Matrix Multiplication

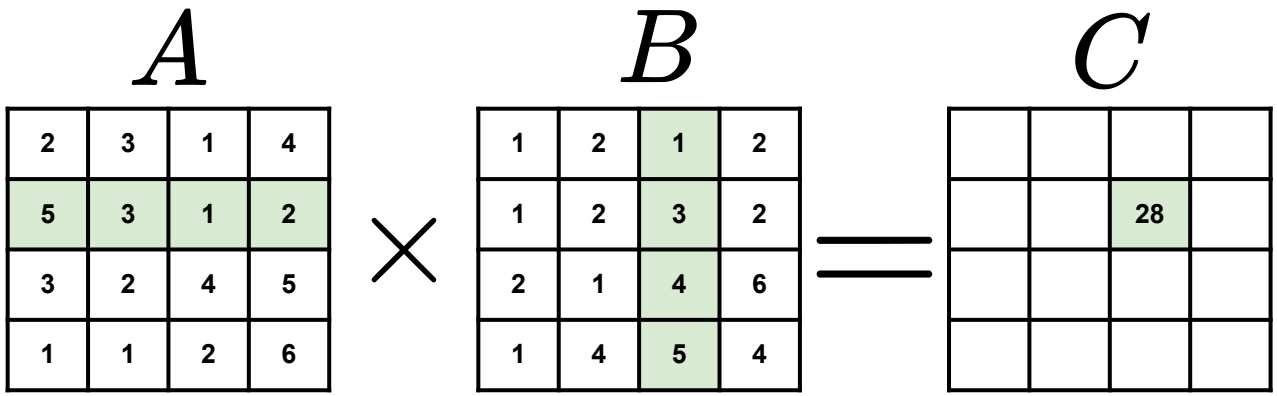
Speaker: Jia-Ming Lin

Outline

- Matrix Multiplication and Applications
- Complete Matrix Multiplication
  - Smaller size, e.g. 32x32
- Block Matrix Multiplication
  - Larger size, e.g. 1024x1024

Matrix Multiplication and Applications

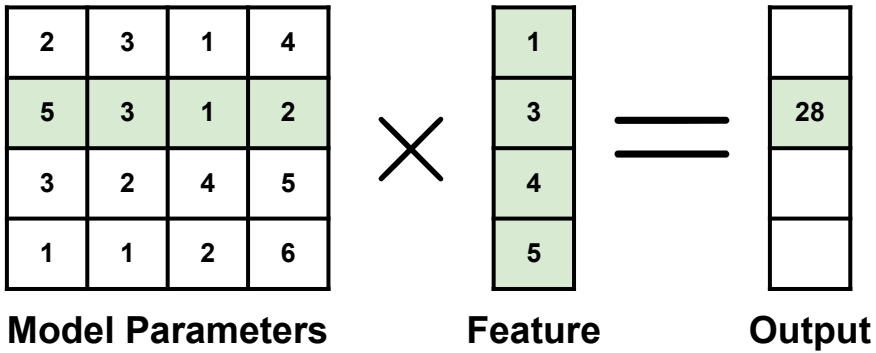
Matrix Multiplication



C[i][j] = inner product of A[i][...] and B[...][j]

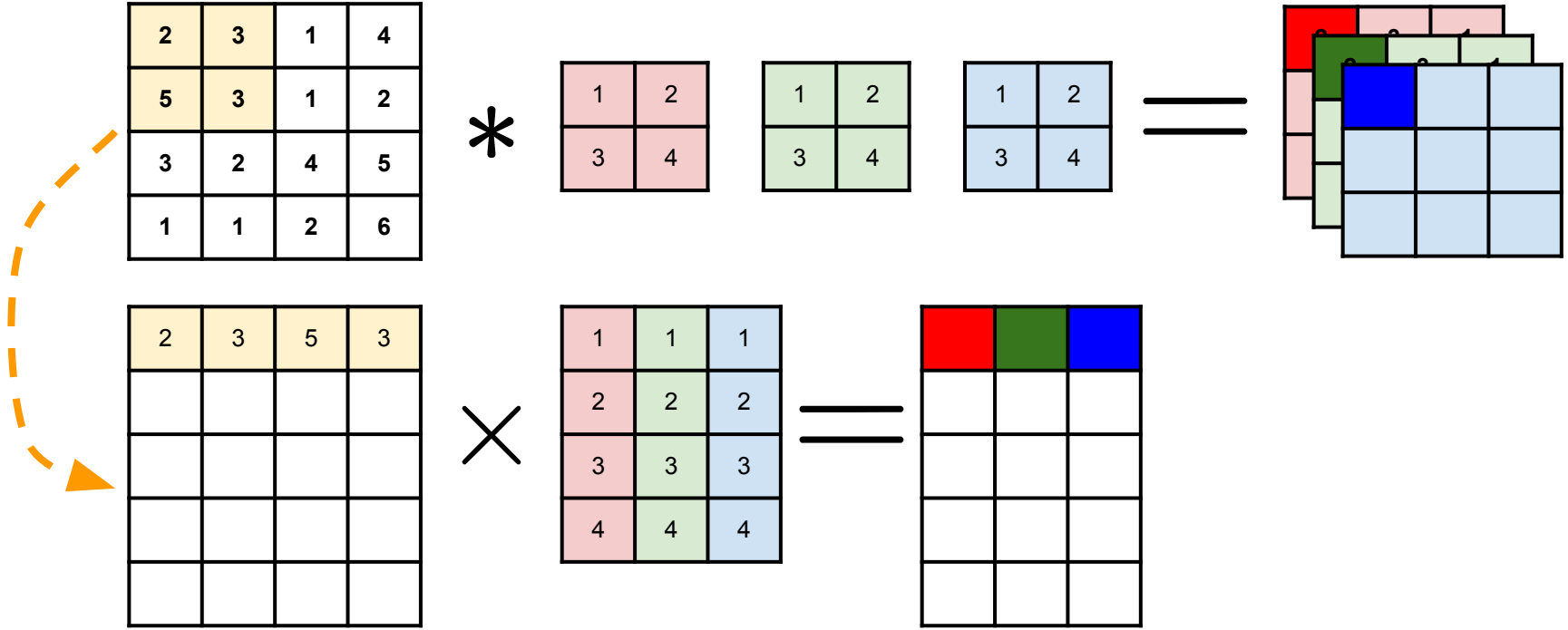
Application

- Computation of Neural Network
  - Fully connected layer



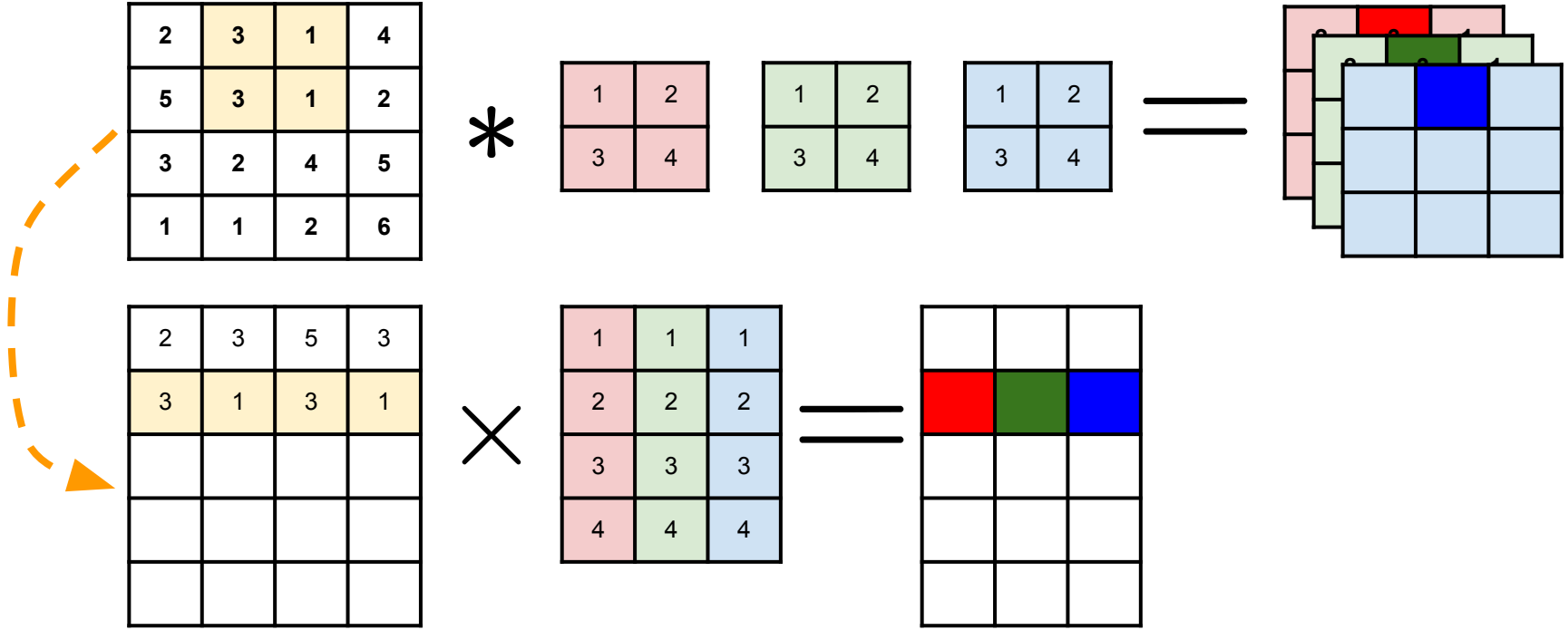
Application

- Computation of Neural Network
  - Convolution Layer



Application

- Computation of Neural Network
  - Convolution Layer



Memory Hierarchy

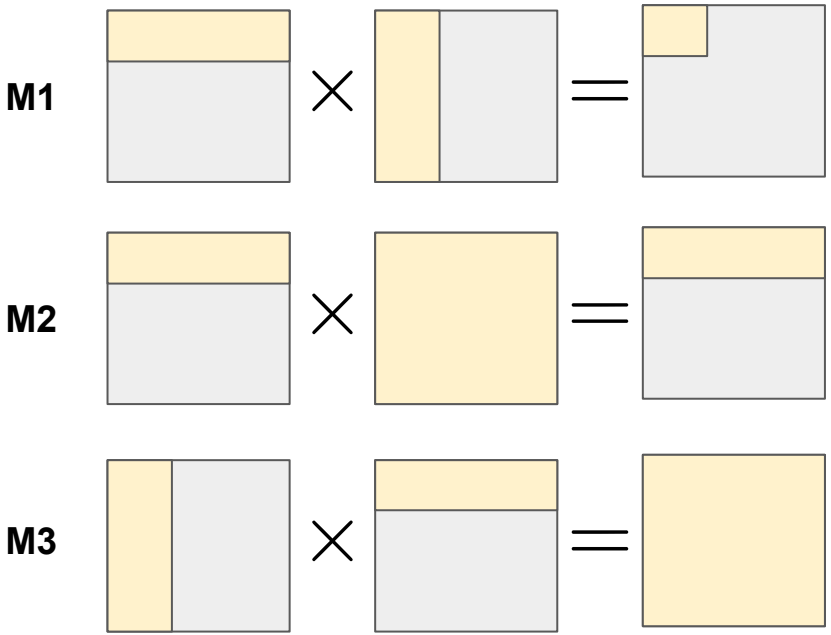
	External Memory	BRAM	FFs
count	1-4	thousands	millions
size	GBytes	KBytes	Bits
total size	GBytes	MBytes	100s of KBytes
width	8-64	1-16	1
total bandwidth	GBytes/sec	TBytes/sec	100s of TBytes/sec

- **External Memory**: highest density, lowest bandwidth
- **FFs**: highest total bandwidth, limited amount of data storage capability.
- **BRAM**: intermediate value between external memory and FFs

Complete Matrix Multiplication

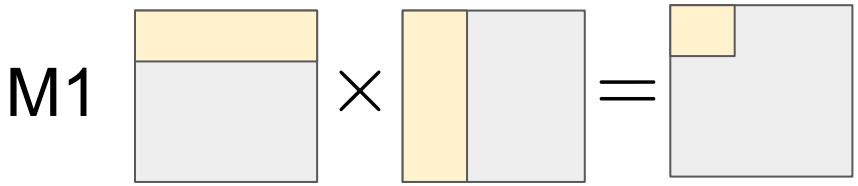
Methodologies

- Comparison
  - Latency, Resource Consumption

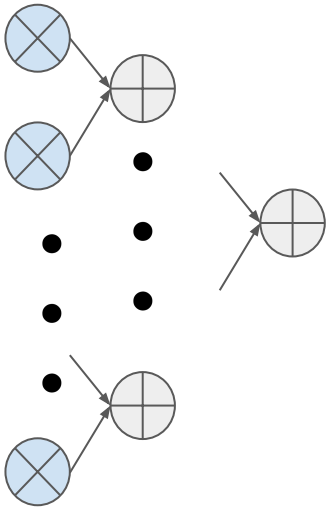


Suppose

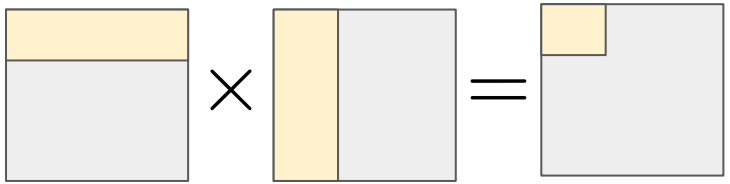
- Matrix size = 32x32
- Numbers in 16-bit integer
- Partial sum in 32-bit integer



- Compute one inner product in adder tree
  - One output: need log(32) = 5 cycles

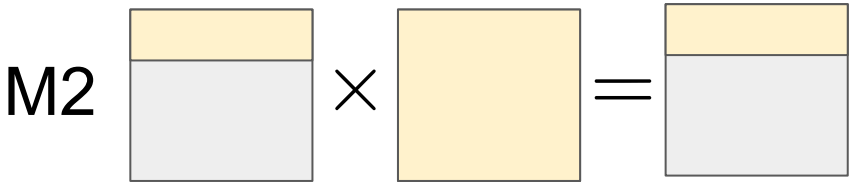


```
void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
    #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
    #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
    /* for each row and column of AB */
    row: for(int i = 0; i < N; ++i) {
        col: for(int j = 0; j < P; ++j) {
            #pragma HLS PIPELINE II=1
            /* compute (AB)i,j */
            int ABij = 0;
            product: for(int k = 0; k < M; ++k) {
                ABij += A[i][k] * B[k][j];
            }
            AB[i][j] = ABij;
        }
    }
}
```

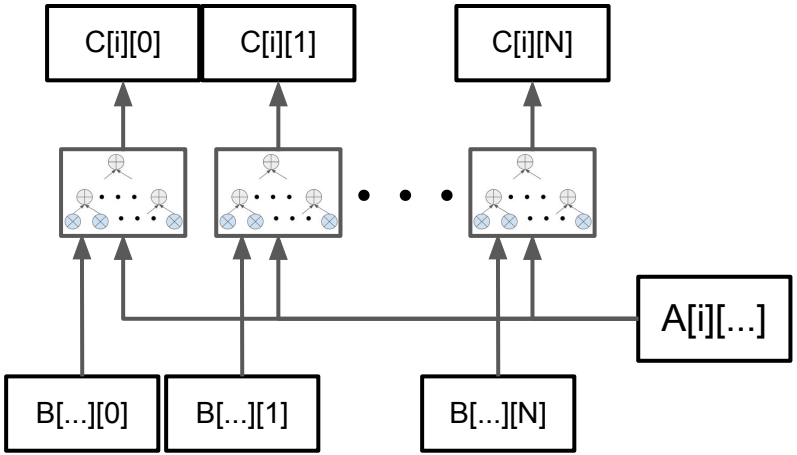
M1 

- Latency:
  - Pipeline Depth = 5
  - # of Iterations = 32x32
  - Initialization Interval = 1
  - Latency = 1024+4 = 1028
- Resource Consumption
  - DSP = 32

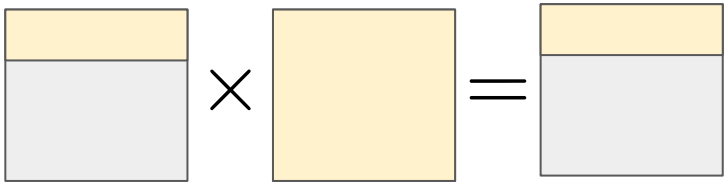
```
void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
    #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
    #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
    /* for each row and column of AB */
    row: for(int i = 0; i < N; ++i) {
        col: for(int j = 0; j < P; ++j) {
            #pragma HLS PIPELINE II=1
            /* compute (AB)i,j */
            int ABij = 0;
            product: for(int k = 0; k < M; ++k) {
                ABij += A[i][k] * B[k][j];
            }
            AB[i][j] = ABij;
        }
    }
}
```



- Multiple parallel adder trees
  - One output: need log(32) = 5 cycles

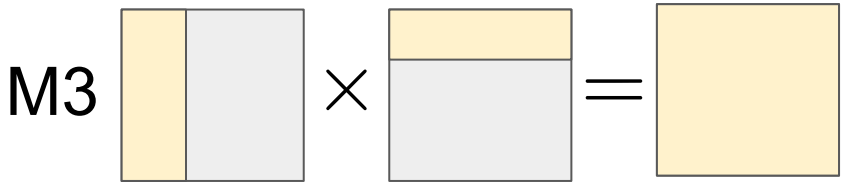



```
void inner_product(A_i, B_j, C_ij){
#pragma HLS INLINE off
        for(k = 0...N){ // Unroll
                C_ij += A_i[k] * B_j[k]
        }
}

void mat_mul(A[N][N], B[N][N], C[N][N]){
#pragma HLS ALLOCATION instances=inner_product
limit=32 function
        for(i = 0…N){
                for(j = 0...N){ // Unroll
                        inner_product(A[i][...], B[...][j], C[i][j])
                }
        }
}
```

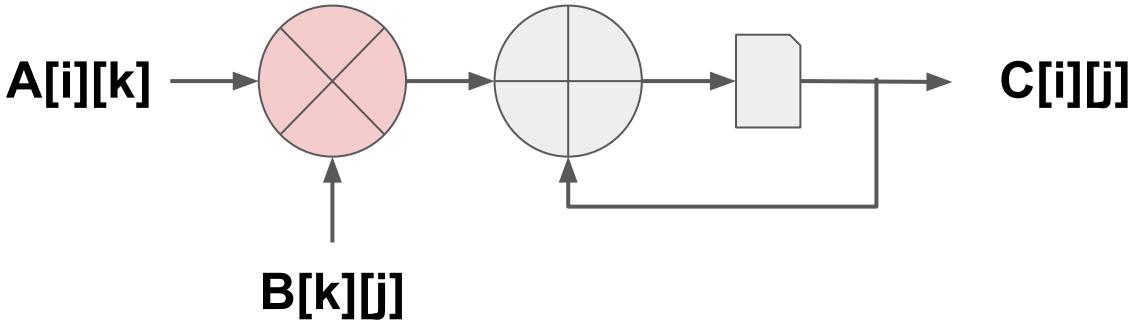
M2 

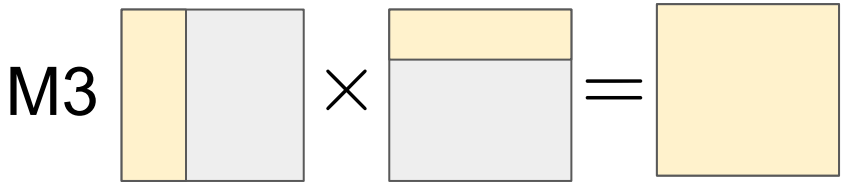
- Latency:
  - Pipeline Depth = 5
  - # of Iterations = 32
  - Initialization Interval = 5
  - Latency = 32 * 5 = 160 (cycles)
- Resource Consumption
  - DSP = 32*32 = 1024

```
void inner_product(A_i, B_j, C_ij){
#pragma HLS INLINE off
        for(k = 0...N){ // Unroll
                C_ij += A_i[k] * B_j[k]
        }
}

void mat_mul(A[N][N], B[N][N], C[N][N]){
#pragma HLS ALLOCATION instances=inner_product
limit=32 function
        for(i = 0…N){
                for(j = 0...N){ // Unroll
                        inner_product(A[i][...], B[...][j], C[i][j])
                }
        }
}
```

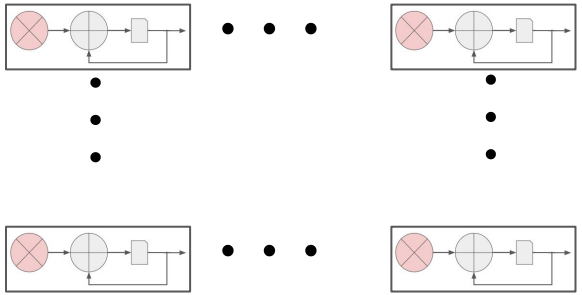


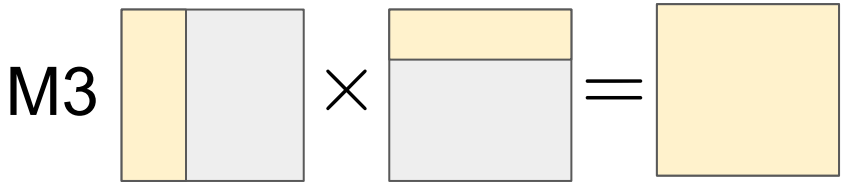
- Parallelize Partial Sum Computations
  - Parallelized outer product



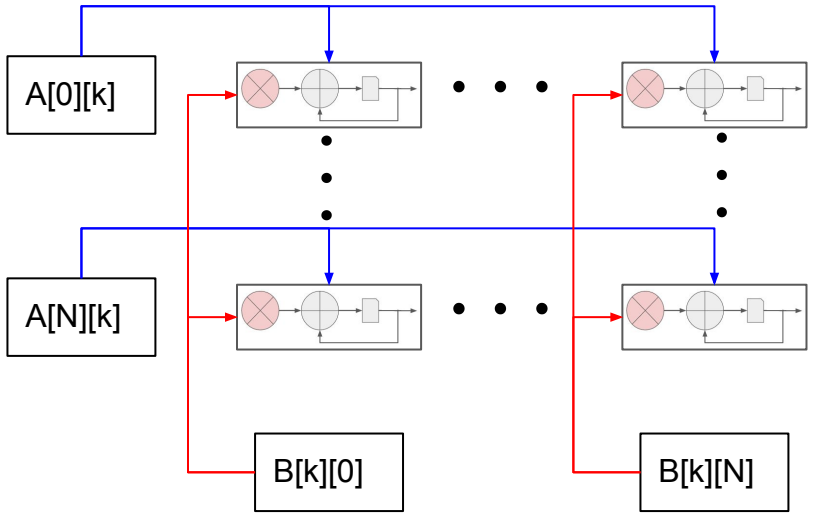


- Parallelize Partial Sum Computations

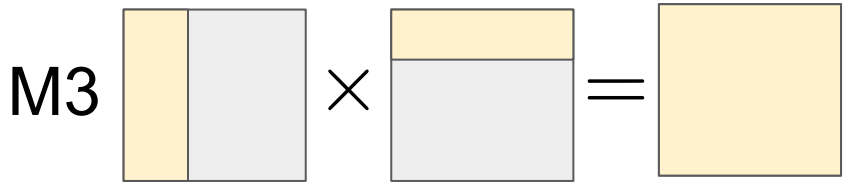




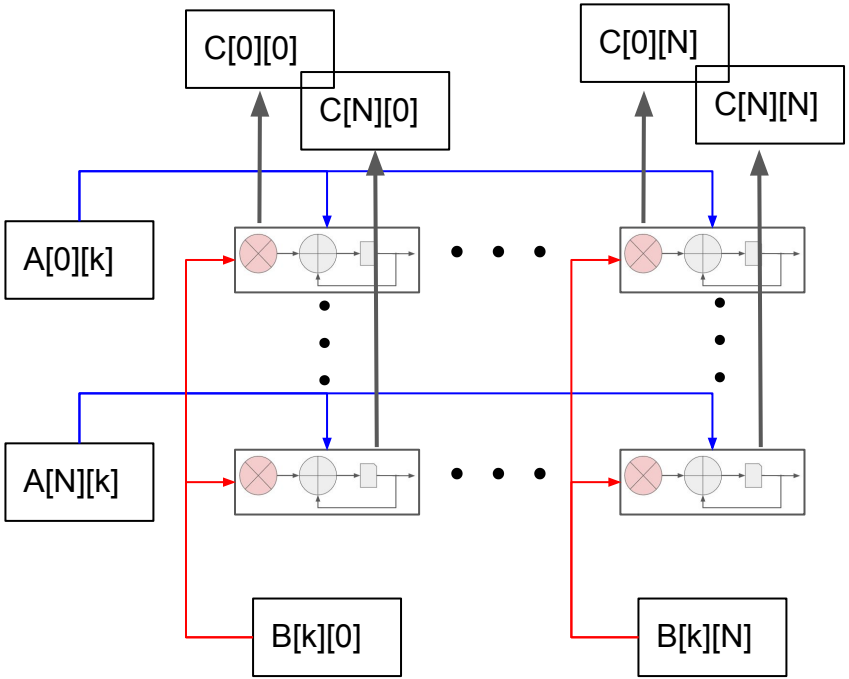
- Parallelize Partial Sum Computations



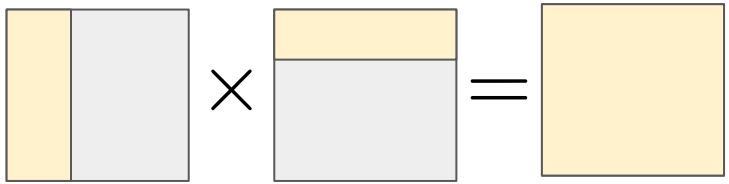
k = 0...N



- Parallelize Partial Sum Computations




```
int32 PE(a, b, c){
        return a*b+c
}

void mat_mul(A[N][N], B[N][N], C[N][N]){
#pragma HLS ALLOCATION instances=PE limit=1024 function
        for(k=0...N){ // pipeline
                A_col = A[...][k];
                B_row = B[k][...];
                for(i = 0...N){ // unroll
                        for(j = 0...N){ // unroll
                                C[i][j] = PE(A_col[i], B_row[j], C[i][j]);
                        }
                }
        }
}
```

M3 

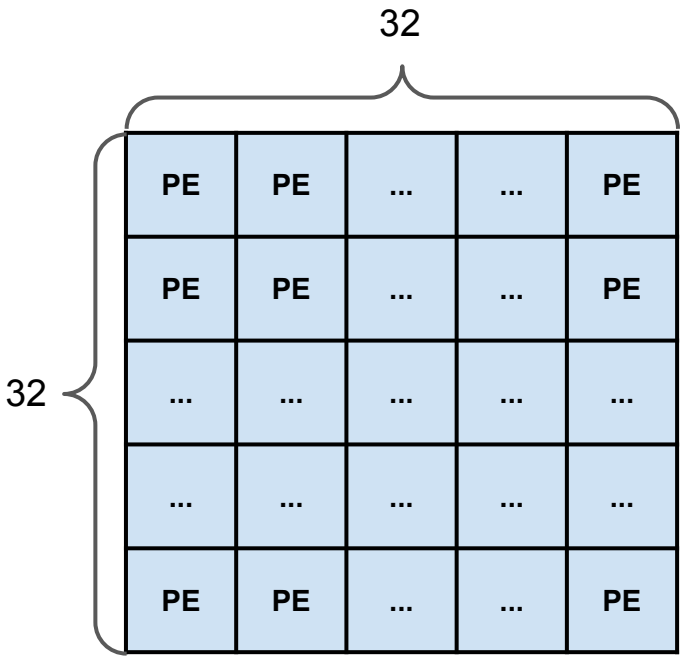
- Latency:
  - Pipeline Depth = 2
  - # of iterations = 32
  - Initialize interval = 1
  - Latency = 1+32 = 33 (cycles)
- Resource Consumption
  - DSP = 32*32 = 1024

```
int32 PE(a, b, c){
        return a*b+c
}

void mat_mul(A[N][N], B[N][N], C[N][N]){
#pragma HLS ALLOCATION instances=PE limit=1024 function
        for(k=0...N){ // pipeline
                A_col = A[...][k];
                B_row = B[k][...];
                for(i = 0...N){ // unroll
                        for(j = 0...N){ // unroll
                                C[i][j] = PE(A_col[i], B_row[j], C[i][j]);
                        }
                }
        }
}
```

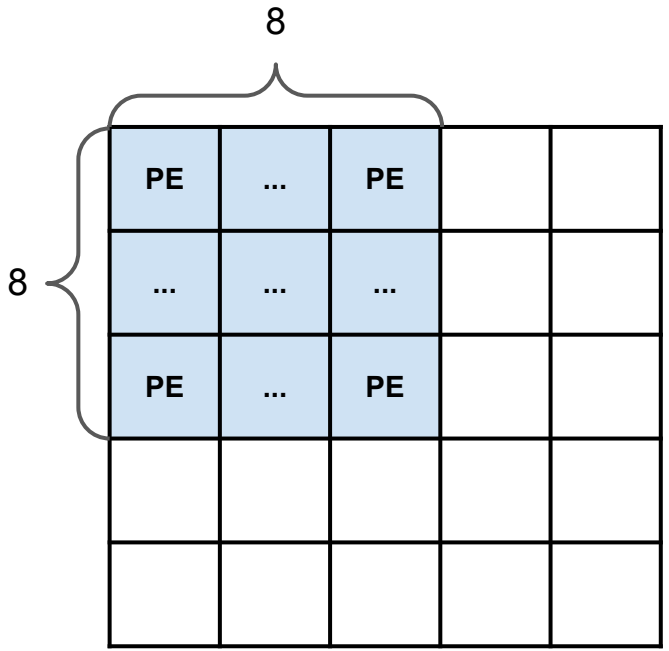
Block Matrix Multiplication

Issues in last design, M3

- Too many DSP cost to implement on PYNQ-Z2

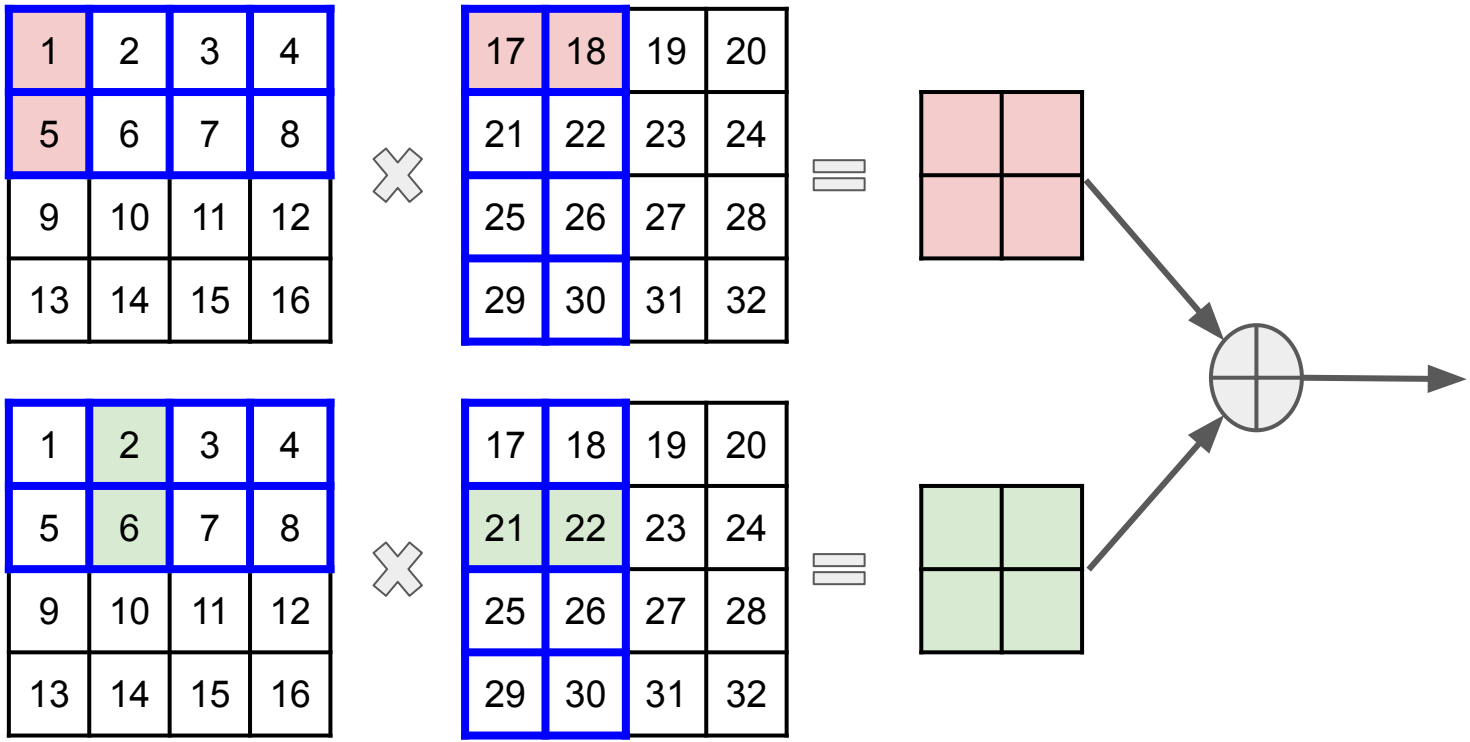


DSP usage = 1024



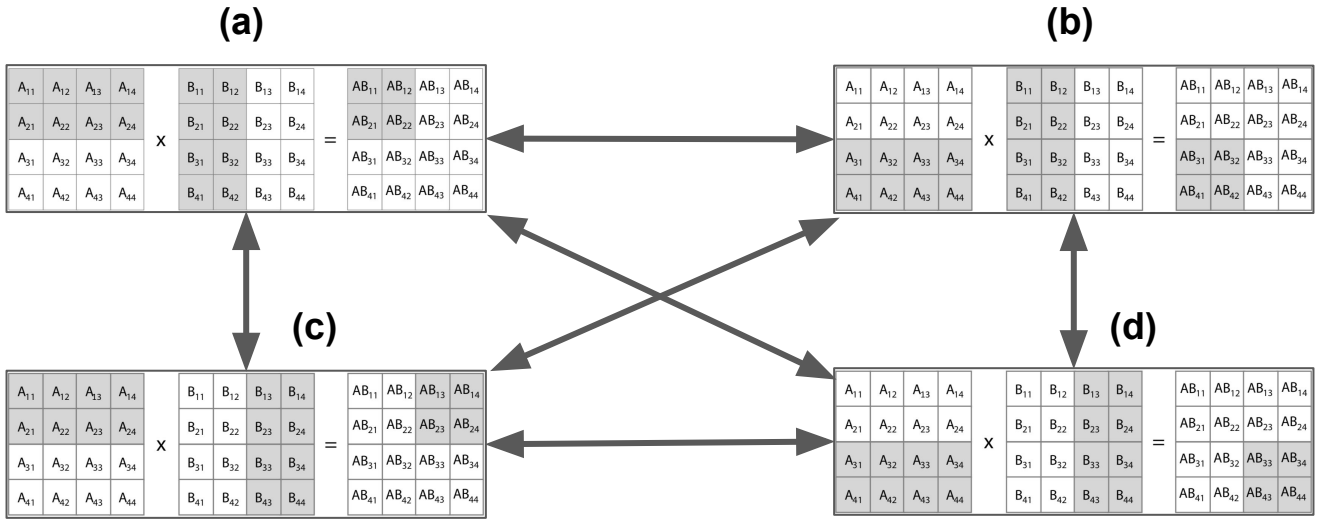
DSP usage = 64

Block Matrix Multiplication with Outer Product



Block Matrix Multiplication with Outer Product

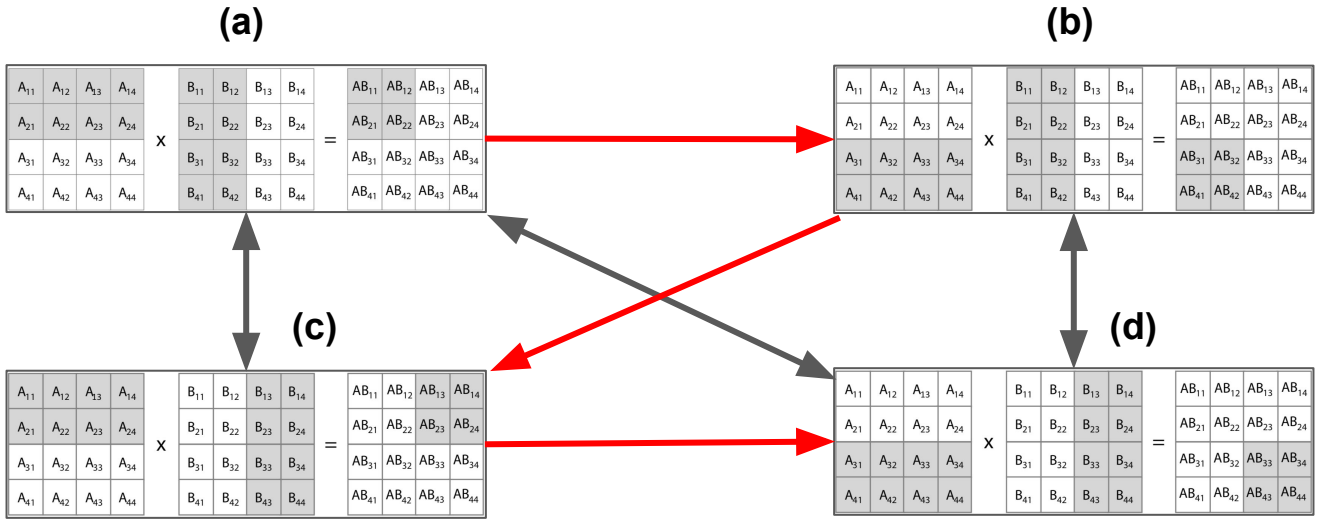
- Considering Memory Hierarchy



Computing order: **(a) → (d) → (b) → (c),** share memory(DRAM) read # = 5

Block Matrix Multiplication with Outer Product

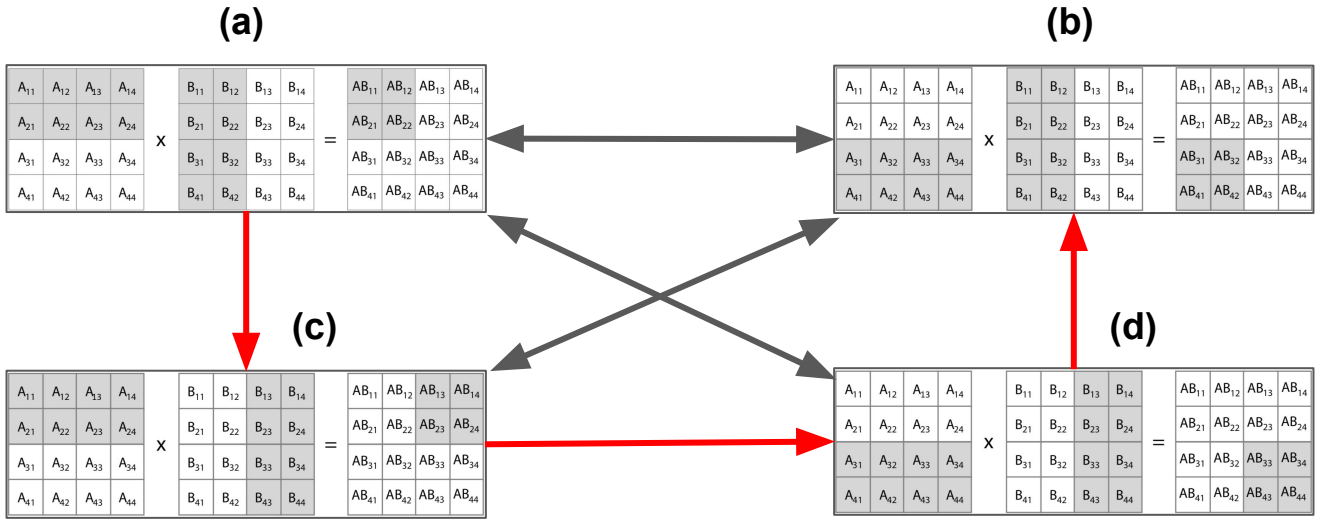
- Considering Memory Hierarchy



Computing order: **(a) → (c) → (b) → (d),** share memory(DRAM) read # = 4

Block Matrix Multiplication with Outer Product

- Considering Memory Hierarchy



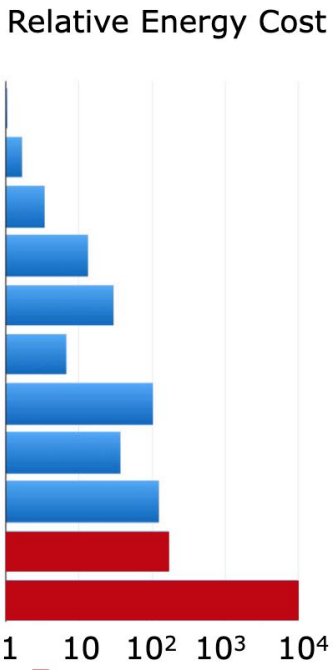
Computing order: **(a) → (c) → (d) → (b),** share memory(DRAM) read # = 3

Block Matrix Multiplication with Outer Product

- Considering Memory Hierarchy

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Multiply	0.2
32b Multiply	3.1
16b FP Multiply	1.1
32b FP Multiply	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

[**Horowitz**, *ISSCC* 2014]



Block Matrix Multiplication with Outer Product

- HLS Implementation: **Read Block of A**

```
  if(counter == 0){ //only load the A rows when necessary
    loadA: for(int i = 0; i < SIZE; i++) {

      blockvec tempA = Arows.read();
      for(int j = 0; j < BLOCK_SIZE; j++) {
#pragma HLS PIPELINE
        A[j][i] = tempA.data[j];
      }
    }
  }
```

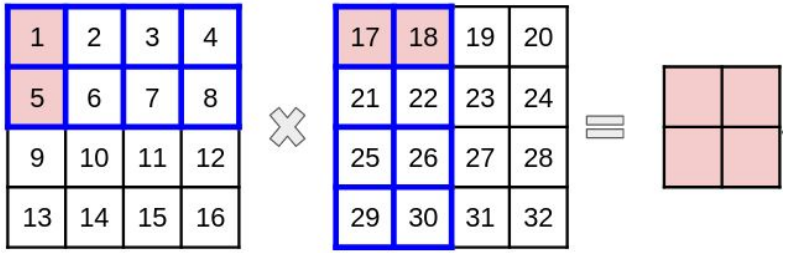
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Baseline Code from: https://github.com/KastnerRG/pp4fpgas/blob/master/examples/block_mm.cpp

Block Matrix Multiplication with Outer Product

- HLS Implementation: **Compute Outer Product**

```
int AB[BLOCK_SIZE][BLOCK_SIZE] = {0};

partialsum: for(int k=0; k < SIZE; k++) {
  blockvec tempB = Bcols.read();
  for(int i = 0; i < BLOCK_SIZE; i++) {
    for(int j = 0; j < BLOCK_SIZE; j++) {
      AB[i][j] = AB[i][j] +  A[i][k] * tempB.data[j];
    }
  }
}
```



Baseline Code from: https://github.com/KastnerRG/pp4fpgas/blob/master/examples/block_mm.cpp

Block Matrix Multiplication with Outer Product

- HLS Implementation: **Write Block MatMul Result**

```
writeoutput: for(int i = 0; i < BLOCK_SIZE; i++) {
  for(int j = 0; j < BLOCK_SIZE; j++) {
    ABpartial.out[i][j] = (DTYPE)AB[i][j];
  }
}
```

Baseline Code from: https://github.com/KastnerRG/pp4fpgas/blob/master/examples/block_mm.cpp

Block Matrix Multiplication with Outer Product

- HLS Implementation: **Synthesis Result**

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.510 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
7195	7547	71.950 us	75.470 us	7195	7547	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	3	0	349	-
FIFO	-	-	-	-	-
Instance	-	-	0	90	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	347	-
Register	-	-	759	-	-
Total	2	3	759	786	0
Available	280	220	106400	53200	0
Utilization (%)	~0	1	~0	1	0

Latency larger than M1 = 1028

Baseline Code from: https://github.com/KastnerRG/pp4fpgas/blob/master/examples/block_mm.cpp

Block Matrix Multiplication with Outer Product

- HLS Implementation: **Latency Break down**

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loadA	352	352	11	-	-	32	no
+ loadA.1	8	8	1	1	1	8	yes
- memset_AB	71	71	9	-	-	8	no
+ memset_AB	7	7	1	-	-	8	no
- partialsum	6976	6976	218	-	-	32	no
+ partialsum.1	216	216	27	-	-	8	no
++ partialsum.1.1	24	24	3	-	-	8	no
- writeoutput	144	144	18	-	-	8	no
+ writeoutput.1	16	16	2	-	-	8	no

```
int AB[BLOCK_SIZE][BLOCK_SIZE] = {0};

partialsum: for(int k=0; k < SIZE; k++) {
  blockvec tempB = Bcols.read();
  for(int i = 0; i < BLOCK_SIZE; i++) {
    for(int j = 0; j < BLOCK_SIZE; j++) {
      AB[i][j] = AB[i][j] + A[i][k] * tempB.data[j];
    }
  }
}
```

Sequentially Compute

How to speedup? Apply parallelism of M3!

Baseline Code from: https://github.com/KastnerRG/pp4fpgas/blob/master/examples/block_mm.cpp

Lab: Combine Block MatMul with method M3

- Input data
  - Two square matrices, A,B in 32x32
  - Number representation:
    - 16-bit integer for data
    - 32-bit integer for partial sum
- Goal:
  - Using block size = 8x8
    - 30~50 cycles for each output block
  - Overall, (30~50)x16 = 480~800 cycles to complete
- Note
  - Or other blocking size to suppress my result

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.268 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
38	38	0.380 us	0.380 us	36	36	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	306	-
FIFO	0	-	389	2172	-
Instance	-	64	4384	1684	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	630	-
Register	-	-	73	-	-
Total	0	64	4846	4792	0
Available	280	220	106400	53200	0
Utilization (%)	0	29	4	9	0