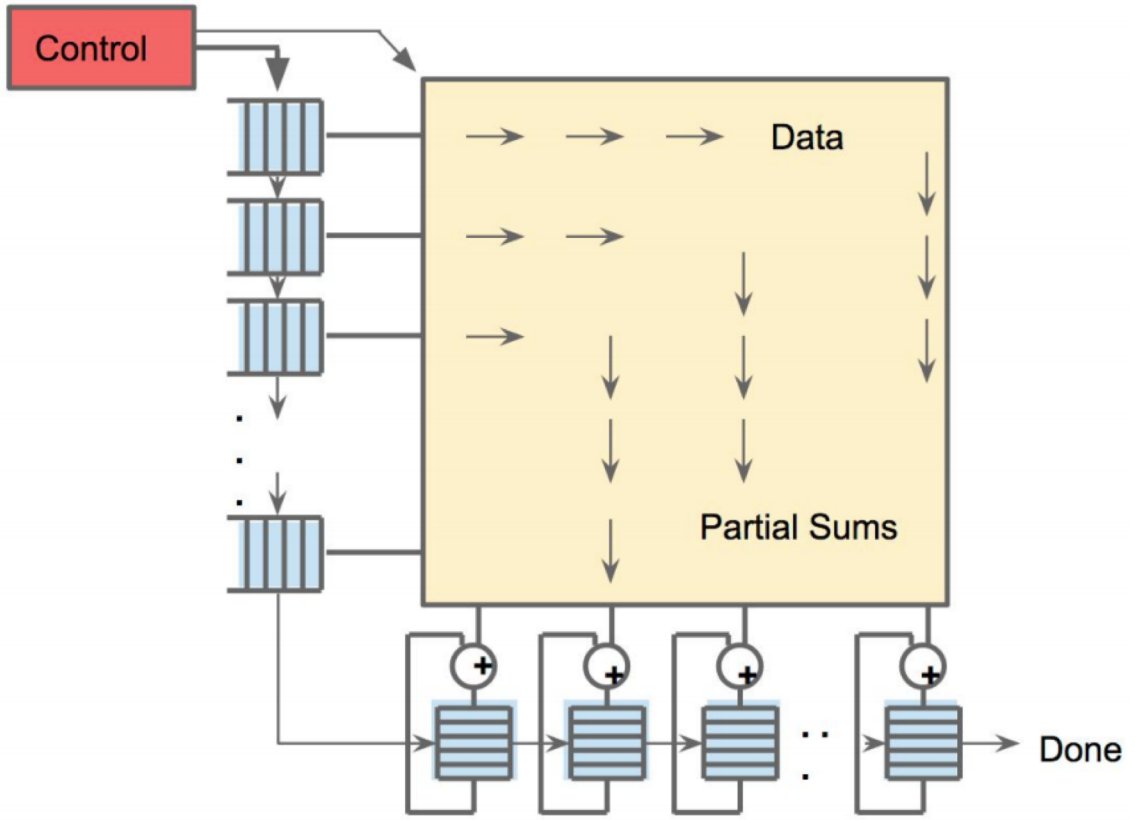
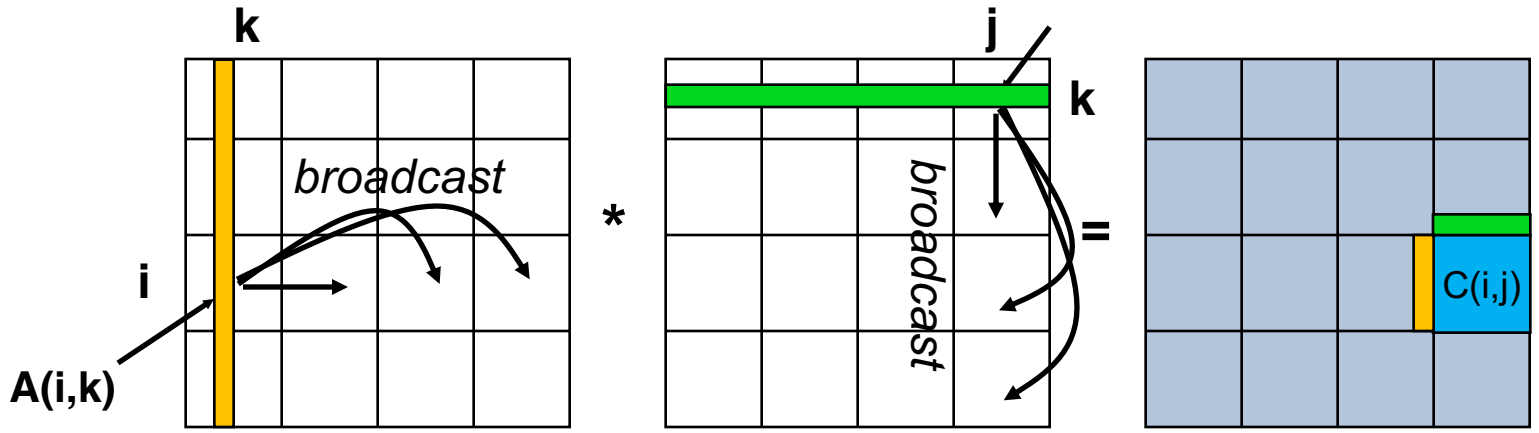


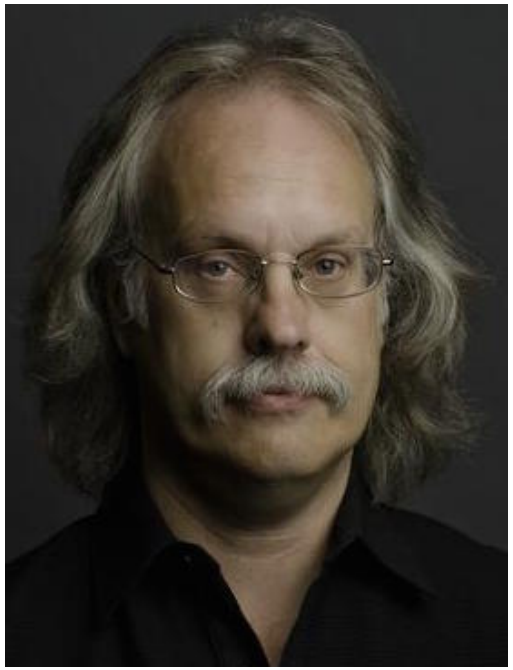
Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Other Algorithms

- Cannon's algorithms (roll-roll-multiply)
- Fox's algorithm (broadcast-roll-multiply)
- SUMMA (broadcast-broadcast-multiply)

SUMMA Algorithm





For k=0 to n/b-1 … where b is the block size

… b = # cols in A(i,k) and # rows in B(k,j)

for all i = 1 to p_r … *in parallel*

owner of A(i,k) broadcasts it to whole processor row

for all j = 1 to p_c … *in parallel*

owner of B(k,j) broadcasts it to whole processor column

Receive A(i,k) into Acol

Receive B(k,j) into Brow

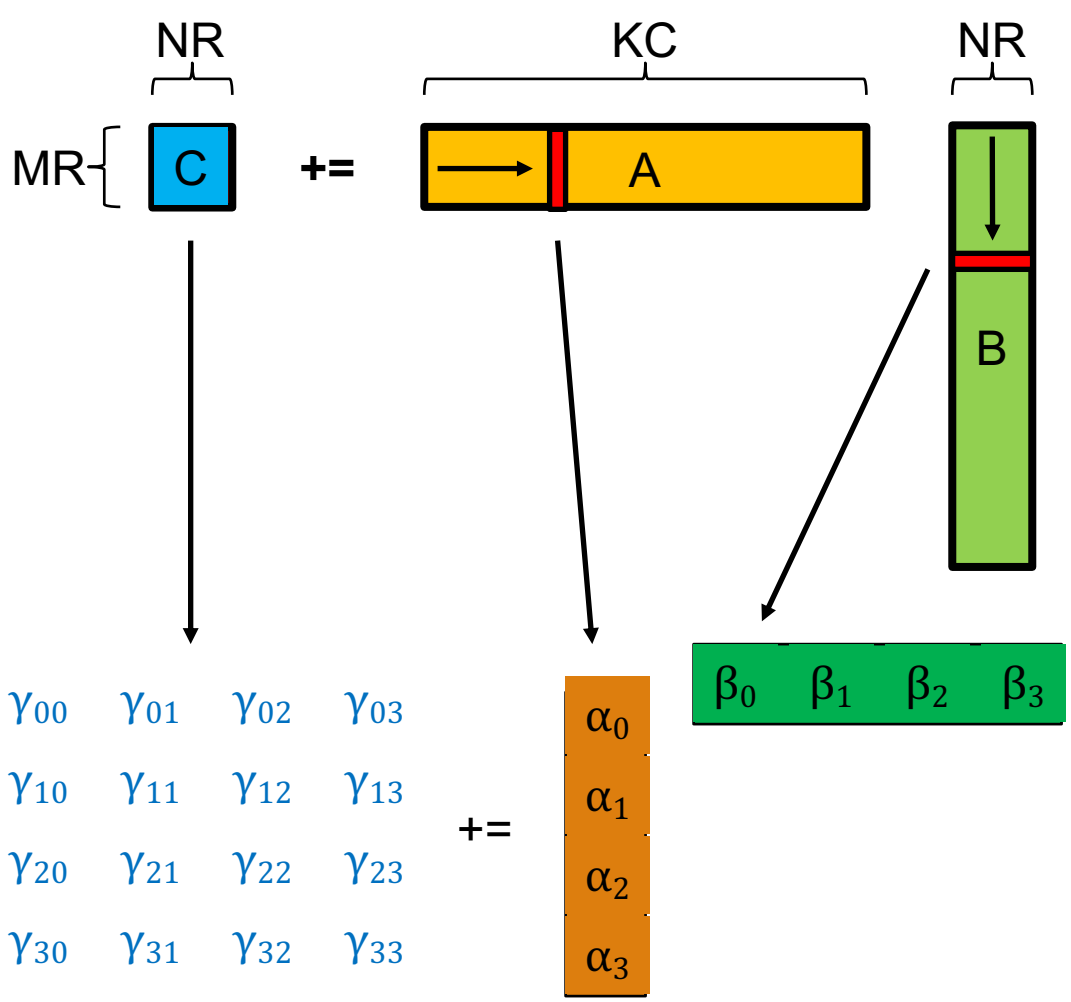
C_myproc = C_myproc + Acol * Brow

Source James Demmel

GEMM Accelerator

- What If we wanted to design a GEMM accelerator?
- What Algorithm to pick?
  - Systolic
  - Canon
  - SUMMA
- What does the architecture look like?
- What was the GEMM micro kernel?
- What is the rank-1 update?

The GEMM micro-kernel

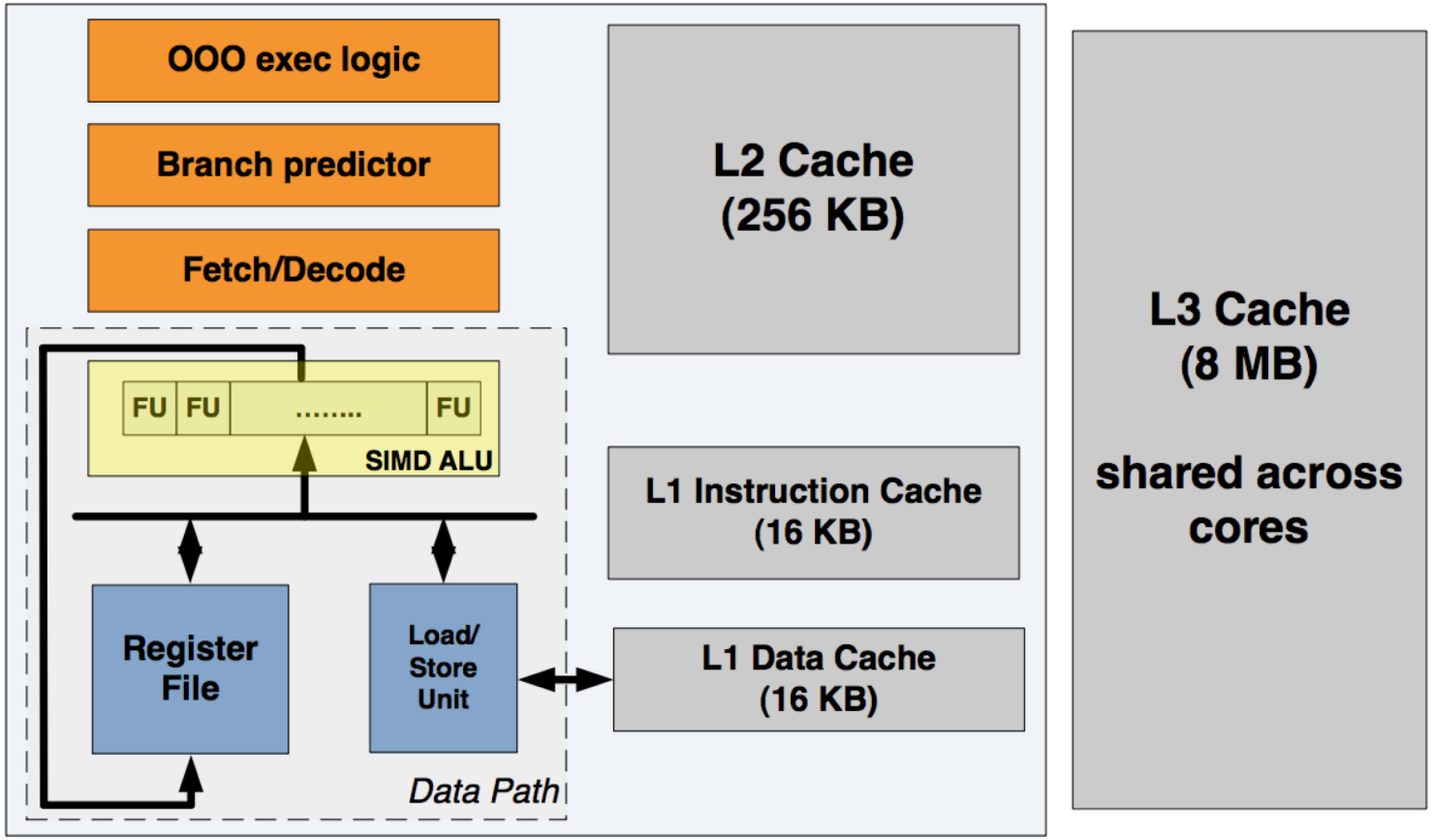



```
for ( 0 to NC: NR )
  for ( 0 to MC: MR )
    for ( 0 to KC: 1 )
      // block-dot product
    endfor
  endfor
endfor
```

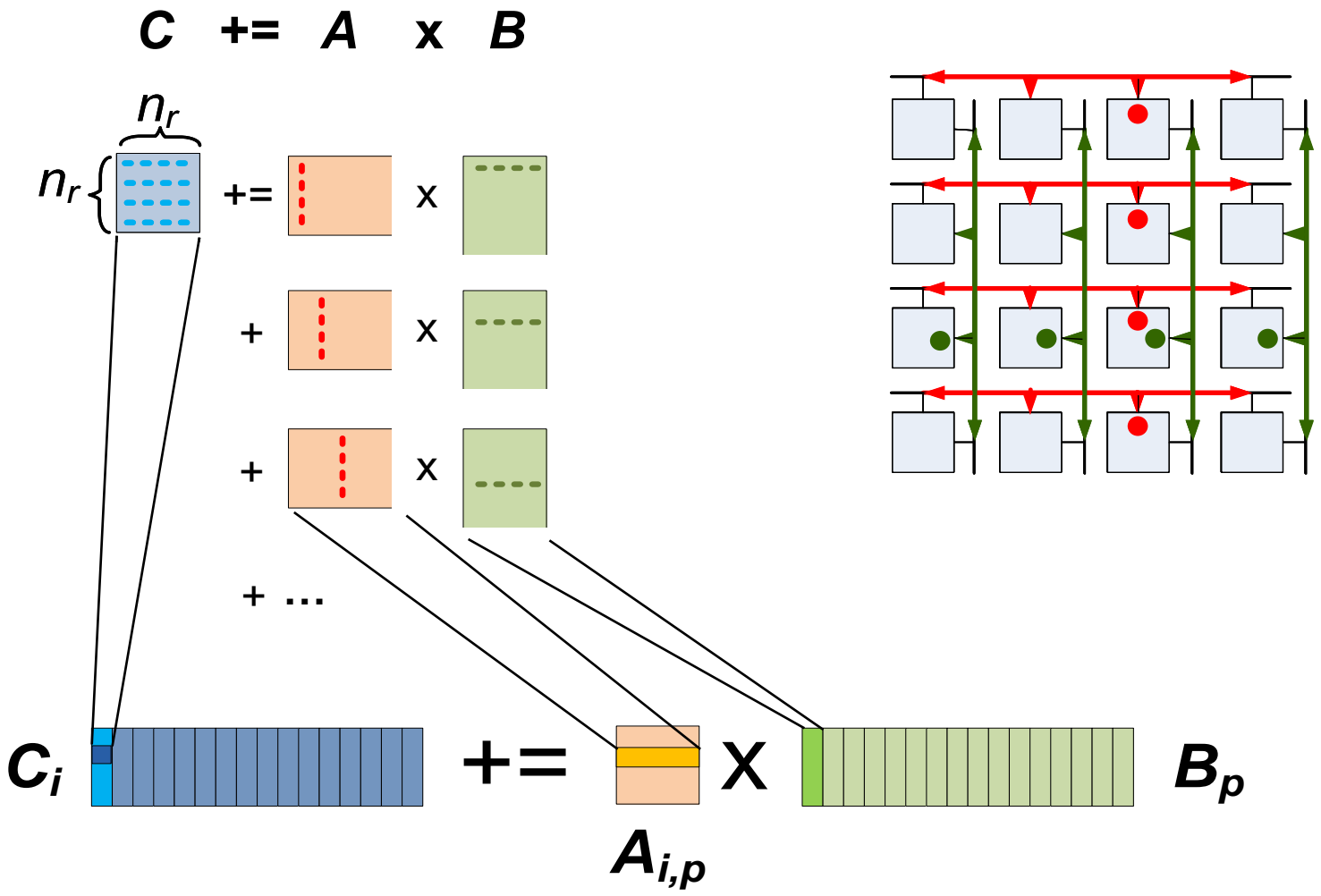
- Typical micro-kernel loop iteration ("block-dot product")
  - Load column of packed A
  - Load row of packed B
  - Compute outer product
  - Update C (kept in registers)

Custom vs. General Purpose Design

- What do we need ?
- What don't we need?
- Caches?
- Register file?
- ALUs?
- OoO execution?
- Branch predictor?
- Fetch/Decode Unit

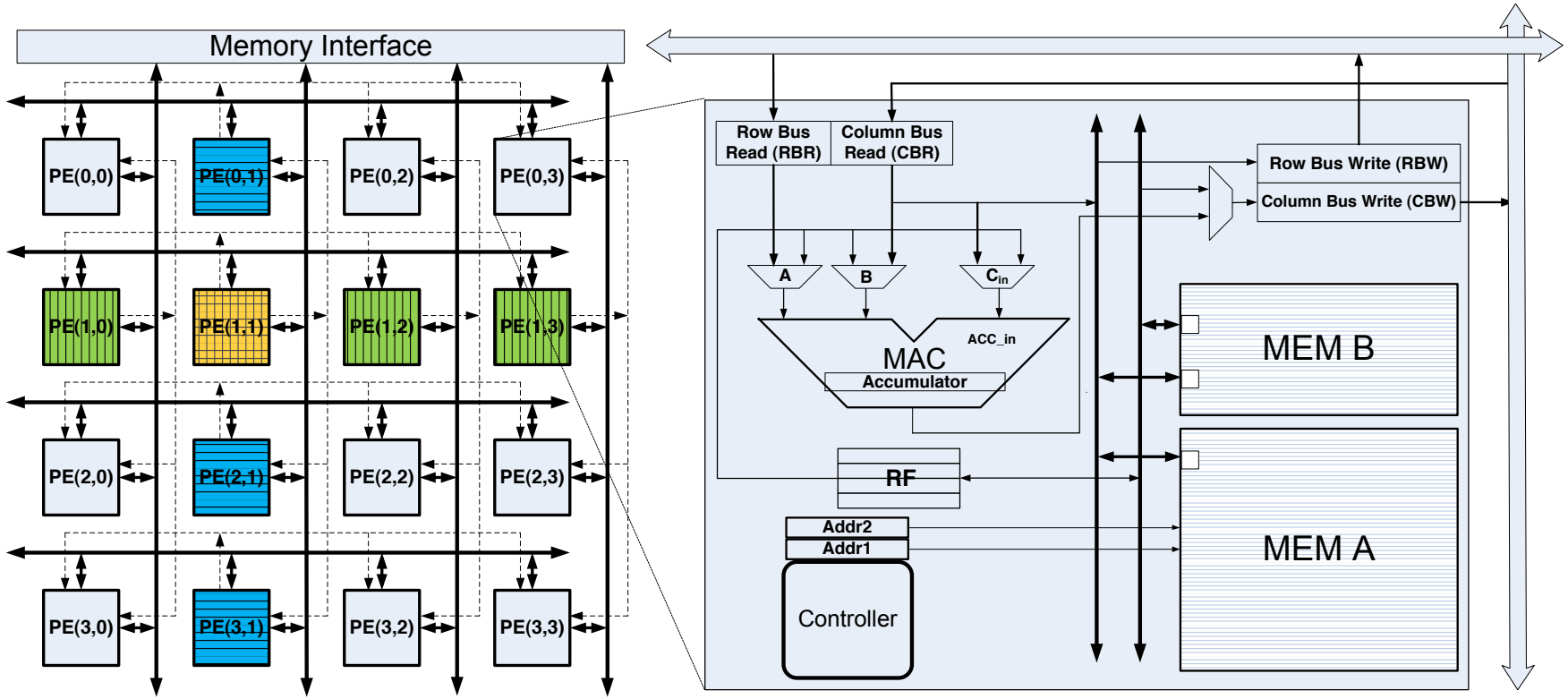


The "Rank-1" Machine: Linear Algebra Core



[Pedram et al. 2011]

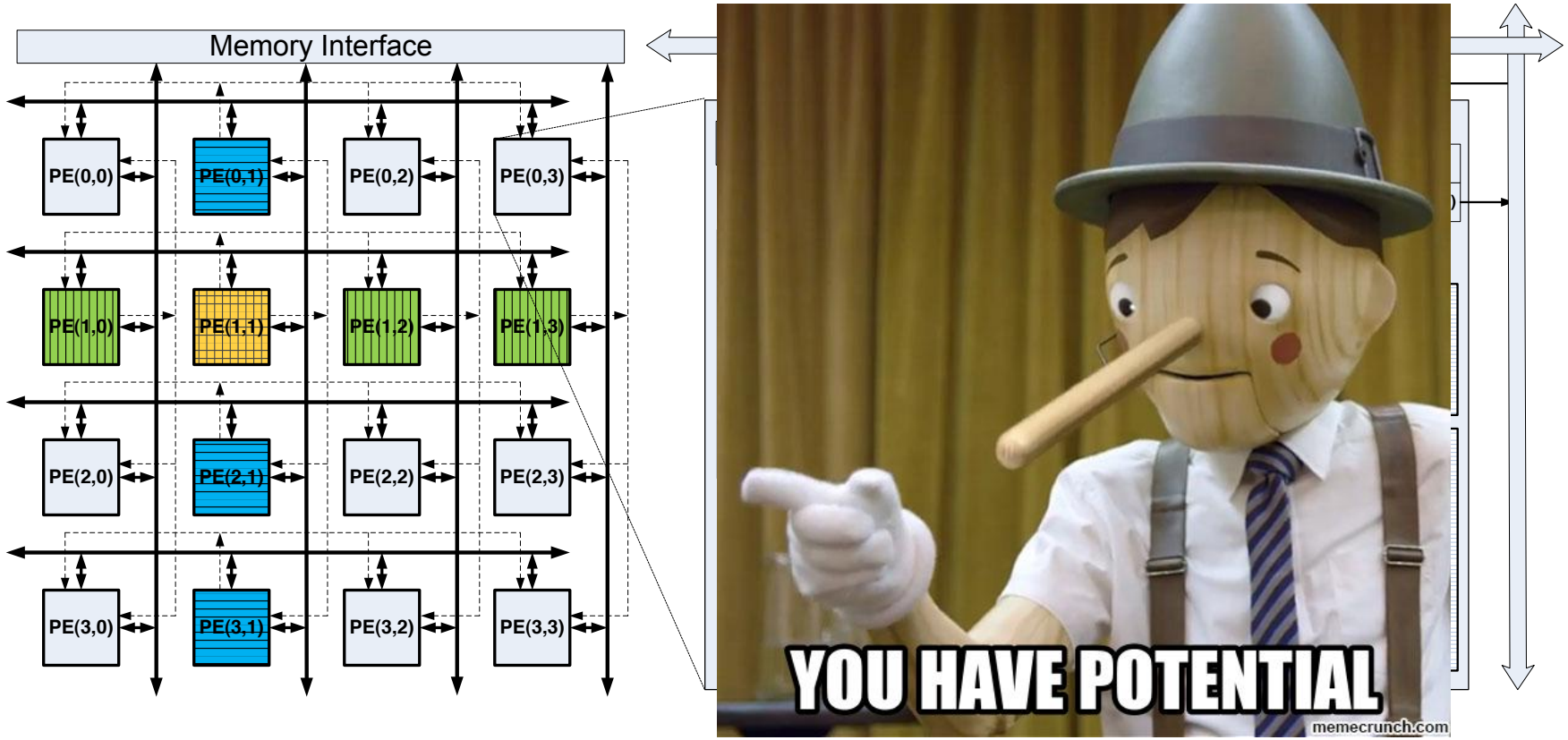
RANK-1 Hardware



- Scalable 2-D array of $n_r \times n_r$ processing elements (PEs)
  - Specialized floating-point units with 1 MAC/cycle throughput
  - Broadcast busses (no need to pipeline up to n_r=16)
  - Distributed memory architecture
  - Distributed, PE-local control

[Pedram et al. 2012]

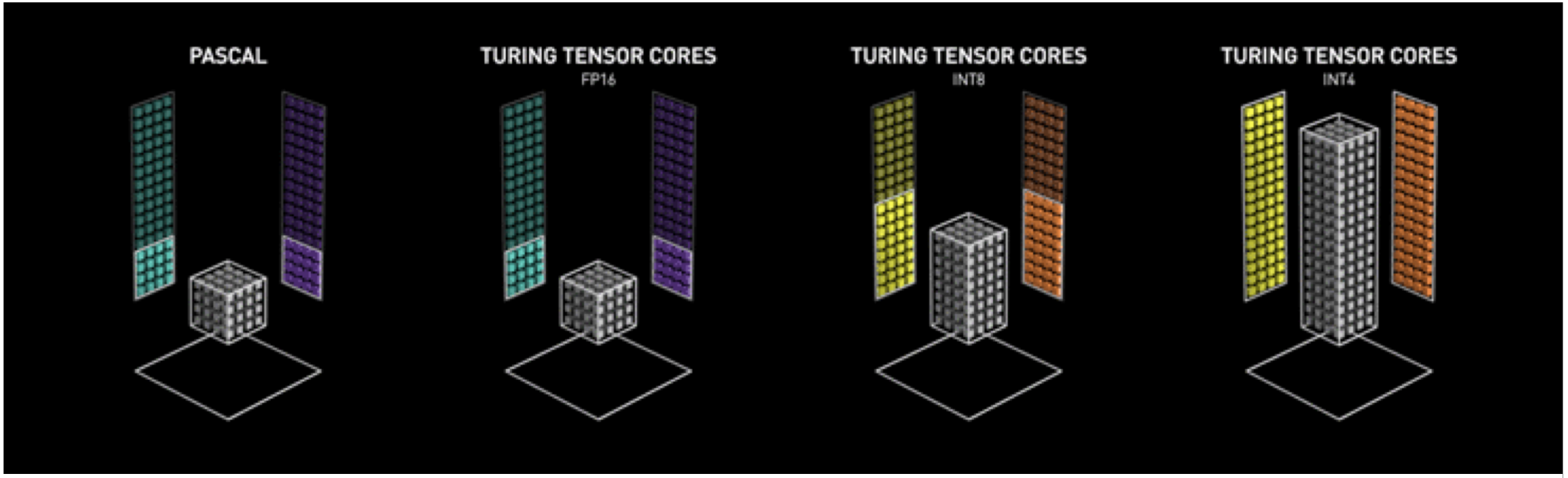
RANK-1 Hardware



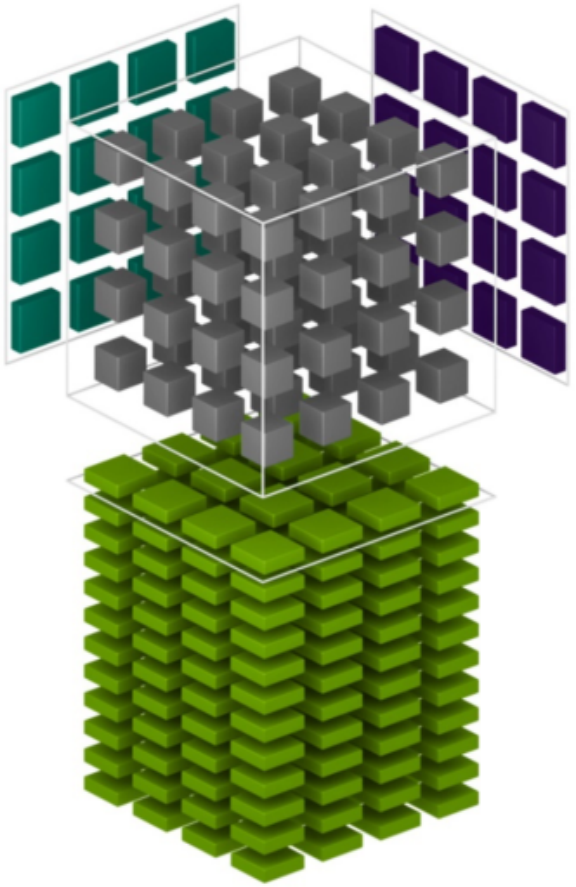
- Scalable 2-D array of $n_r \times n_r$ processing elements (PEs)
  - Specialized floating-point units with 1 MAC/cycle throughput
  - Broadcast busses (no need to pipeline up to n_r=16)
  - Distributed memory architecture
  - Distributed, PE-local control

[Pedram et al. 2012]

Tensor Core Unit of Nvidia



Tensor Core Unit of Nvidia

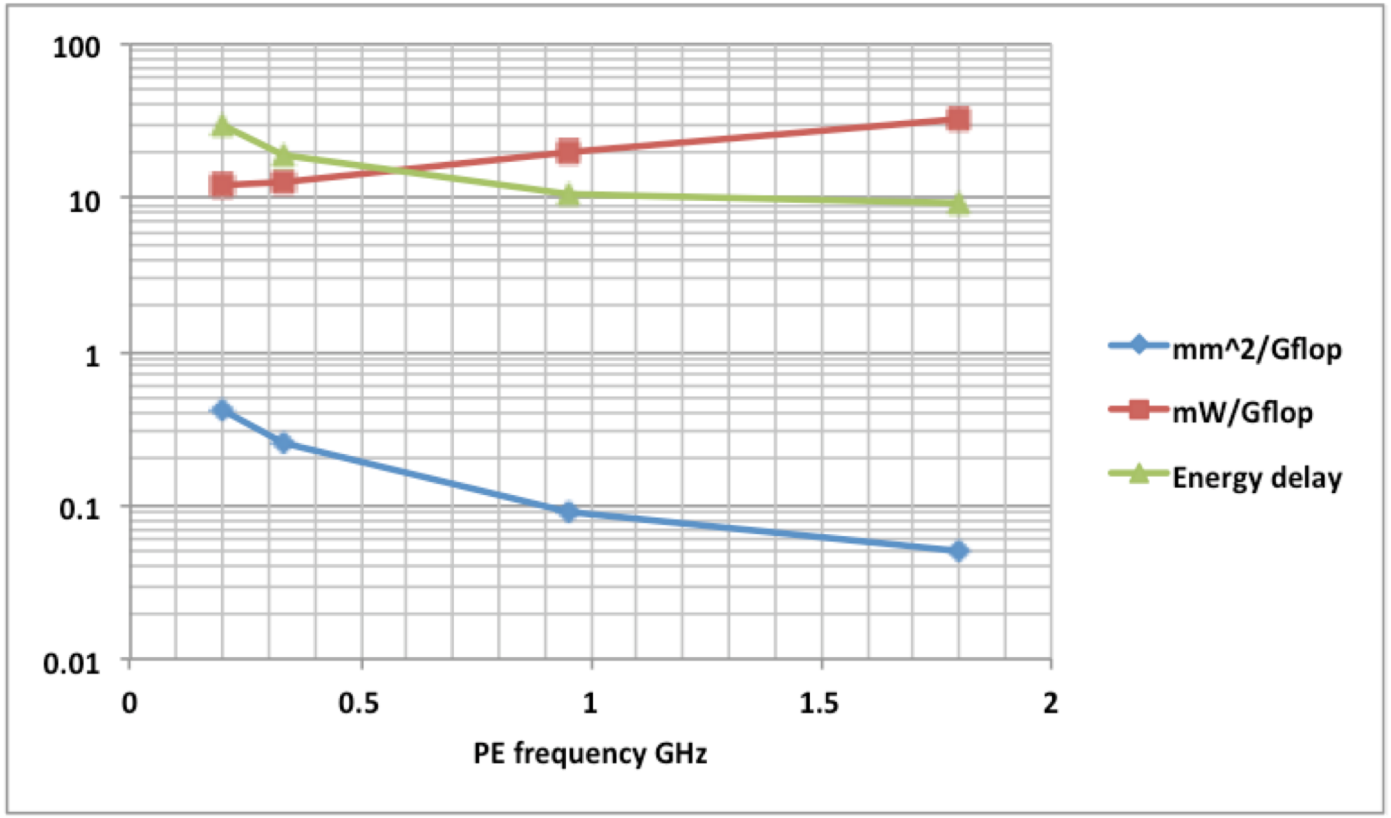


Recap: Math vs. Memory

Operation	16 bit (integer)		64 bit (DP-FP)	
	E/op PJ	vs. Add	E/op PJ	vs. Add
ADD	0.18	1.0 ×	5	1.0 ×
Multiply	0.62	3.4 ×	20	4.0 ×
16-Word Register File	0.12	0.7 ×	0.34	0.07 ×
64-Word Register File	0.23	1.3 ×	0.42	0.08 ×
4 K-word SRAM	8	44 ×	26	5.2 ×
32 K-word SRAM	11	61 ×	47	9.4 ×
DRAM	**640**	**3556×**	**2560**	**512 ×**

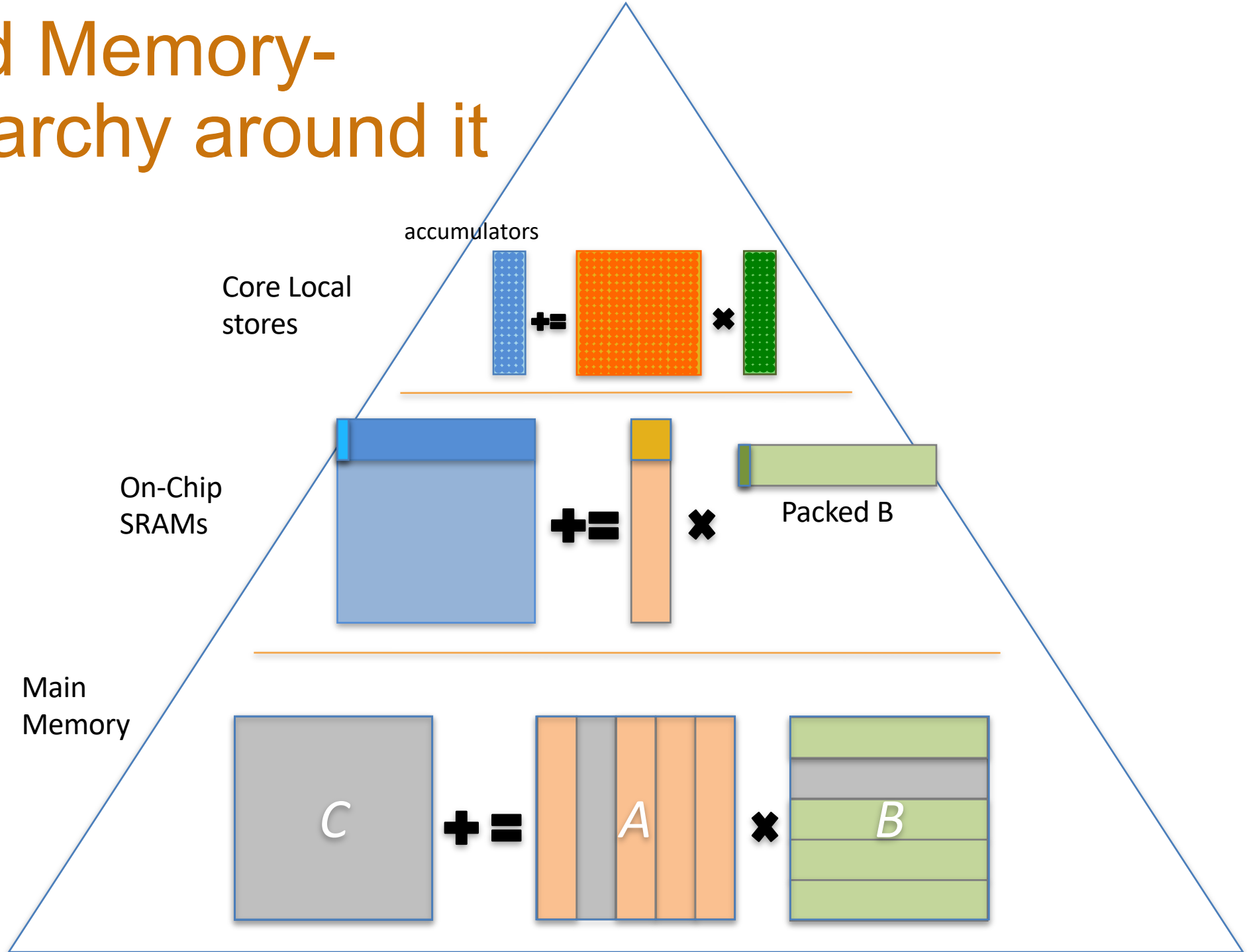
PE-Level Implementation @45nm

- Area
  - SRAM Dominates
- Power
  - MAC Dominates
- Double-precision
  - 60 GFLOPS/W upper limit

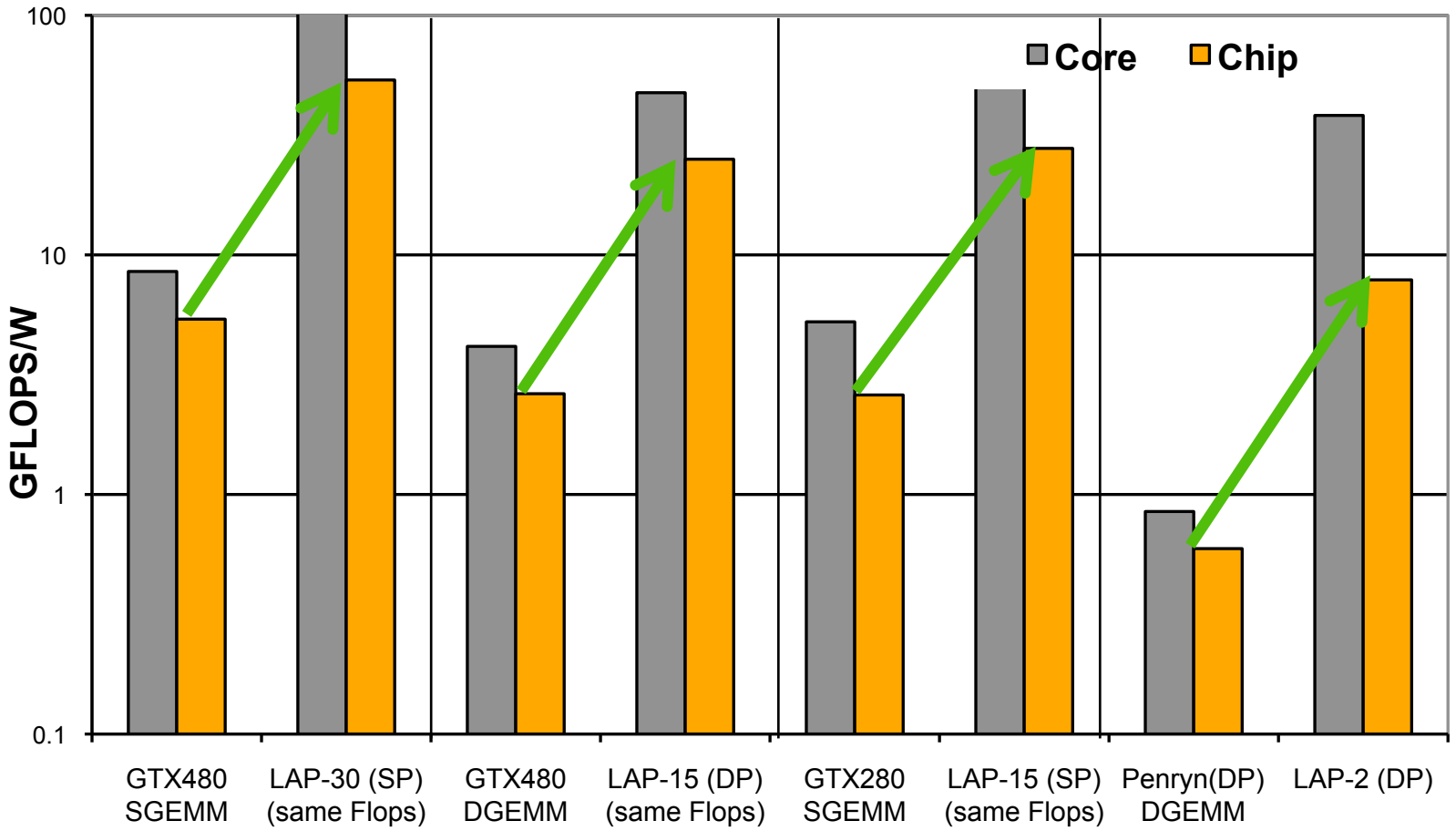


➢ 1 GHz sweet spot of performance vs. efficiency

Build Memory-Hierarchy around it

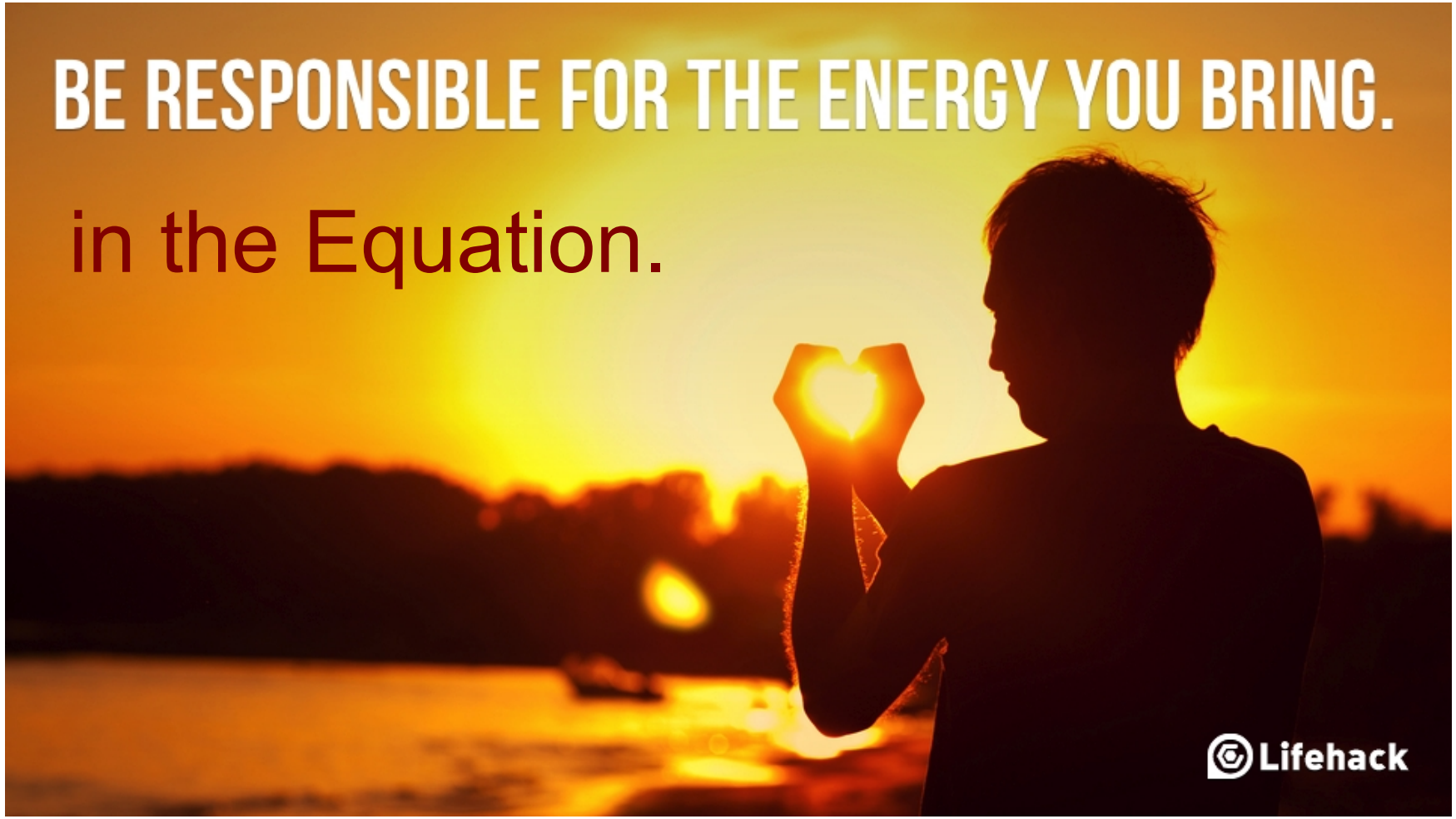


GEMM Accelerator

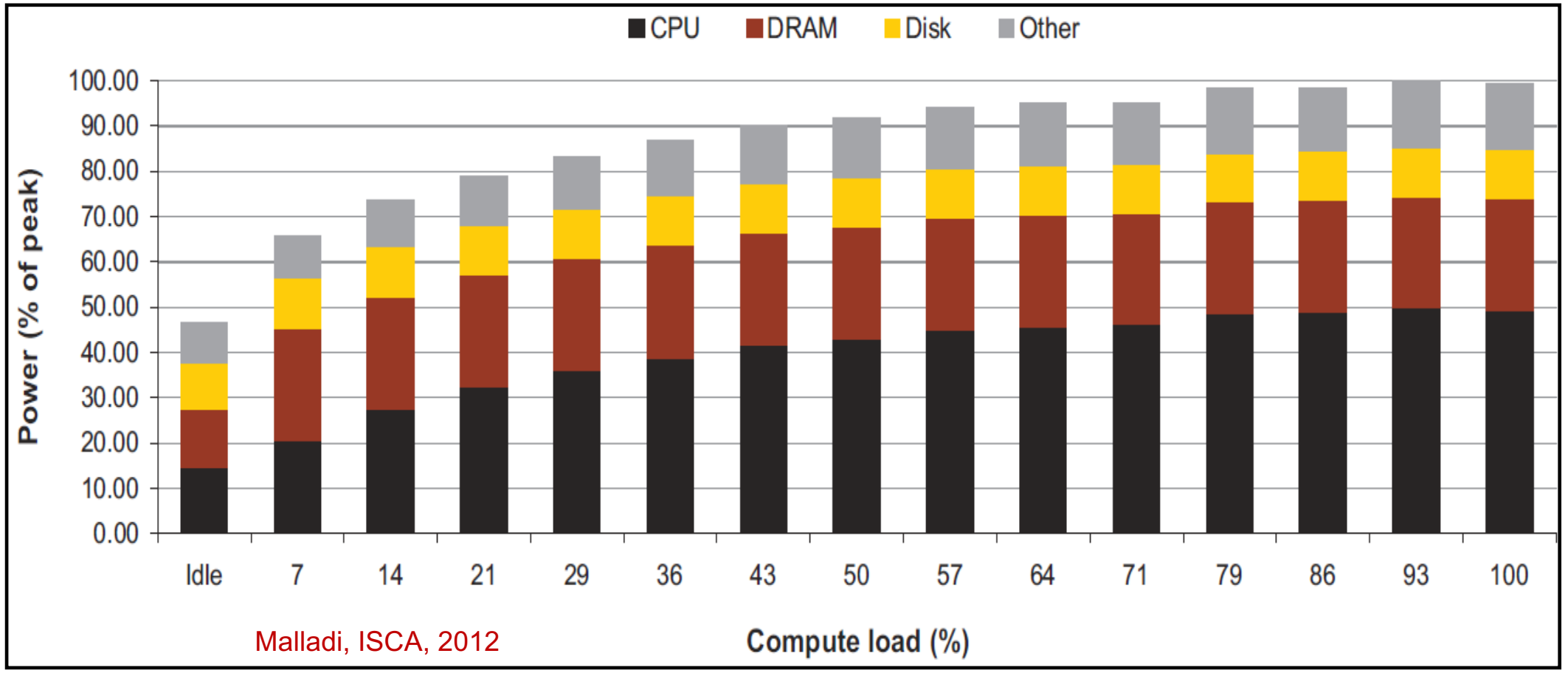


- ***Orders of Magnitude*** Better Energy Efficiency ?

Energy

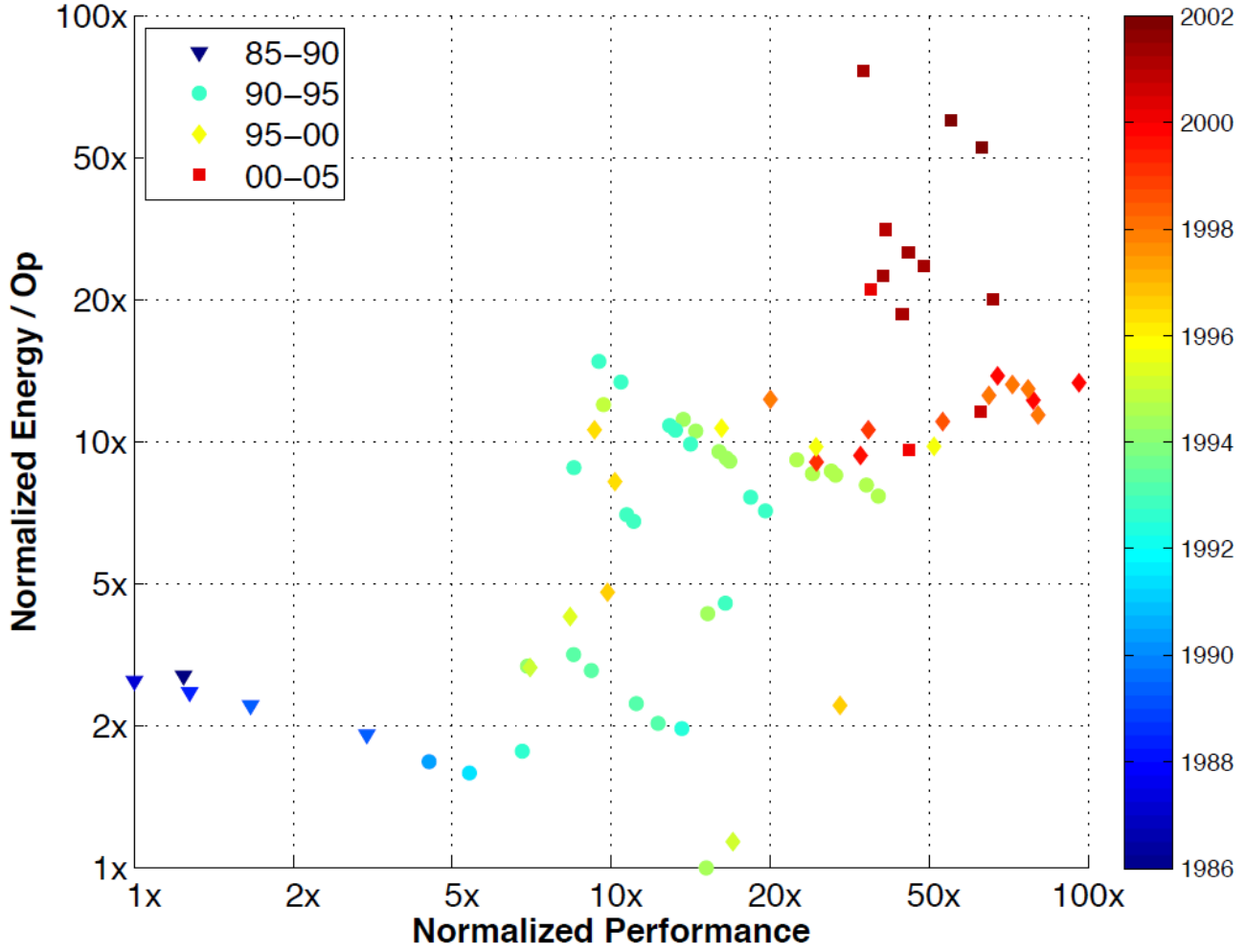

BE RESPONSIBLE FOR THE ENERGY YOU BRING.
in the Equation.
Lifehack

Example 1: Don't Forget Memory System Energy

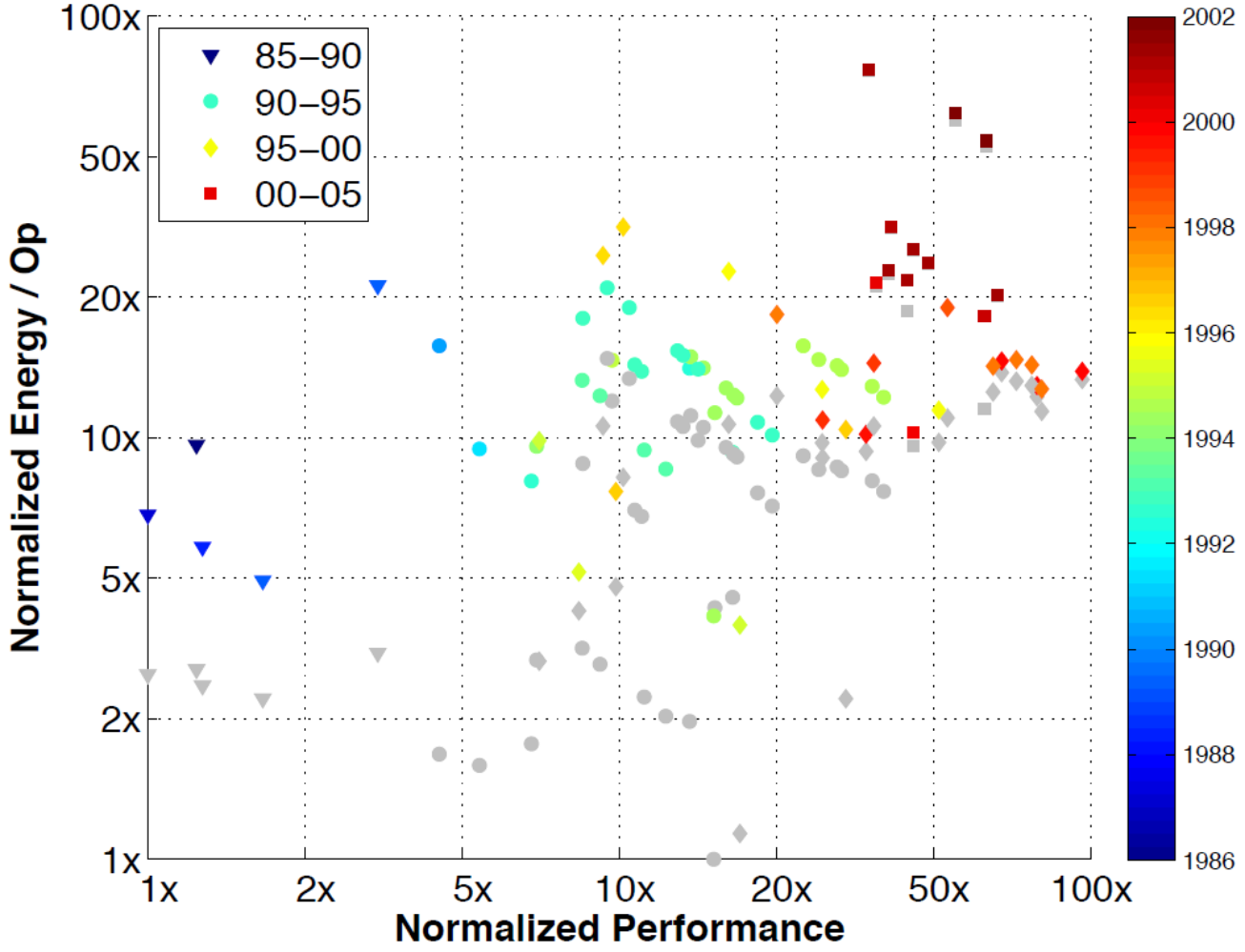


Malladi, ISCA, 2012

Example 2: Don't Forget Cache Energy

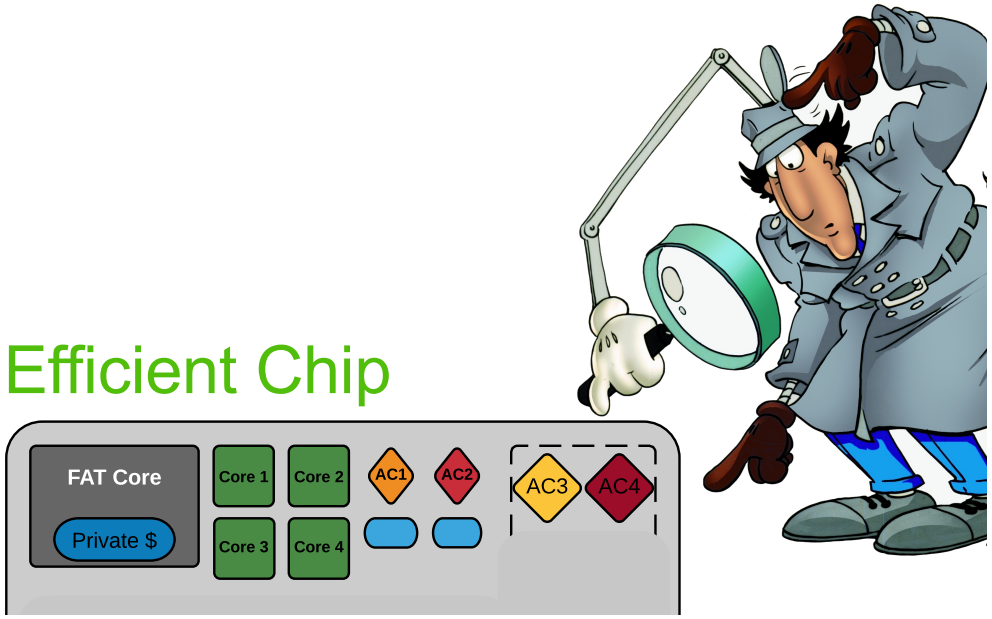


Example 2:
Energy with Corrected Cache Size



Dark Silicon View

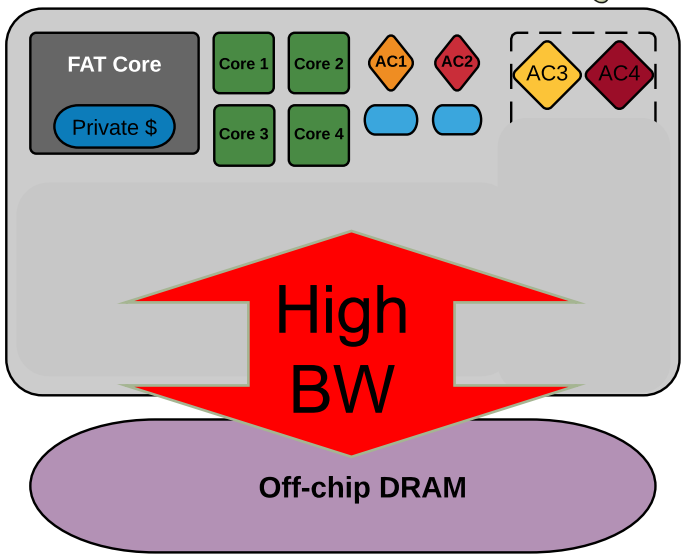
- Efficient Accelerators
- Increase Chip's
  - Energy Efficiency
  - Area Efficiency
- Report 10~1000x improvements



Real Perspectives

- Inefficient System
- Increase ~~Chip's~~
  - Energy Efficiency
  - Area Efficiency
- Report ~~10~1000x~~ improvements

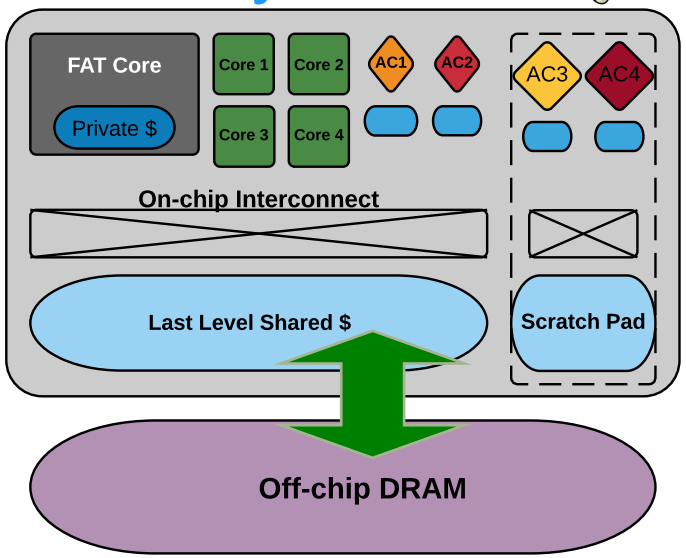
Inefficient System
Efficient Chip



Dark Memory Perspectives

- Efficient System
- On-chip buffers
  - Less efficient chip
- Lower Off-chip Memory Bandwidth

Efficient System
Locality

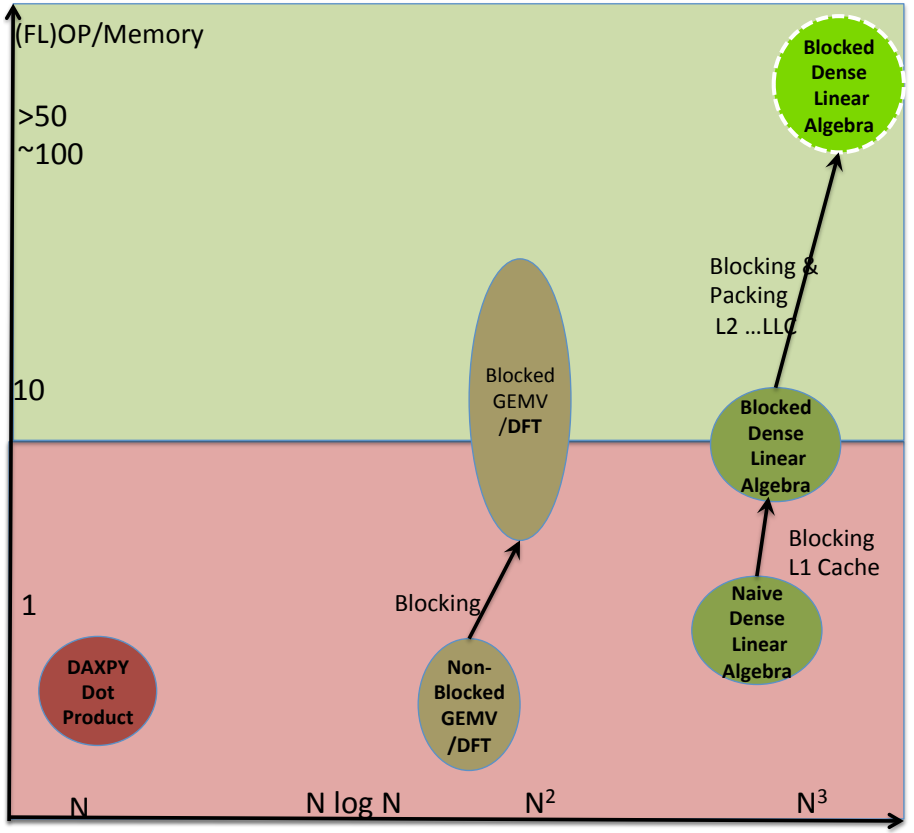


DARK MEMORY

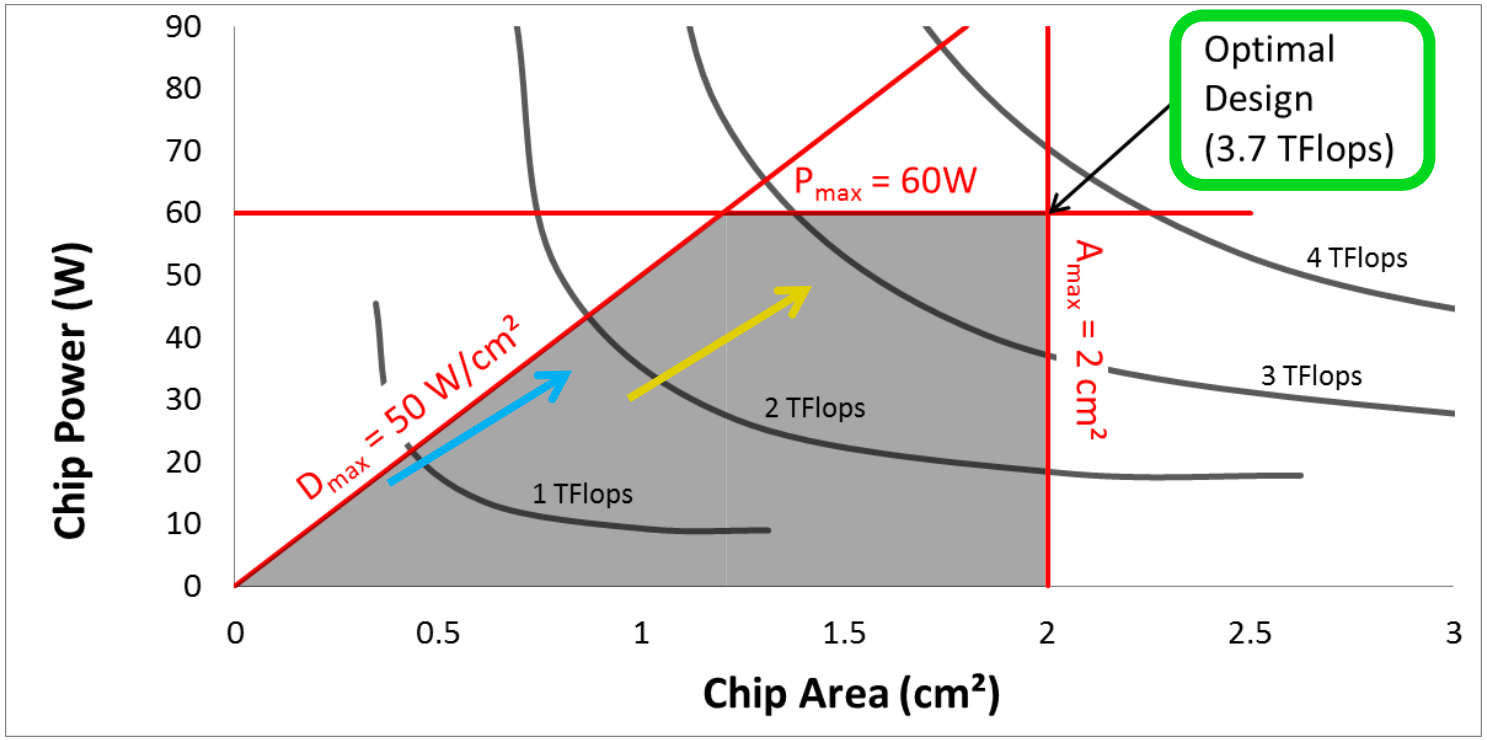
- Keep DRAM and Memory Hierarchy Dark
- First Focus on Memory Hierarchy

DARK MEMORY

- Keep DRAM and Memory Hierarchy Dark
- First Focus on Memory Hierarchy
- Algorithm: minimize DRAM access
- High chip level locality when parallelized



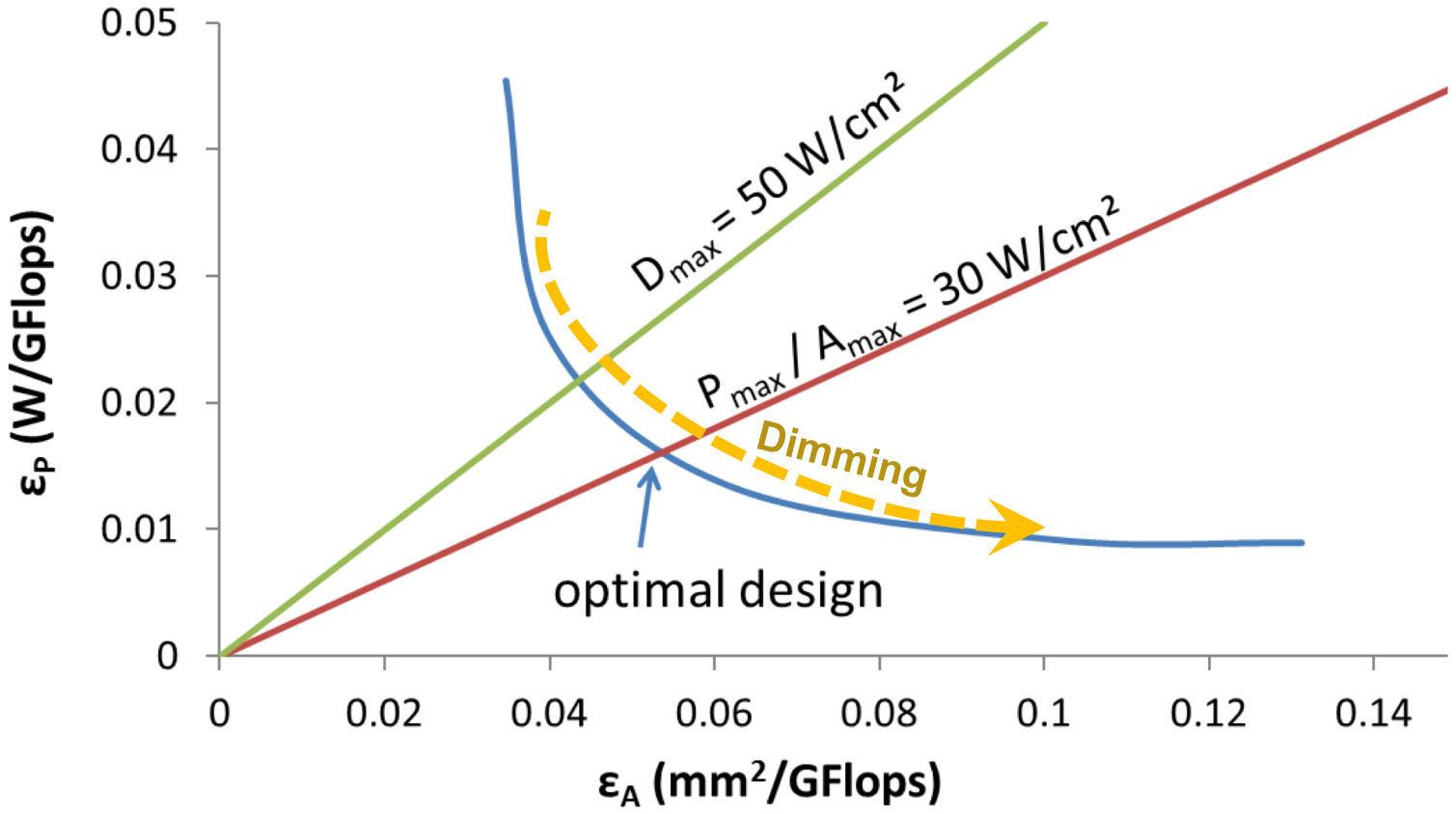
Metrics for Resource Constrained Computing



- Data Parallel Space
- More Performance

- More Hardware
  - More Power
  - More Area

Scale by Performance



- Energy/OP
- Area/OP

- One Pareto Optimal
- *Dim*ing the chip

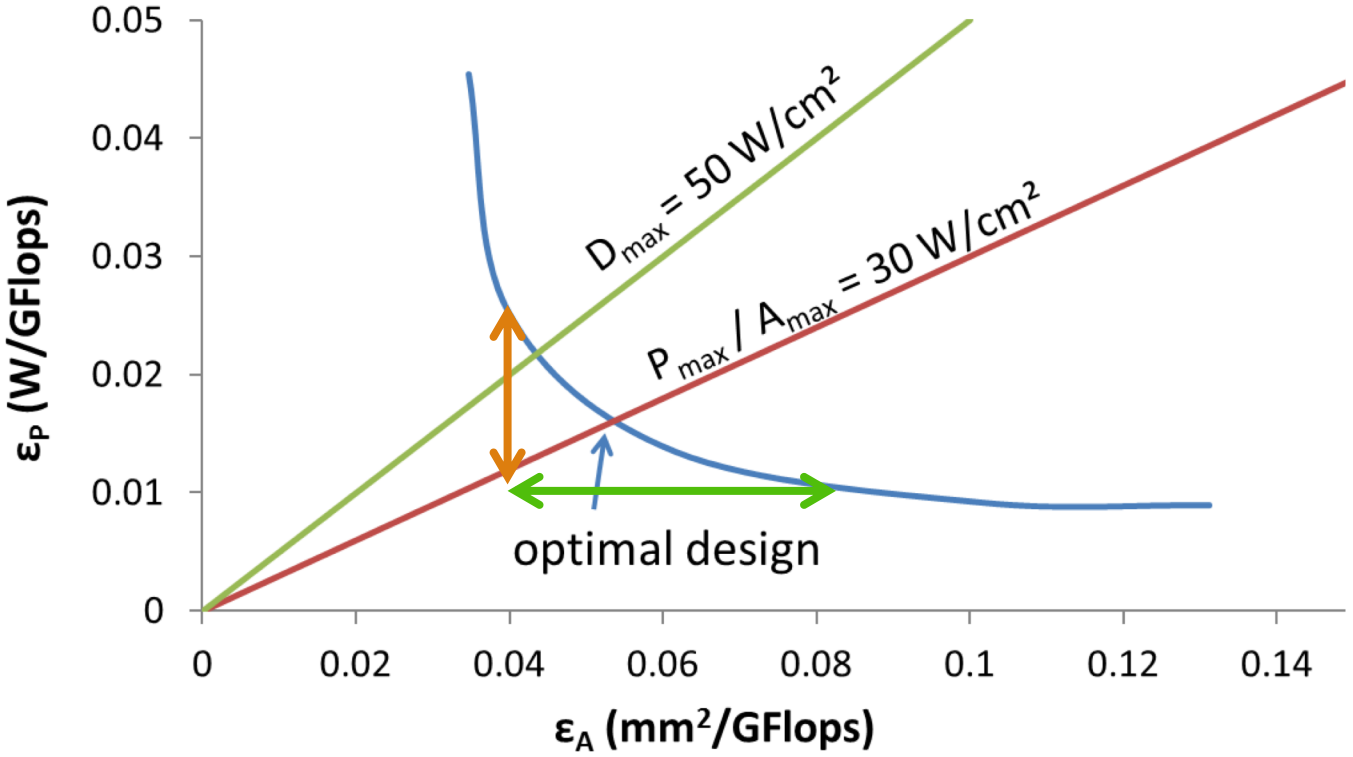
Always (FL)OPs? [Be Careful]

- Define *op:*

  Invariant across the different implementations

  - DFT and FFT
    - (Op is one Transform)
  - (Convolutional) Neural Networks
    - Op is one Inference
  - Dense and Sparse Solvers
    - Op is a number of solutions

System Design Optimization



- Sensitivity study along the Pareto Curves (Puzzle)
- Trade Area/Performance of one IP with another
- Both Scalable and Non-scalable

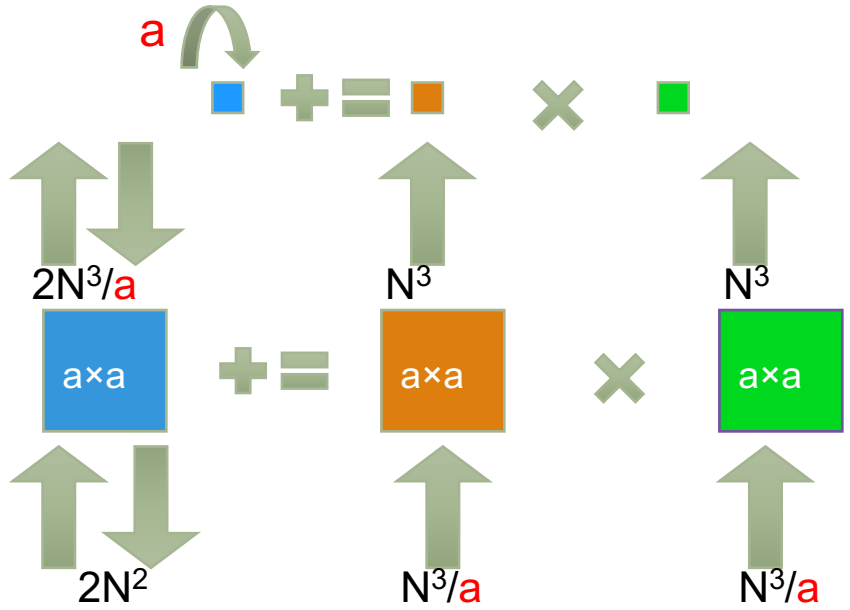
Example: Blocked GEMM

- A or B:

  From axa buffer
  - N^3/a

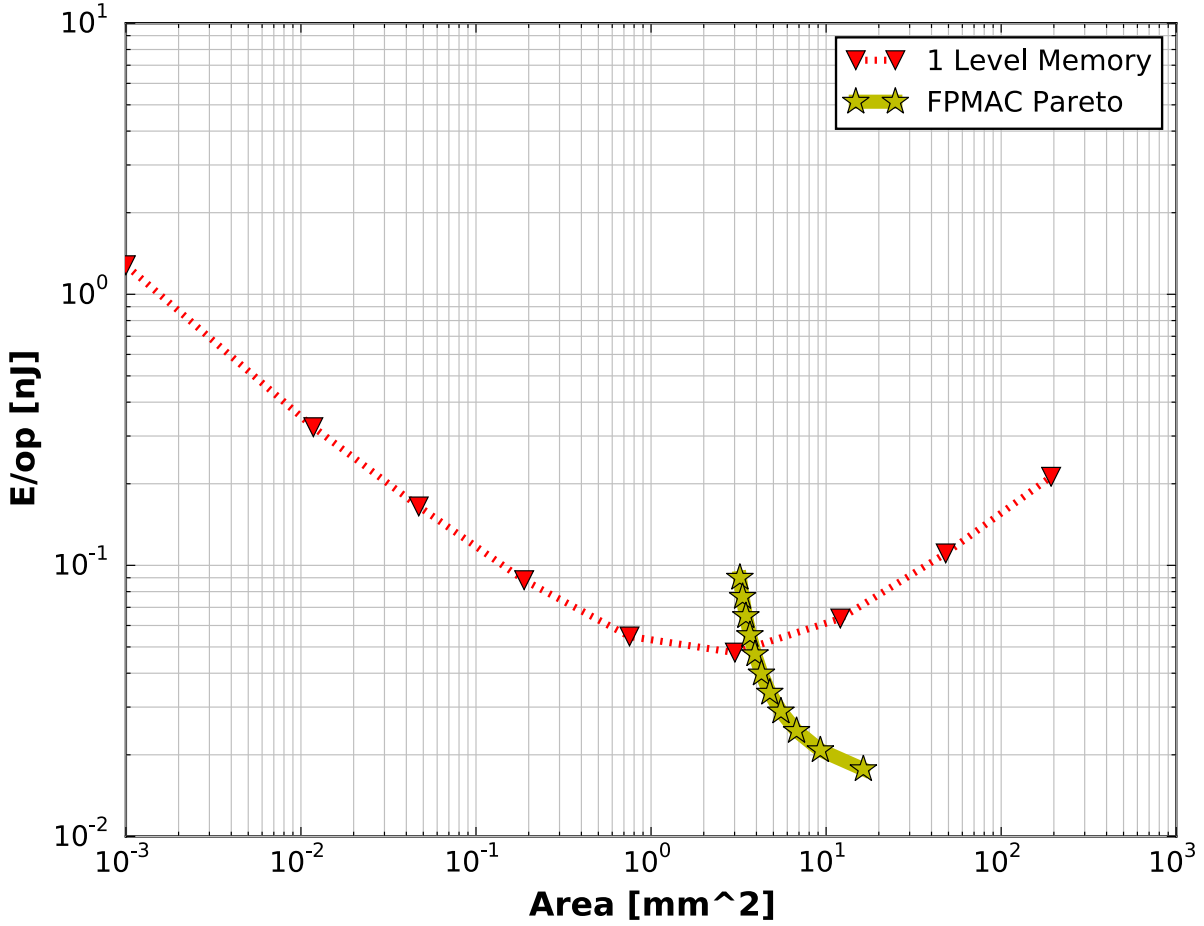
- C:

Access for last level cache is $2N^2$



- Highest level always suffers from constant N^3 accesses to A and B.

One Level Caching DP-FP



The Energy vs. Size of Top Level

- Energy Consumption

  Boosts significantly as size grows

- Memory Size:

  Does not compensate for the access savings

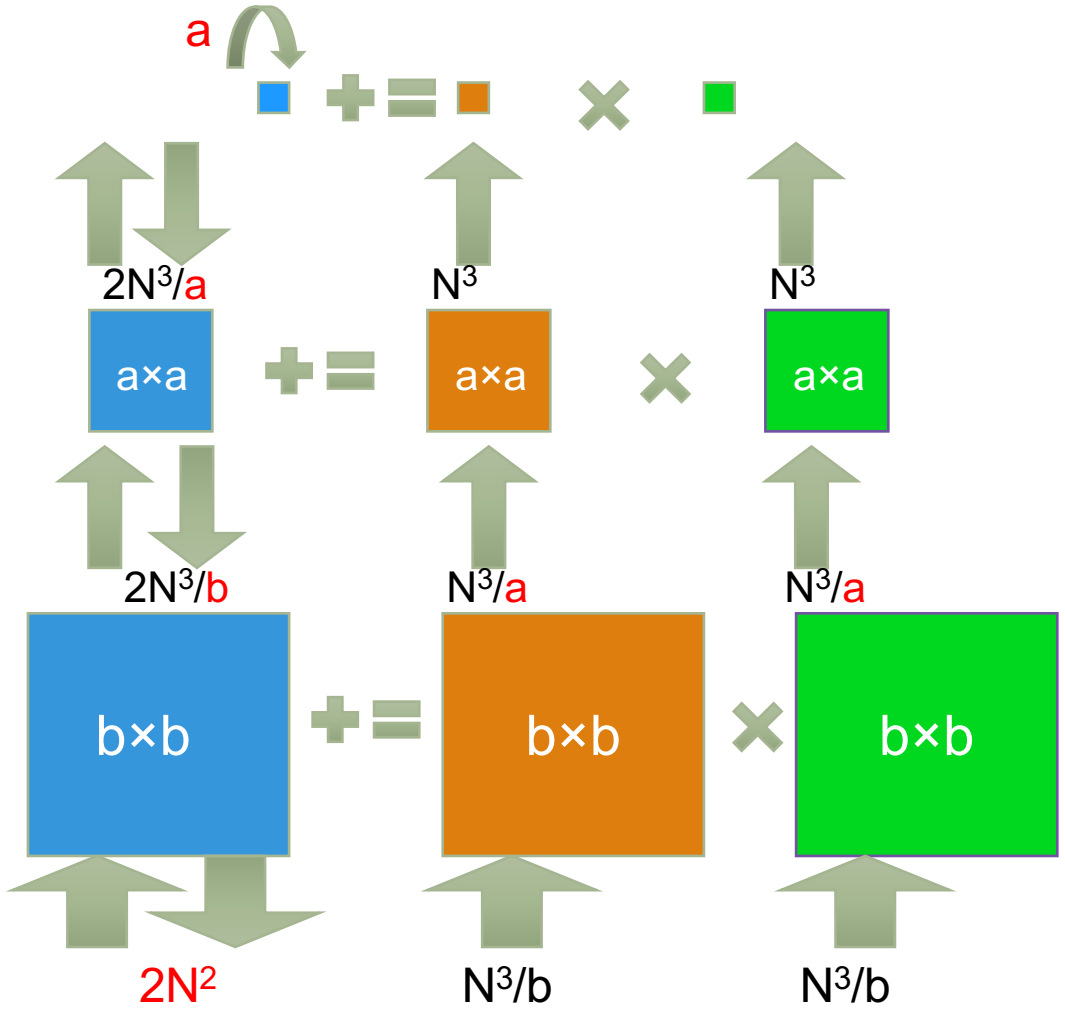
Let's Add Another Layer

- A or B:

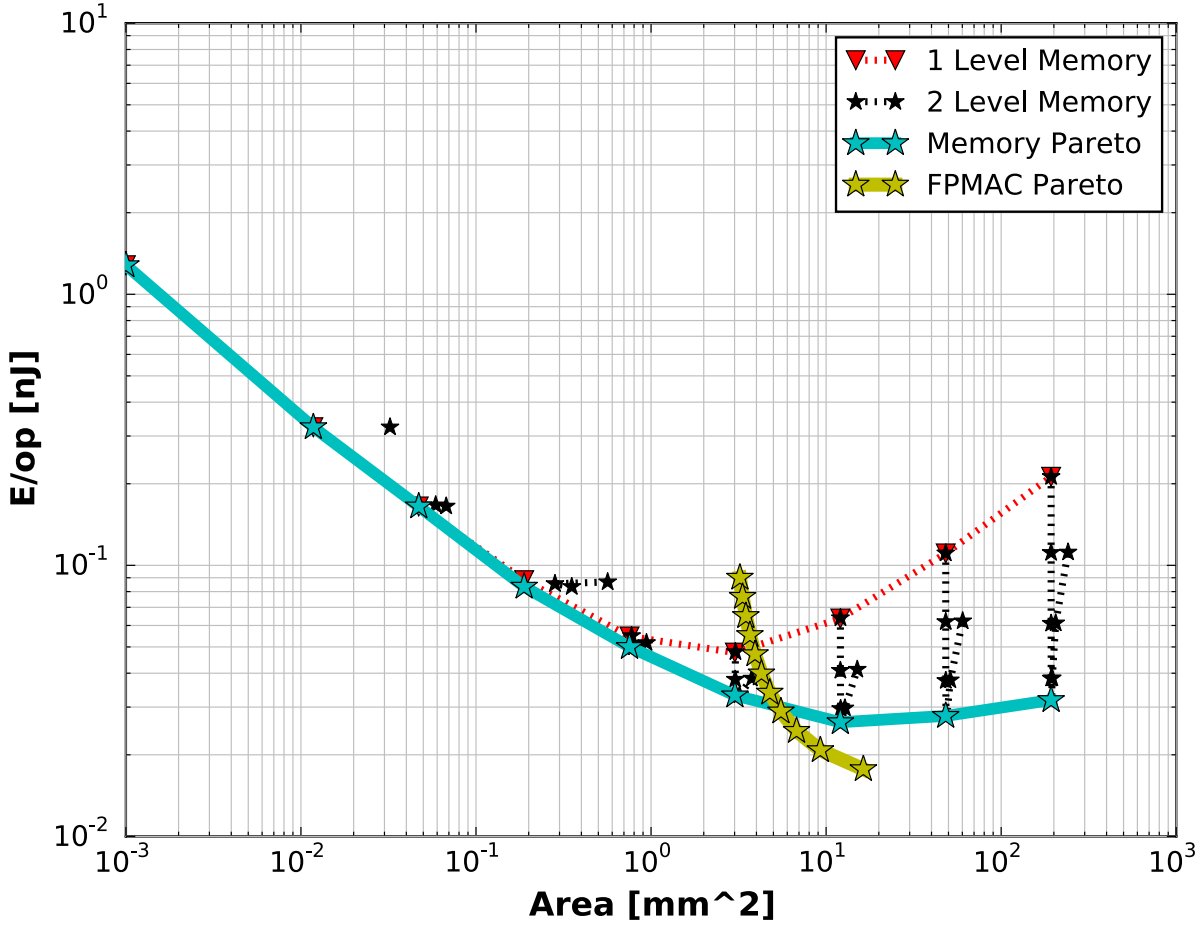
  From a bxb buffer
  - N^3/b

- C :

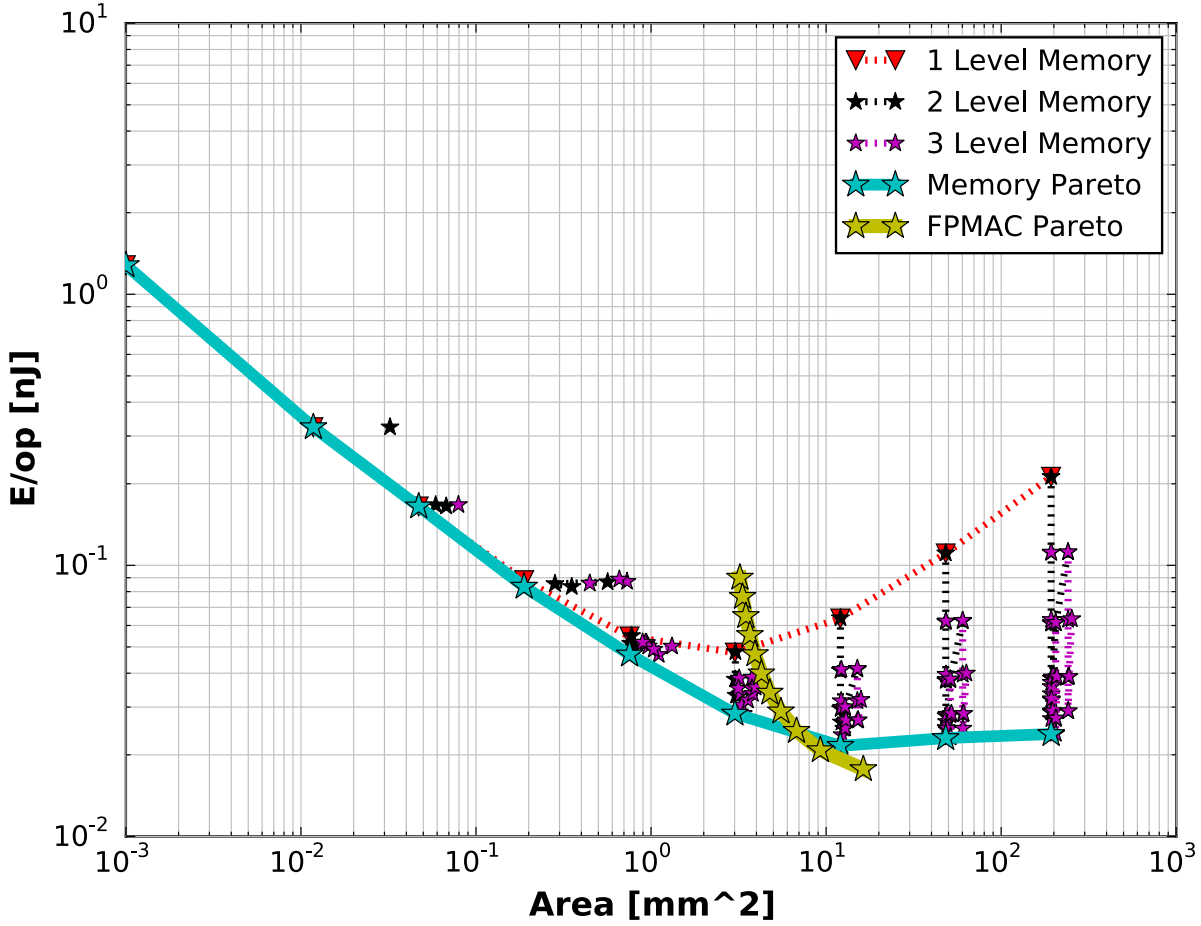
  Access for last level cache is $2N^2$



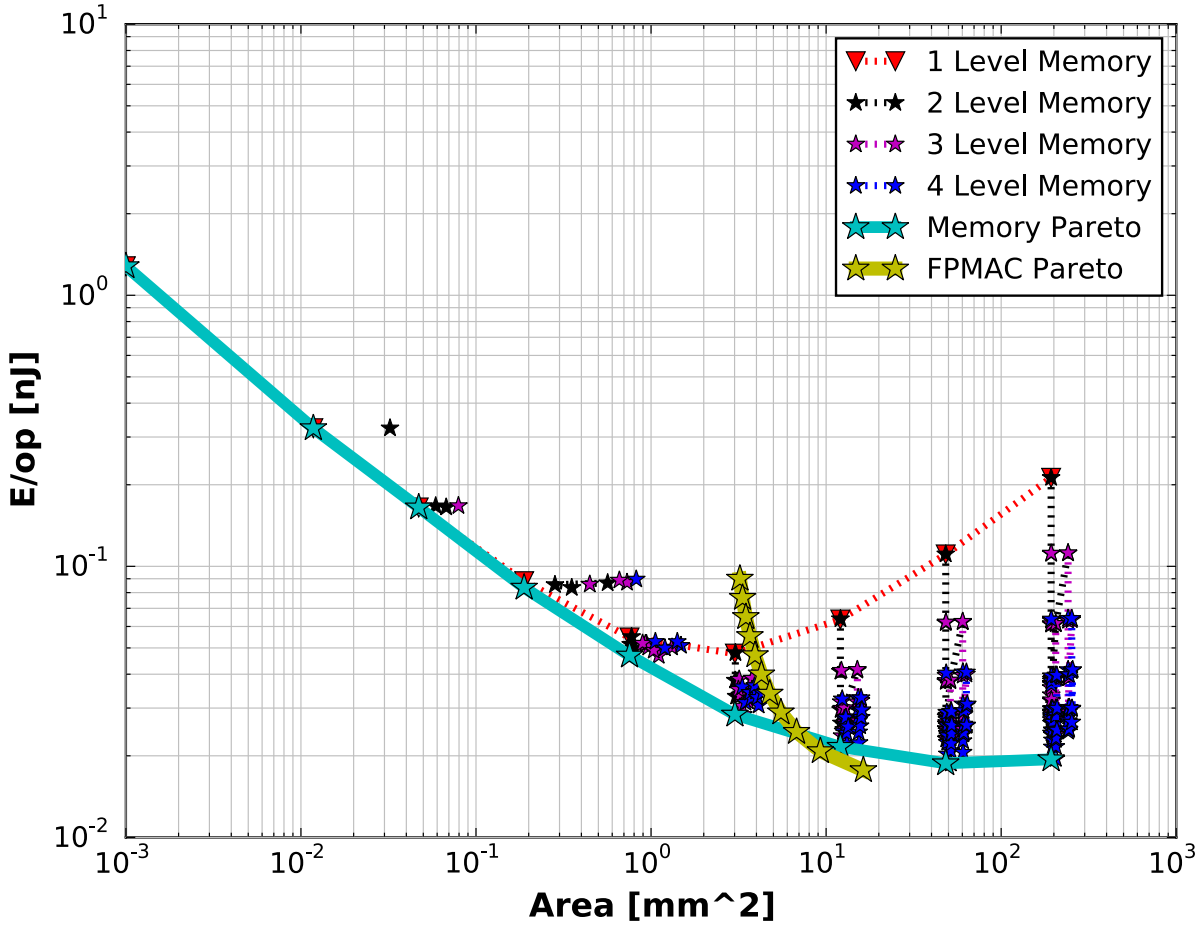
Second-level Caching



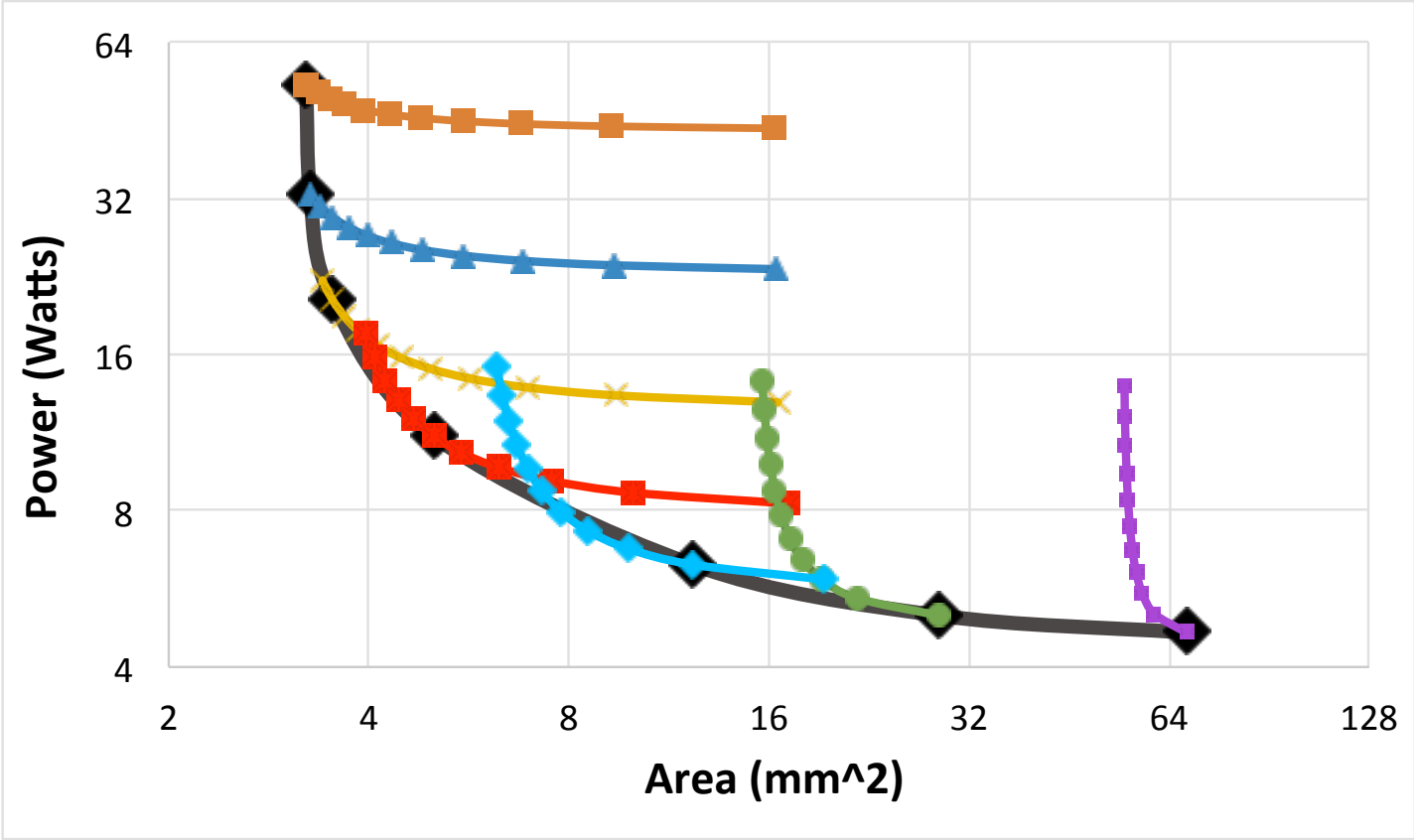
Three Levels



Memory Subsystem= Half Energy



Putting It All Together



- Merging FPU Pareto with Memory Pareto

Algorithm/ Architecture Codesign Methodology