RC64 Many-Core DSP
Architecture, System, Software and Algorithms

(Shared Memory Many-core with Hardware Scheduling)

Ran Ginosar

Technion & Ramon Chips

MPSoC 2017

Outline

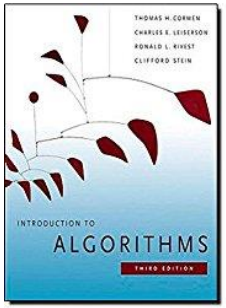
- Motivation: Programming model
- Architecture
- Implementation
- Programming model
- Algorithms

many-cores

- Many-core is:
  - a single chip
  - with many cores (how many?) and on-chip memory (how much?)
  - running one (parallel) program at a time, solving one problem
  - an accelerator
- Many-core is NOT:
  - Not a “normal” multi-core
  - Not running an OS
- Contending many-core architectures
  - Shared memory (RC64)
  - Networked (Tilera, Xeon Phi)
  - GPU (Nvidia)
- Contending programming models
  - Shared memory
  - Message passing

One (parallel) program ?

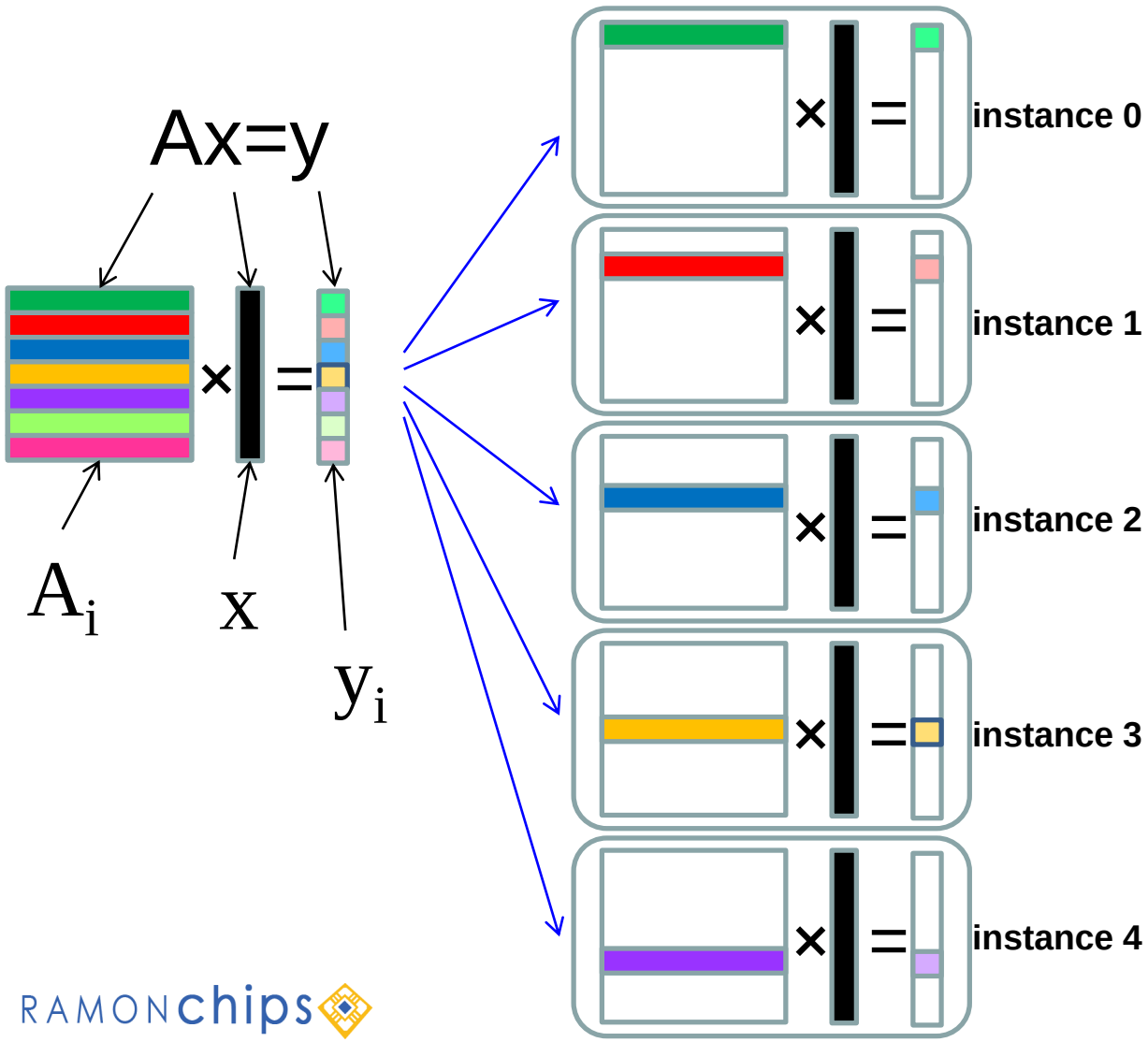
- Best formal approach to parallel programming is the PRAM model
- Manages
  - all cores as a single shared resource
  - all memory as a single shared resource
- and more…



Cormen, Leiserson, Rivest, Stein,
Introduction to algorithms,
2009

Joseph F. JaJa,
Introduction to Parallel Algorithms,
1992

PRAM matrix-vector multiply



The PRAM algorithm

i is row index

Begin

$y_i = A_i x$

End

A,x,y in shared memory
(Concurrent Read of x)

Temporary variables
in private memories

Any core may
execute instance i

Advantages of PRAM-like programming

- Simpler program
  - Flat memory model
  - Same data structures as in serial code
  - No code for finding and moving the data
  - Easier programming, lower energy, higher performance
  - Scalable to higher number of cores

Advantages of PRAM-like programming

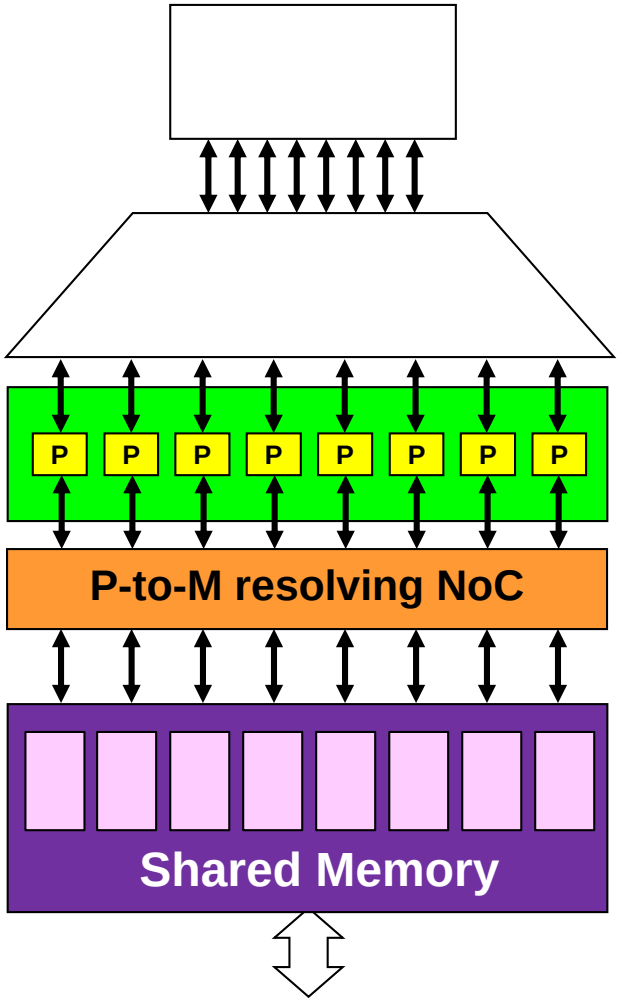
- Same-node Scalability
  - Easy to define high levels of parallelism
  - Scalable to more cores running slower at lower voltage
    - on same technology node
  - Example: same-node-scaling from N to 2N cores
    same-node-scaling of Vdd and f by α = 0.8, … ,0.5

	N cores, Vdd, f	2N cores, αVdd, αf	α = 0.8	α = 0.7	α = 0.6	α = 0.5
Perf	$P(N) = Nf$	$2N\alpha f = 2\alpha P(N)$	$P(N) \cdot 1.6$	$P(N) \cdot 1.4$	$P(N) \cdot 1.2$	$P(N)$
Time	$T(N) = \frac{W}{Nf}$	$\frac{W}{2N\alpha f} = \frac{T(N)}{2\alpha}$	$\frac{T(N)}{1.6}$	$\frac{T(N)}{1.4}$	$\frac{T(N)}{1.2}$	$T(N)$
Power	$PW(N) = NCV^2 f$	$2NC\alpha^2 V^2 \alpha f = 2\alpha^3 PW(N)$	$PW(N)$	$PW(N) \cdot 0.7$	$PW(N) \cdot 0.4$	$PW(N) \cdot 0.25$
Energy	$E(N) = P(N)T(N) = WCV^2$	$2\alpha^3 PW(N) \cdot T(N)/2\alpha = \alpha^2 E(N)$	$E(N) \cdot 0.64$	$E(N) \cdot 0.5$	$E(N) \cdot 0.36$	$E(N) \cdot 0.25$
Perf / Pwr	$PPR(N) = 1/CV^2$	$\frac{PPR(N)}{\alpha^2}$	$PPR(N) \cdot 1.5$	$PPR(N) \cdot 2$	$PPR(N) \cdot 2.8$	$PPR(N) \cdot 4$

Outline

- Motivation: Programming model
- Architecture
- Implementation
- Programming model
- Algorithms

RC64 conceptual architecture: part I



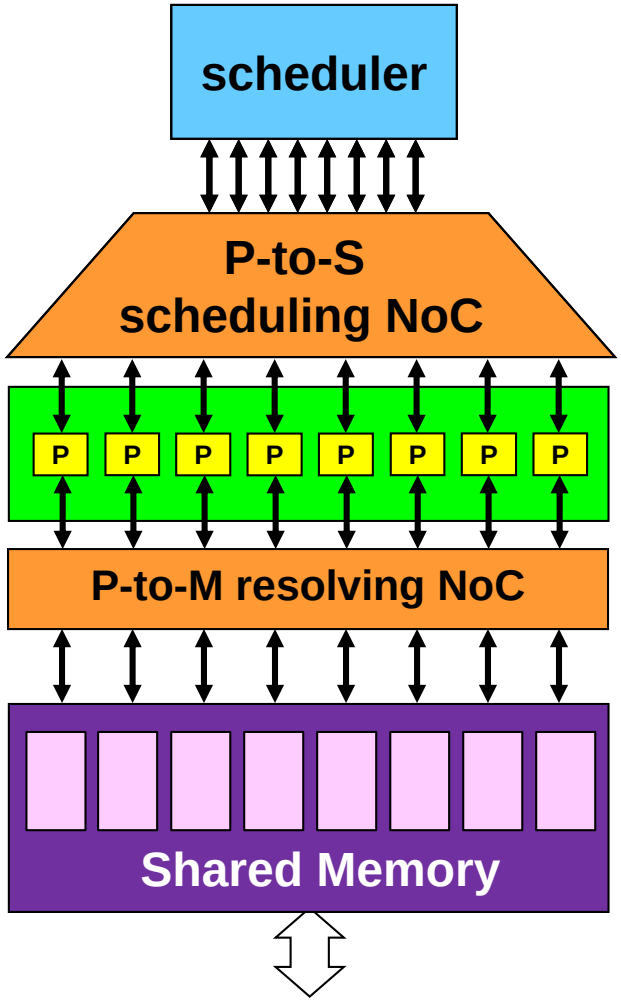
Many small processor cores
Small local memories (scratchpad, L1 caches)

Fast NOC to memory
(Multistage Interconnection Network)
NOC resolves conflicts

SHARED memory, many banks
~Equi-distant from cores (a few cycles)

“Anti-local” address interleaving
Negligible conflicts

RC64 conceptual architecture: part II



Hardware scheduler / dispatcher / synchronizer

Low (zero) latency parallel scheduling
enables fine granularity

Many small processor cores
Small local memories (scratchpad, L1 caches)

Fast NOC to memory
(Multistage Interconnection Network)
NOC resolves conflicts

SHARED memory, many banks
~Equi-distant from cores (a few cycles)

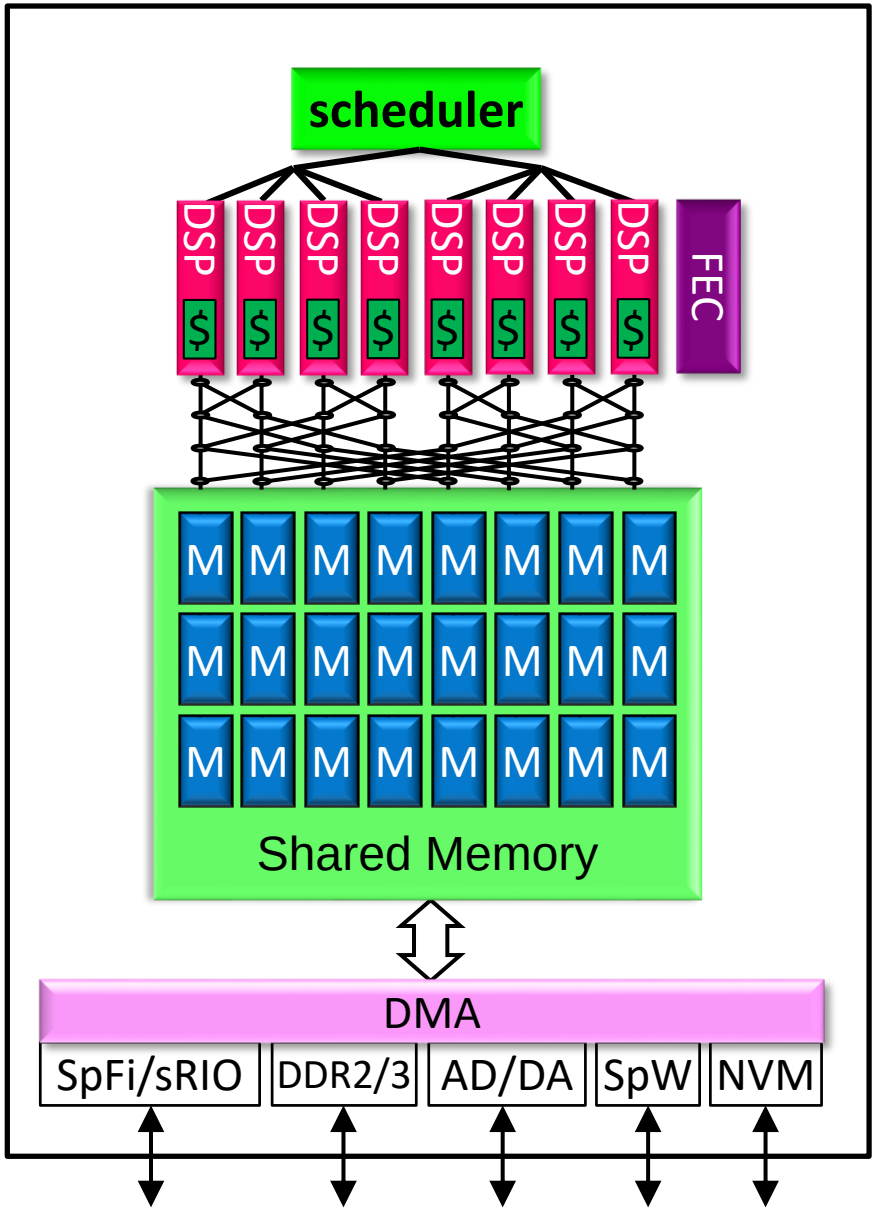
“Anti-local” address interleaving
Negligible conflicts

Outline

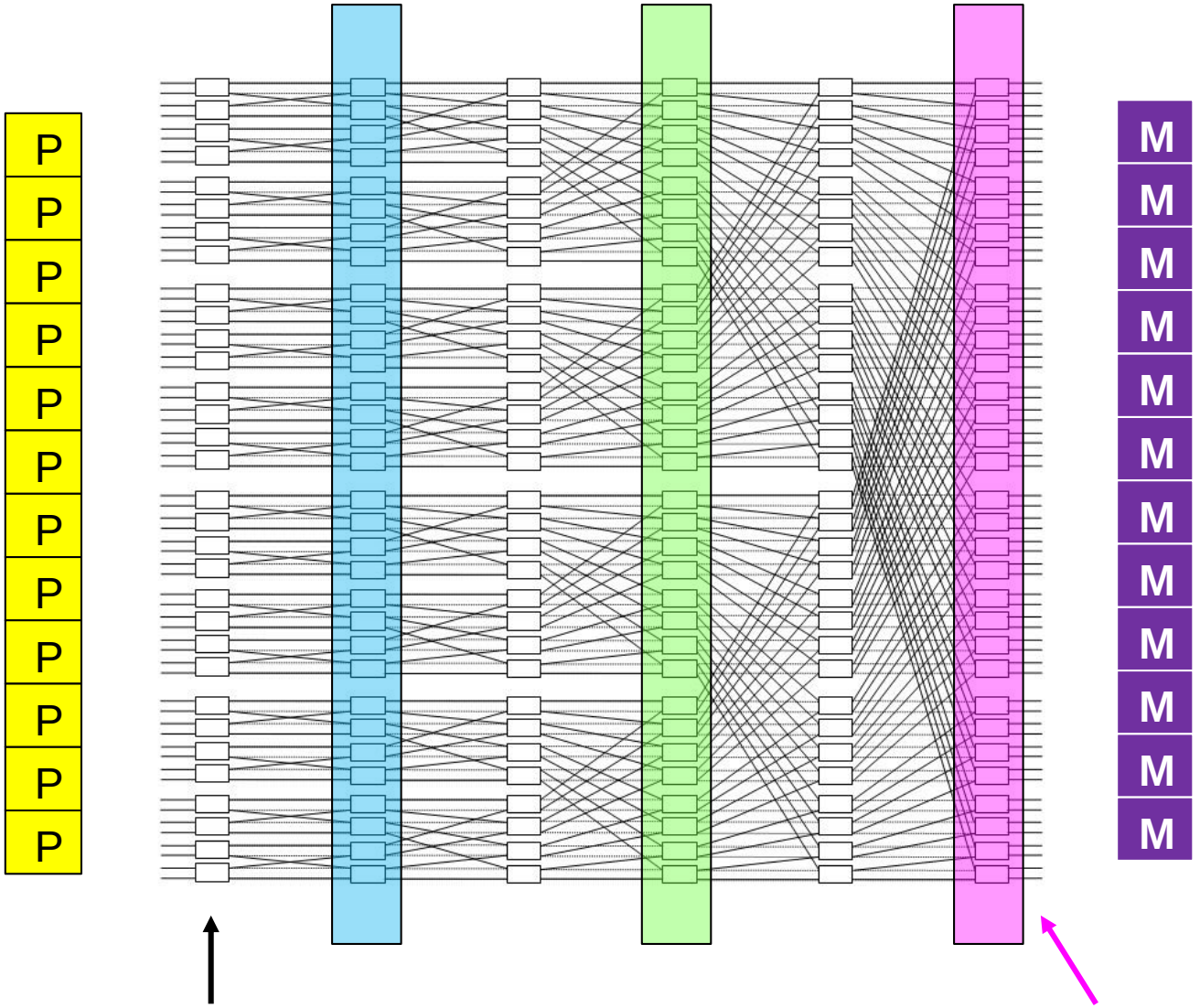
- Motivation: Programming model
- Architecture
- Implementation
- Programming model
- Algorithms

RC64

- 64 DSP cores
  - CEVA X1643
  - 300 MHz, 38 GFLOPS, 150 GOPS, 20 GIPS
  - SPM, I$, D$
- HW scheduler
- Modem HW accelerators
- 4 Mbyte shared memory
- Fast I/O
- Rad-Hard, FDIR
- 65nm LP TSMC
- Scalable up to 10 Watt
- PBGA & CCGA 624 (729)
- Designed for **SOFTWARE-DEFINED-SATELLITES**



Logarithmic multistage interconnection network



64 cores plus 30 DMA controllers

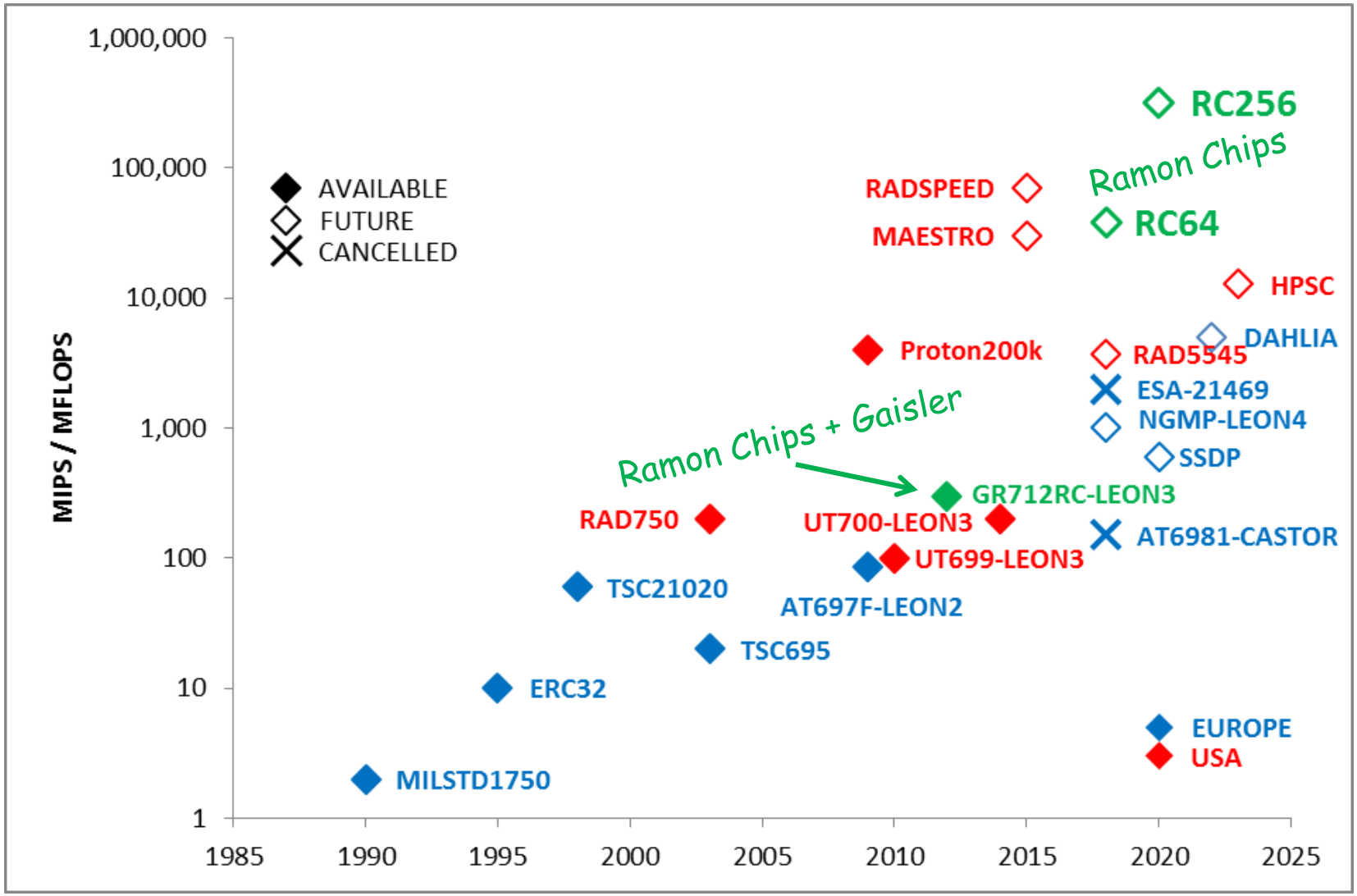
256 memory banks

Combinational switches

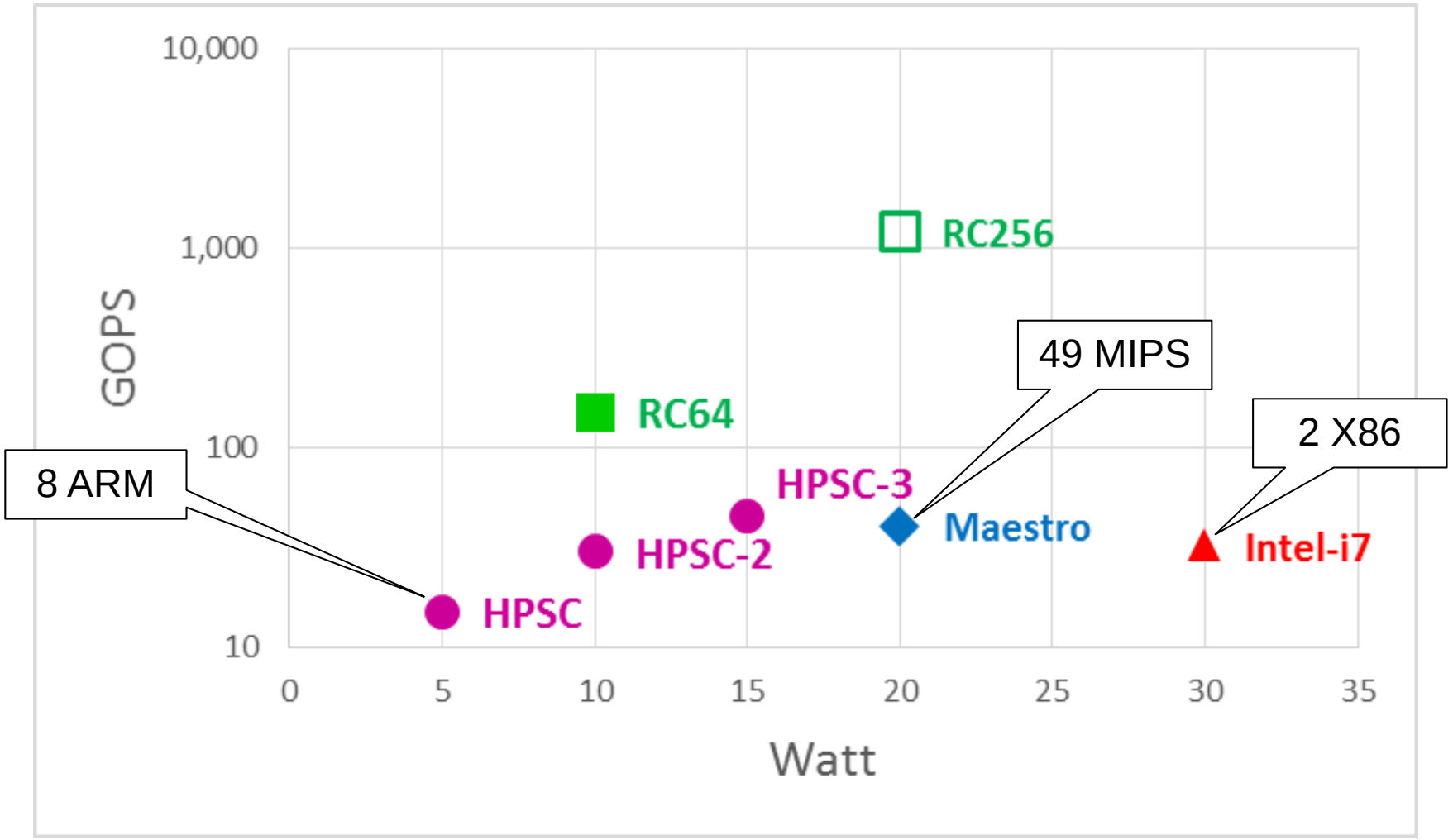
Pipeline stage (registers)

RC64 Floor plan: 64 DSP cores (24KB each) & 4MB shared memory take 352 mm^2 on 65nmLP

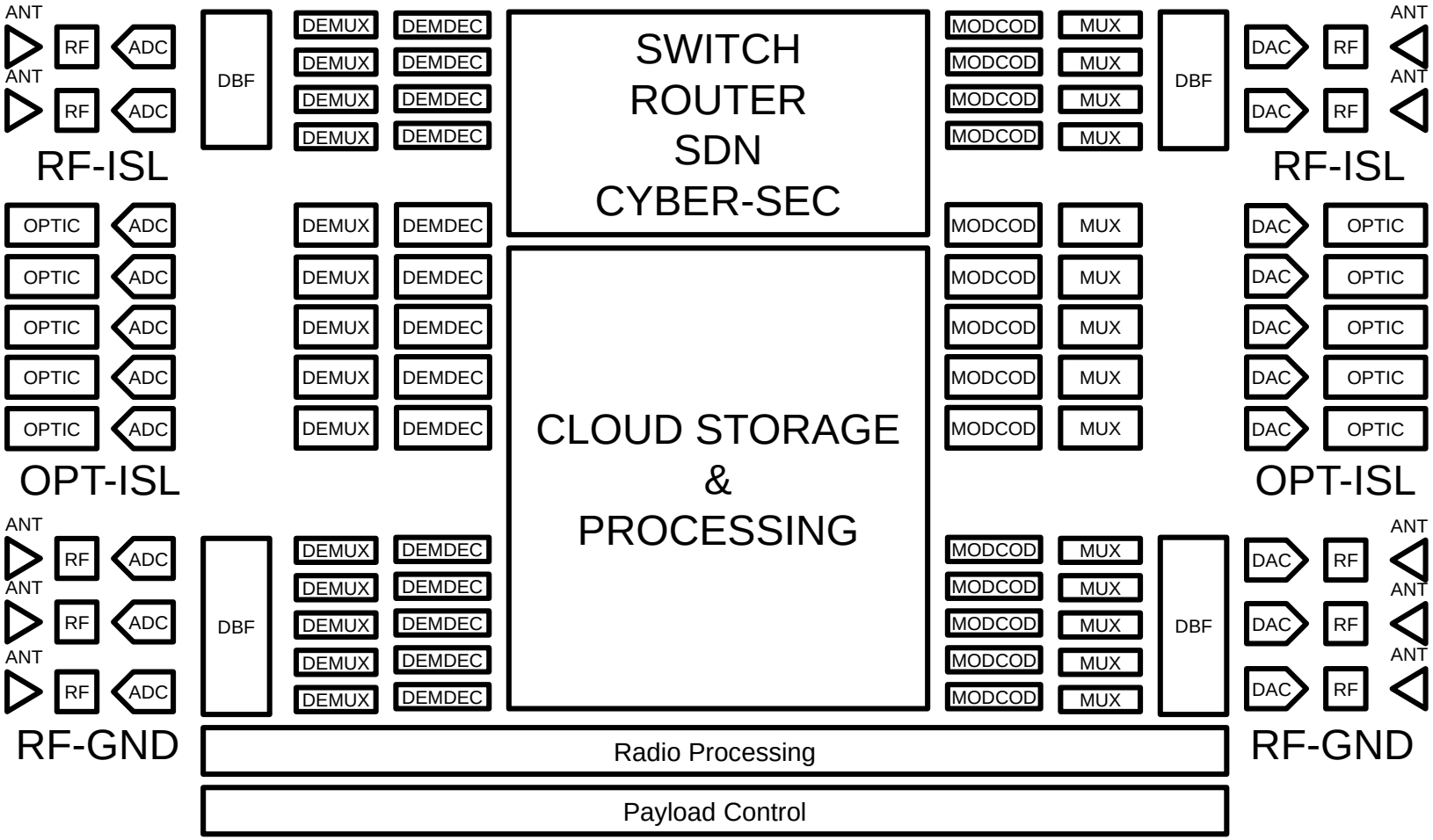
RC64 *vs* other space processors



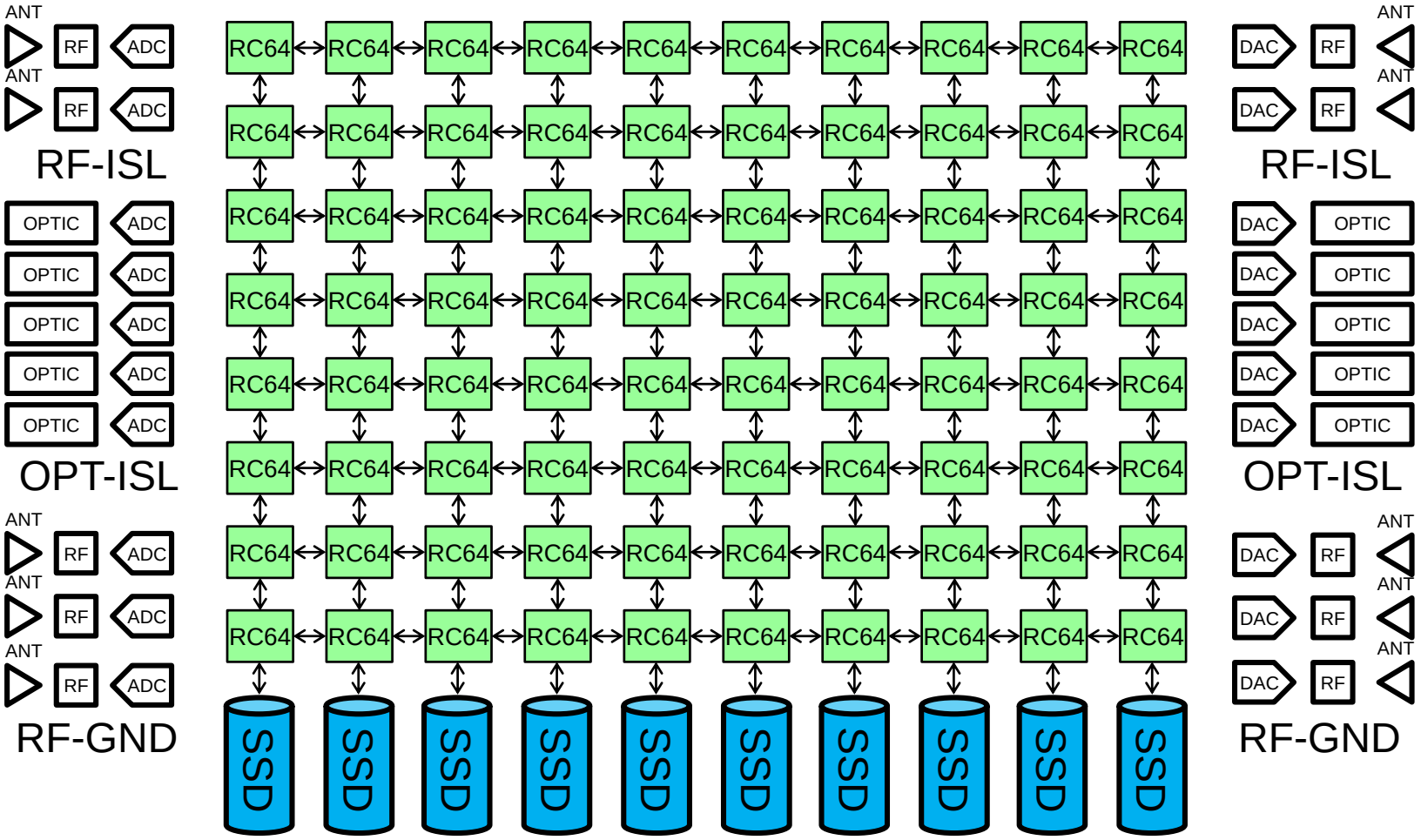
RC64 *vs* other space processors



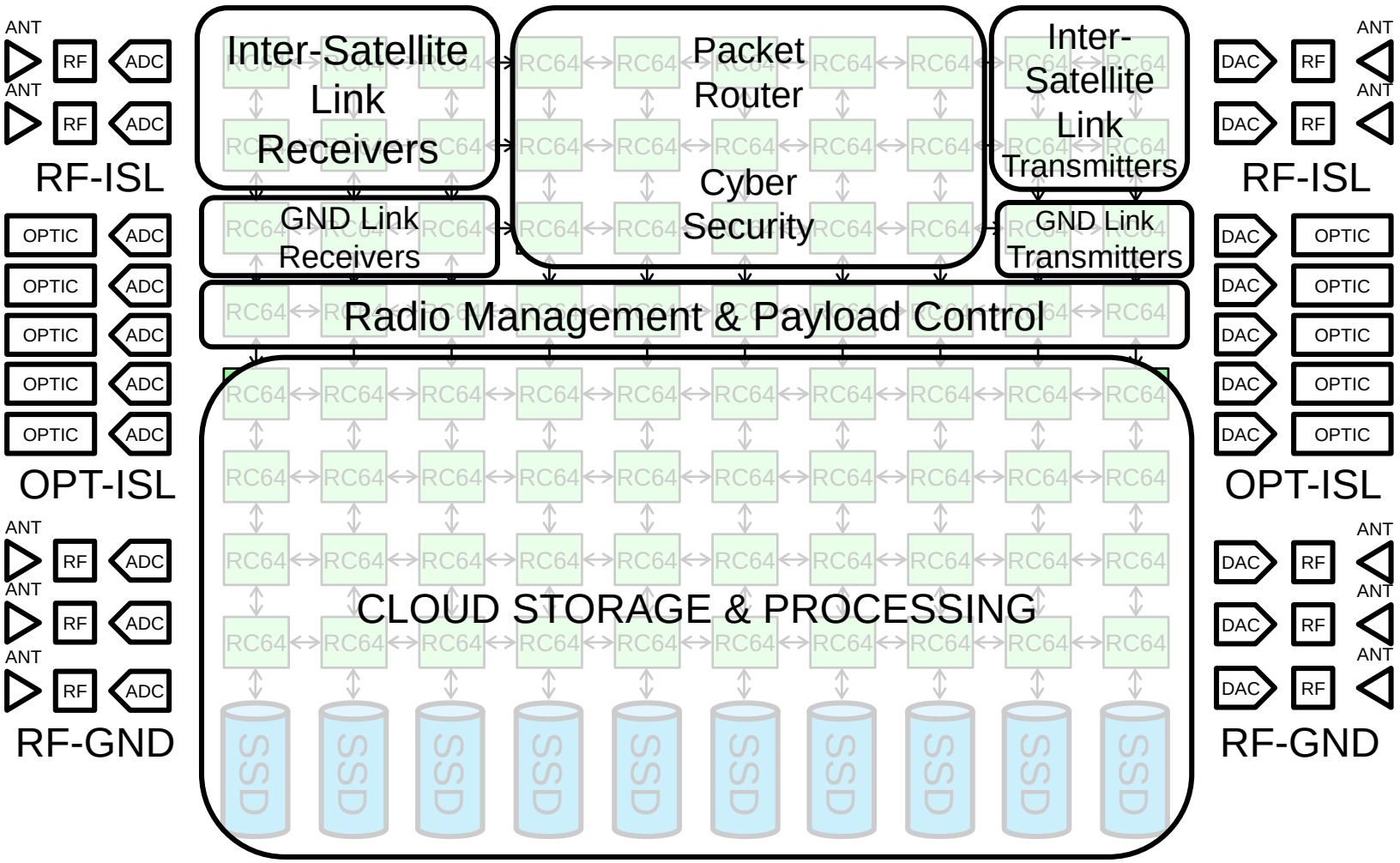
Many-RC64 system: comp/stor/comm satellite



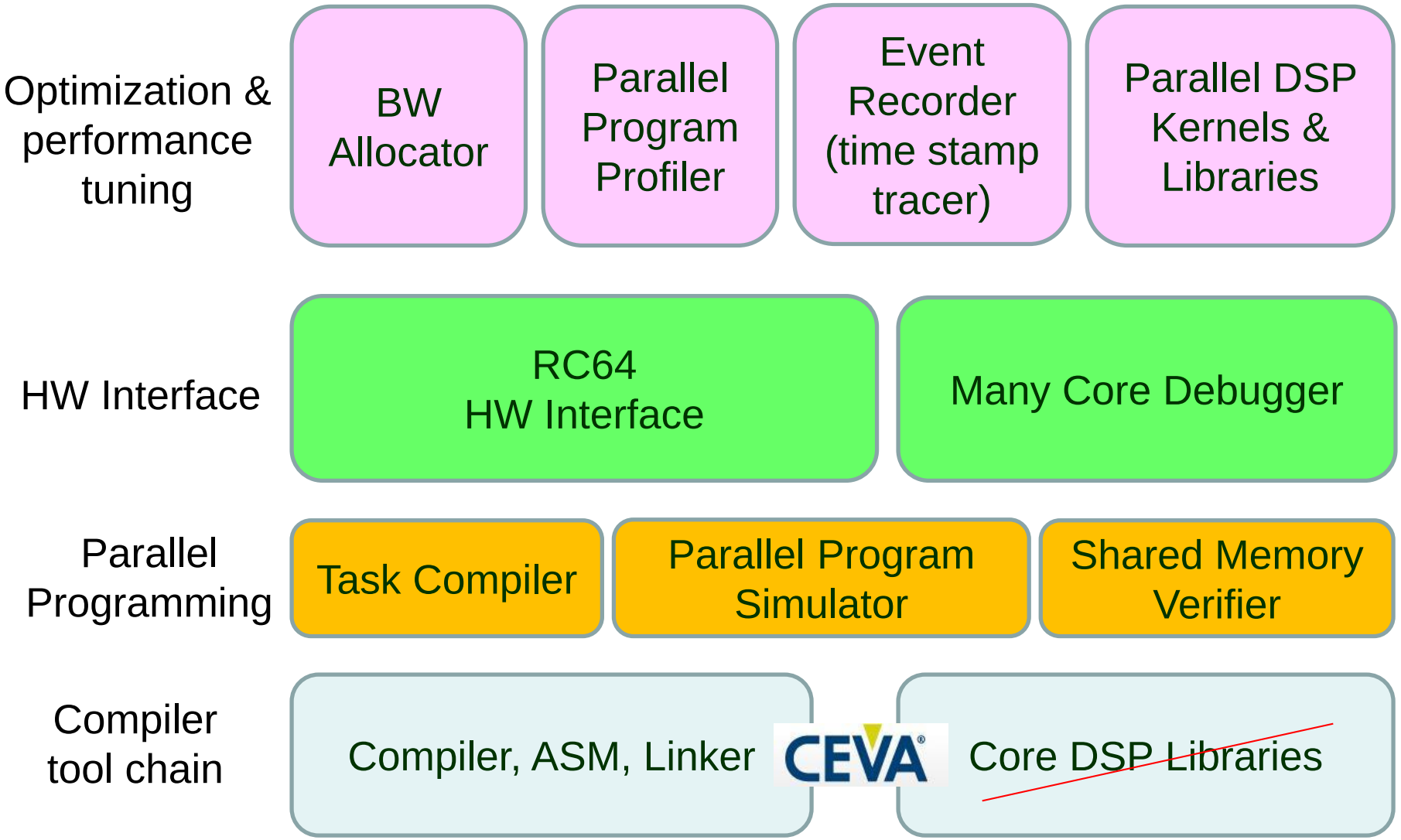
Many-RC64 system: comp/stor/comm satellite



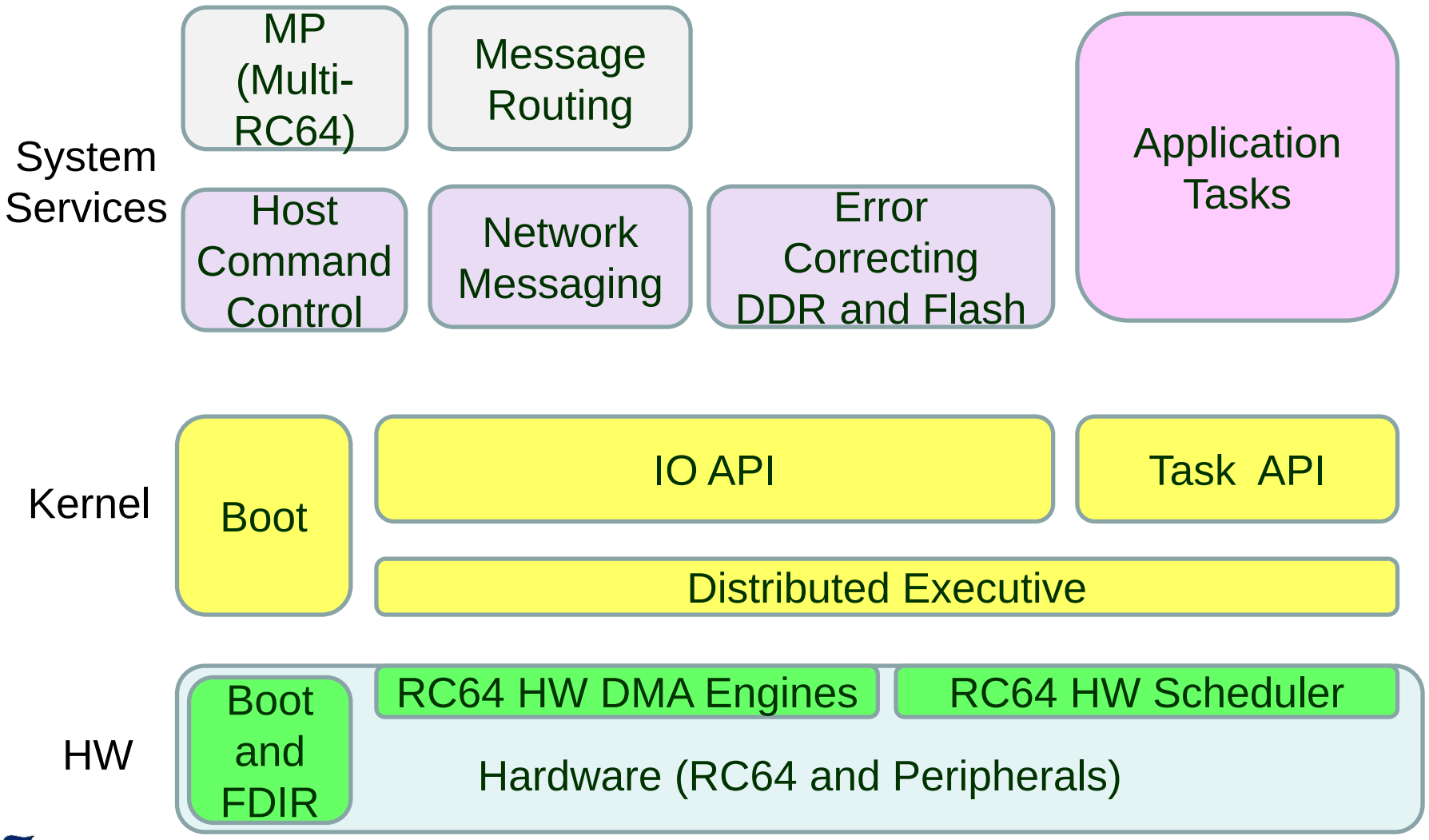
Many-RC64 system: comp/stor/comm satellite



RC64 SW Development Tools



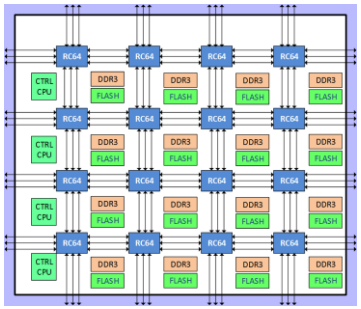
RC64 Run Time Model



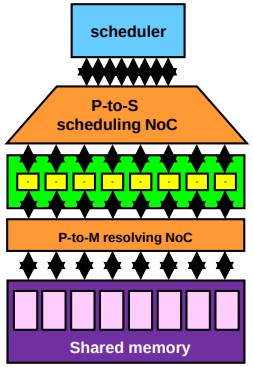
Outline

- Motivation: Programming model
- Architecture
- Implementation
- Programming model
- Algorithms

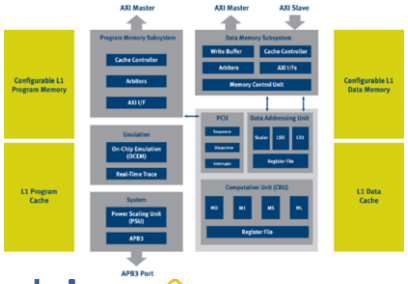
Three levels of “parallel” programming



- Multiple RC64 chips
  - Distributed computing (message passing)
  - OR: shared memory



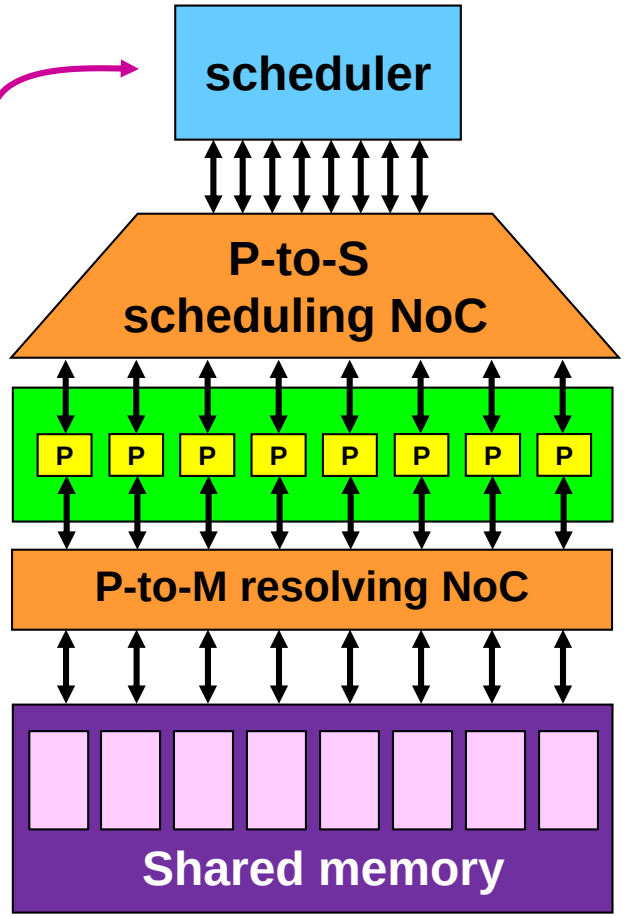
- One RC64 chip
  - 64 cores, shared memory



- A high performance core
  - VLIW + SIMD
  - NO VECTORIZATION

RC64 task-oriented programming model

- Programmer generates TWO parts:
  - Task-dependency-graph
  - Sequential task codes
- Task graph loaded into scheduler
- Tasks loaded into memory



Task template:

regular
duplicable taskName **(** instance_id **)**

{

... instance_id ....

// instance_id is instance number

.....

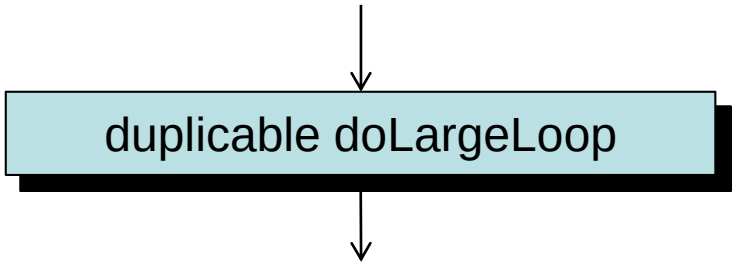
}

Fine Grain Parallelization

Convert (independent) loop iterations

(potential data parallelism)

```
for ( i=0; i<10000; i++ ) { a[i] = b[i]*c[i]; }
```



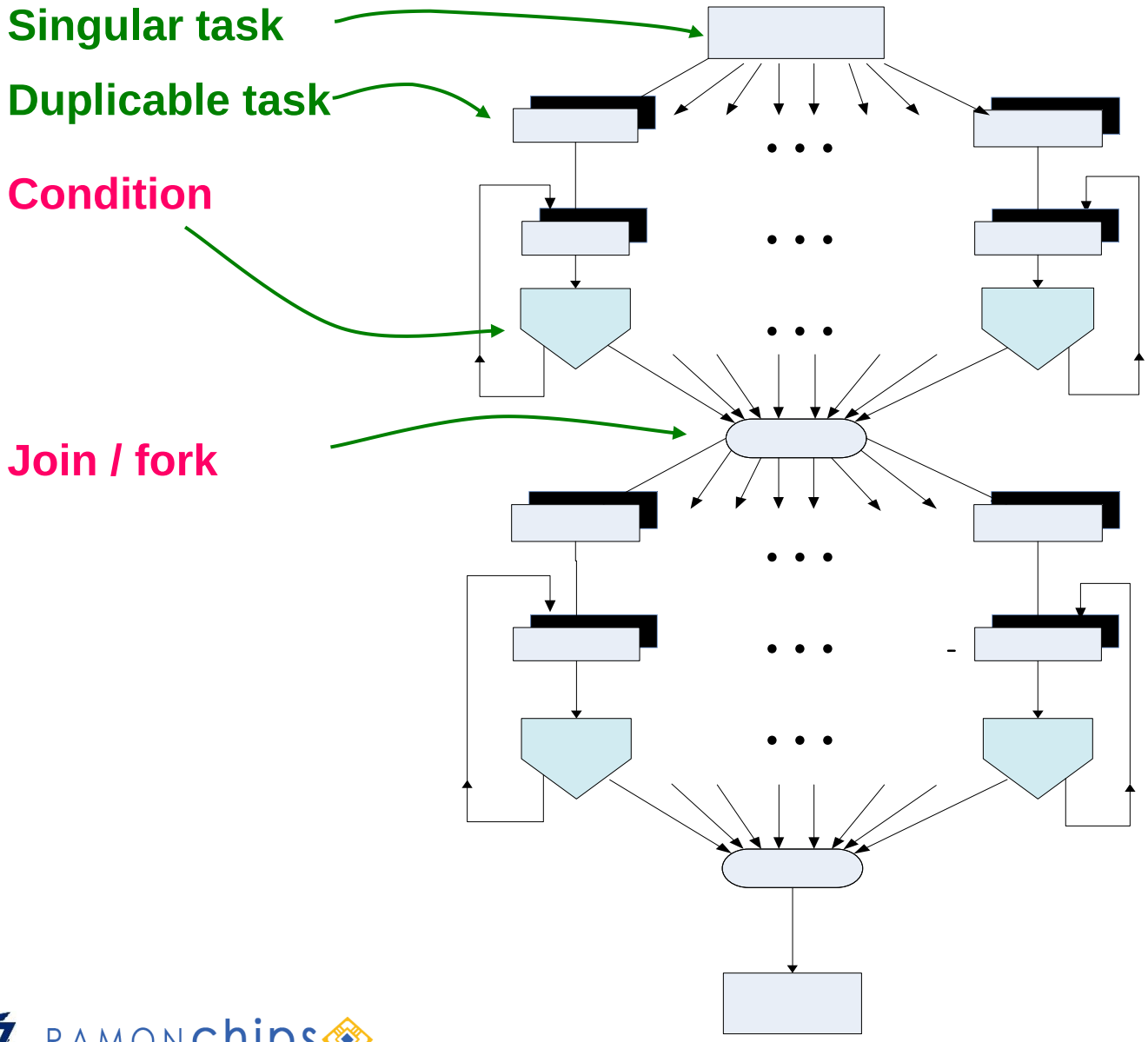
into parallel tasks

```
set_task_quota(doLargeLoop, 10000)
```

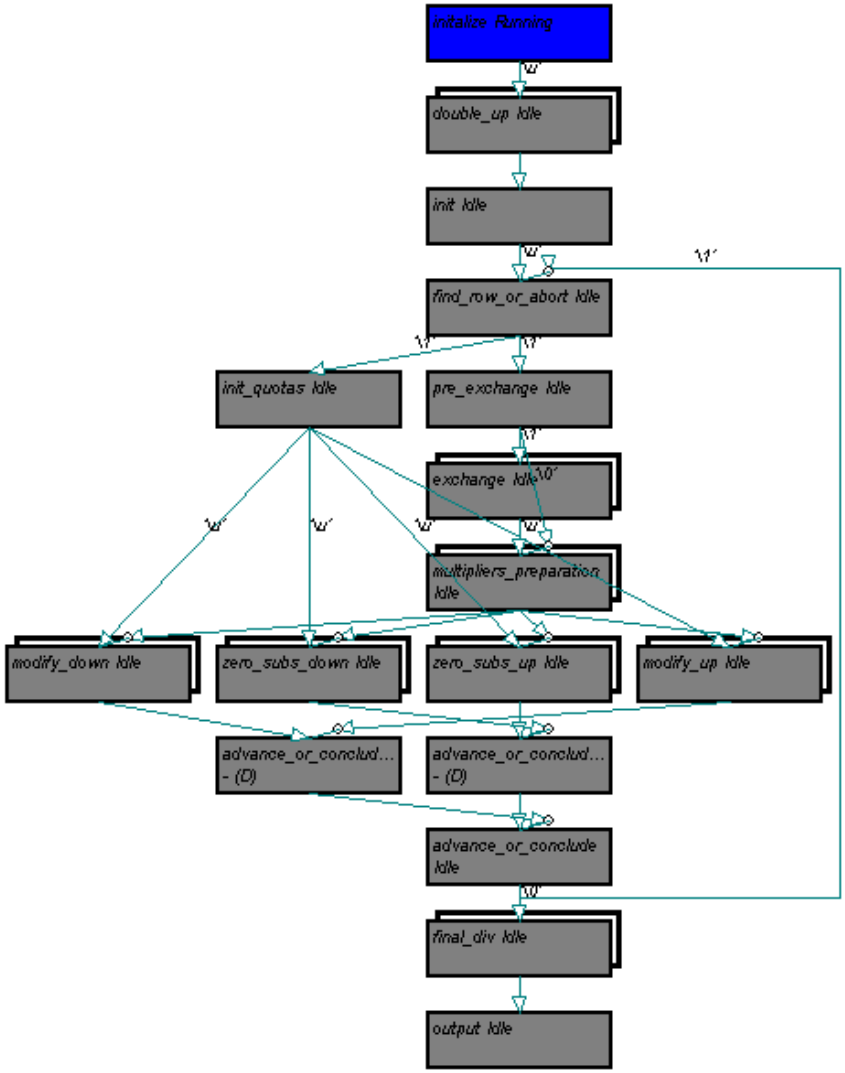
```
void doLargeLoop(unsigned int id)
{ a[id] = b[id]*c[id]; }  //id is instance number
```

(data parallelism → explicit task parallelism)

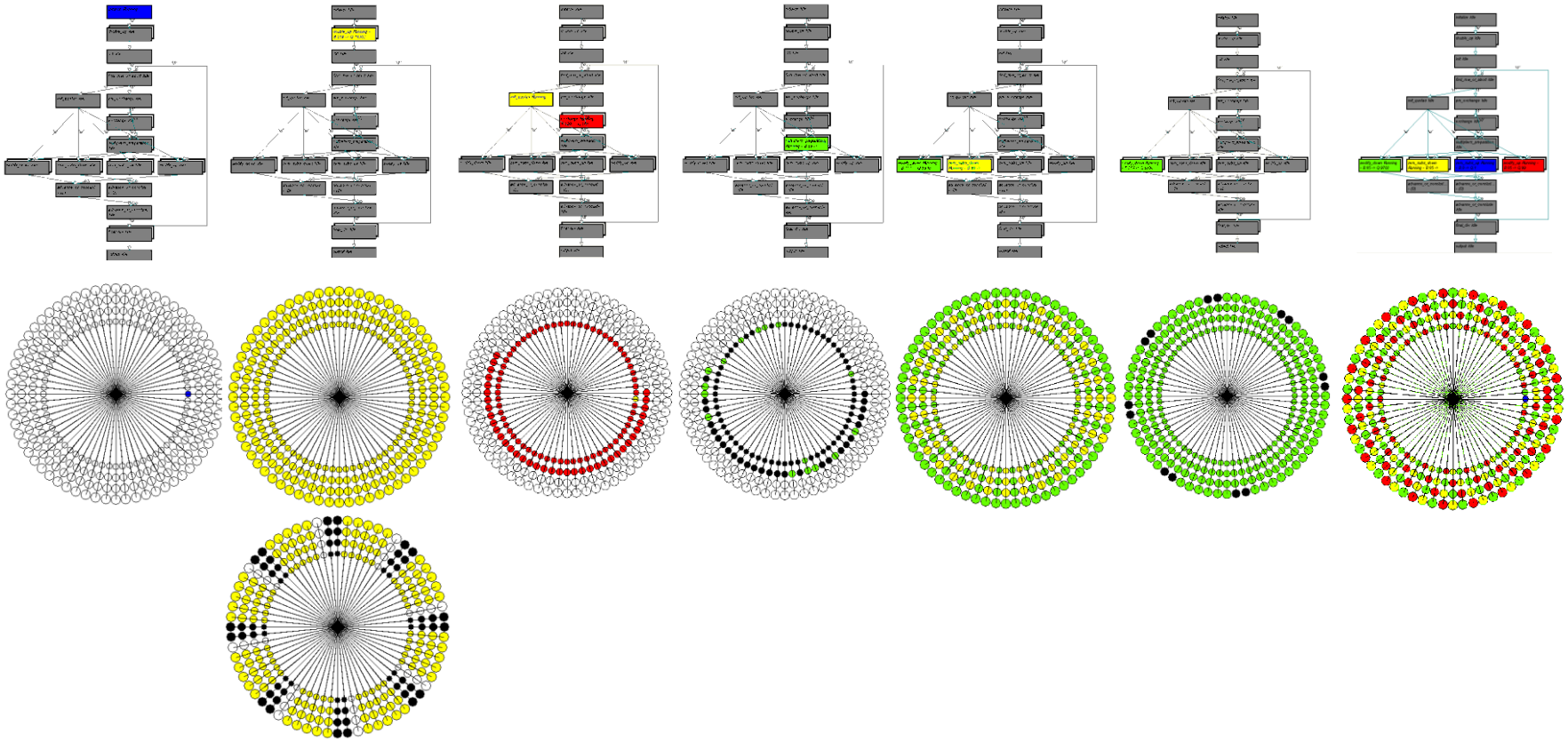
Task graph example (2D FFT)



Another task graph (linear solver)

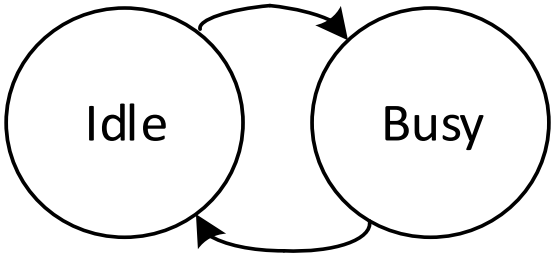


Linear Solver: Simulation snap-shots

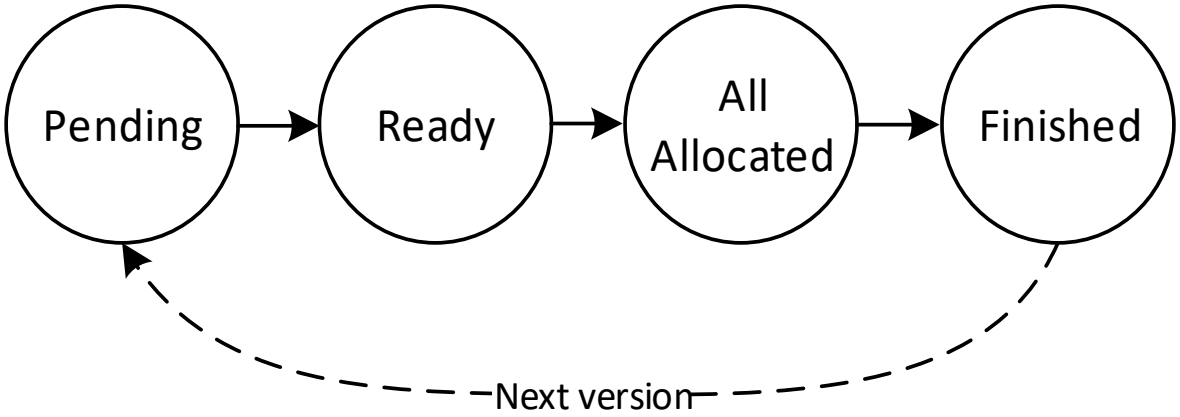


Cores and Tasks

Core



Task



Hardware Scheduler: Under the hood

core #

	state	task #	Instance #	…
0				
1				
2				

⋮

task #

	total instances	dependencies	state	# already allocated	…
0					
1					
2					

⋮

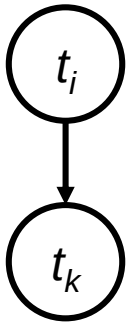
task graph

RC64 Task Rules

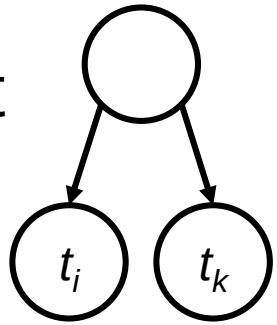
- Tasks are sequential
- All ready tasks, or any subset, can be executed in parallel on any number of cores
- All computing organized in tasks. All code lines belong to tasks
- Tasks use shared data in shared memory
  - May employ local private memory, BUT its contents disappear after task completion
- Nesting task spawning is easy and natural
- Conditions on tasks checked by scheduler
- Tasks are not functions
  - No arguments, no inputs, no outputs
- No synchronization points other than task completion
  - No locks, no BSP, no barriers
  - Sharing data is correct-by-construction

Concurrency in RC64

- Non-preemptive execution
- Task graph defines tasks and dependencies
- Task graph is executed by the scheduler
- ∃ path $t_i \rightarrow t_k \Rightarrow t_i$, t_k are non-concurrent
  - Execution of t_i must complete before start of execution of t_k



- Otherwise, t_i, t_k are concurrent
  - May execute simultaneously or at any order



- Task graph is decomposable into *concurrent sets*

(verifiable) Shared Memory Access Rules

1. Predictable Addressing
   - Shared memory addresses should be known at compile time
     - No data-dependent shared memory addresses
     - Predictable *malloc()* address
2. Exclusive Write (EW)
   - IF task t_i writes into *A*, *the compiler can verify that* no concurrent task t_k is allowed to access *A* (neither read nor write)
3. Concurrent Read (CR)
   - *The compiler can verify that* concurrent tasks may read from same address but none of them may write into it

Outline

- Motivation: Programming model
- Architecture
- Implementation
- Programming model
- Algorithms

Matrix Multiplication on RC64

$$C = A \times B$$

$$C_{i,j} = \sum_{m} A_{i,m} \times B_{m,j}$$

- Each result element $C_{i,j}$ is computed by a task
  - For N×N matrices, N×N tasks (regardless of #cores)
- Later, each task computes an entire row of C
  - Only N tasks

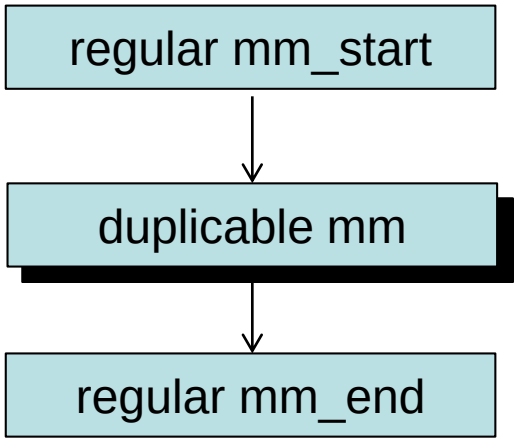
Matrix Multiplication on RC64

CODE (plain C)

```
#define MSIZE 100
float  A[MSIZE][MSIZE],      B[MSIZE][MSIZE],
       C[MSIZE][MSIZE];
int  mm_start()                              REGULAR
{    int i,j;
     for (i=0; i< MSIZE; i++)
          for (j=0; j< MSIZE; j++)
               { A[i][j] = 13;  B[i][j] = 9;   }
}
void  mm (unsigned int id)          DUPLICABLE
{    int  i,j,m;  float sum = 0;
     i = id % MSIZE;      j = id / MSIZE;
     for (m=0; m < MSIZE; m++)
          sum += A[i][m]*B[m][j];
     C[i][j]=sum;
}
int  mm_end ()                               REGULAR
{  printf("finished mm\n");  }
```

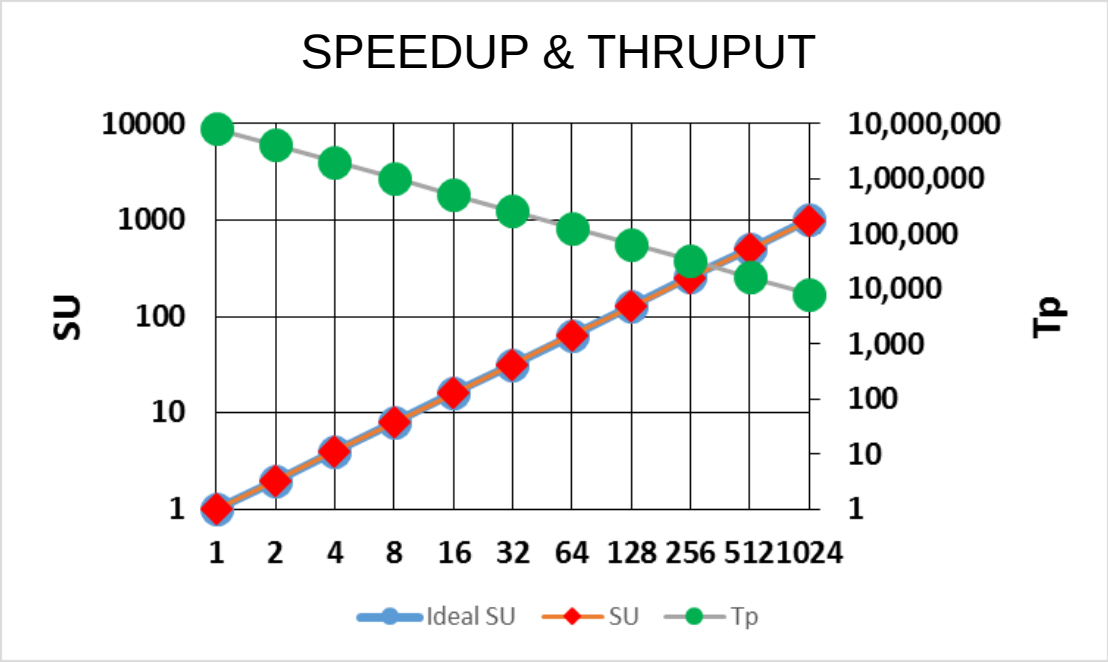
TASK GRAPH

```
#define MSIZE 100
#define MMSIZE 10000

regular       mm_start()
duplicable  mm(mm_start) MMSIZE
regular      mm_end(mm)
```



Matrix Multiplication on RC64

P	Tp	SU	Eff
1	8,190,021	1	1.00
2	4,095,021	2	1.00
4	2,047,521	4	1.00
8	1,023,771	8	1.00
16	511,896	16	1.00
32	256,368	32	1.00
64	128,604	64	1.00
128	64,722	127	0.99
256	32,781	250	0.98
512	16,401	499	0.98
1024	8,211	997	0.97



- Why is SU(1024) still less than 1024?

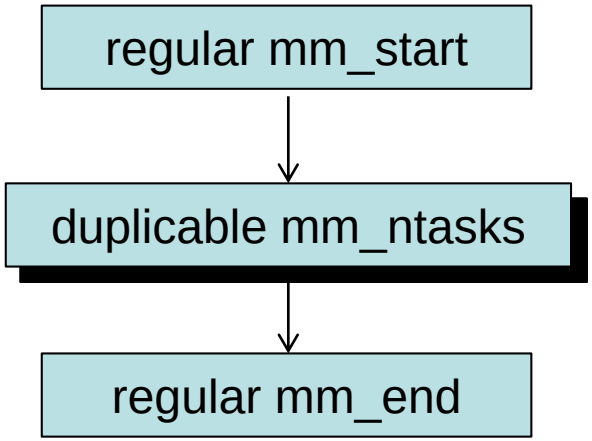
Matrix Multiplication using only N tasks

CODE (plain C)

```
#define MSIZE 100
float  A[MSIZE][MSIZE],   B[MSIZE][MSIZE],
       C[MSIZE][MSIZE];
int     mm_start ()                    REGULAR
{   int i,j;
    for (i=0; i< MSIZE; i++)
        for (j=0; j< MSIZE; j++)
            { A[i][j] = 13;  B[i][j] = 9;   }
}
void mm_ntasks (unsigned int id)          DUP
{   int  m, k; float sum = 0;
    for (k=0; k<MSIZE; k++) {
        sum = 0;
        for (m=0; m < MSIZE; m++)
            sum += A[id][m]*B[m][k];
        C[id][k]=sum;
    }
}
int     mm_end ()                      REGULAR
{   printf("finished mm with N tasks\n");  }
```

TASK GRAPH

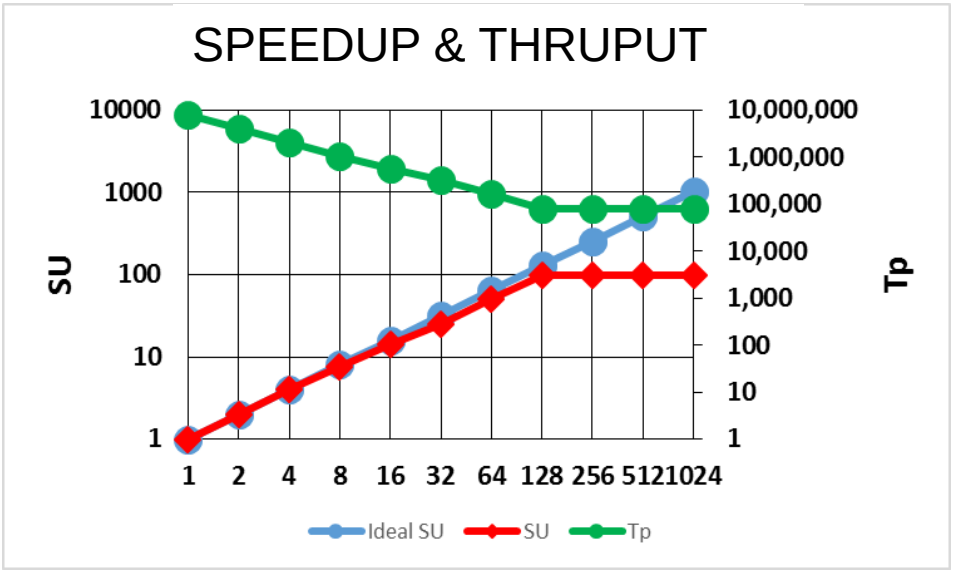
```
#define MSIZE 100

regular mm_start()
duplicable  mm_ntasks(mm_start) MSIZE
regular  mm_end(mm_ntasks)
```

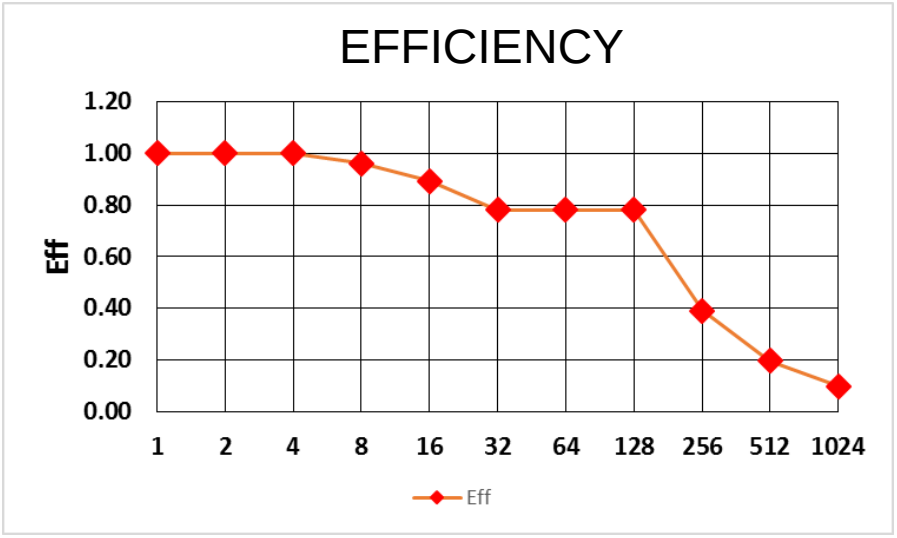


Matrix Multiplication using only N tasks

P	Tp	SU	Eff
1	8,140,021	1	1.00
2	4,070,021	2	1.00
4	2,035,021	4	1.00
8	1,058,221	8	0.96
16	569,821	14	0.89
32	325,621	25	0.78
64	162,821	50	0.78
128	81,421	100	0.78
256	81,421	100	0.39
512	81,421	100	0.20
1024	81,421	100	0.10

- What went wrong ?





SW development flow: MATLAB to RC64

1. MATLAB float, unrestricted (also SIMULINK)
2. MATLAB float, restricted memory size and I/O
3. MATLAB fixed point 16-bit
   - Insert DSP library functions
   - Create Golden model
4. Convert to C
   - Sequential code on laptop
   - Bit-exact comparison to Golden model
5. Parallelize for RC64 many-core. Create task graph
   - Simulate using “many-task emulator” on laptop
   - Bit-exact comparison to Golden model
6. Transfer to RC64
   - Execute on hardware, or
   - Simulate using cycle-accurate RC64 simulator
   - Bit-exact comparison to Golden model

Advantages of RC64 architecture

- Shared, uniform (~equi-distant) memory
  - no worry which core does what
  - no advantage to any core because it already holds the data
- Many-bank memory + fast P-to-M NoC
  - low latency
  - no bottleneck accessing shared memory
- Fast scheduling of tasks to free cores (many at once)
  - enables fine grain data parallelism
- Any core can do any task equally well on short notice
  - scales well
- Programming model:
  - Intuitive to programmers
  - CREW verifiable
  - Simple model facilitates parallelizing compiler

Summary

- Simple manycore architecture
  - Inspired by PRAM
- Hardware scheduling
- Task-based programming model
  - Fine grain tasks
  - #instances >> #cores
  - Many-Flow
- Designed to achieve the goal of ‘more cores, less power’
- RC64 implementation
- Potentially scalable efficiently to 256, 1024 cores