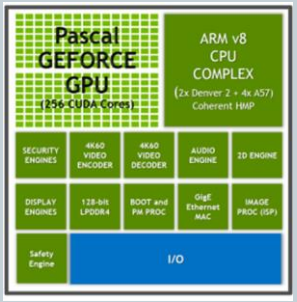
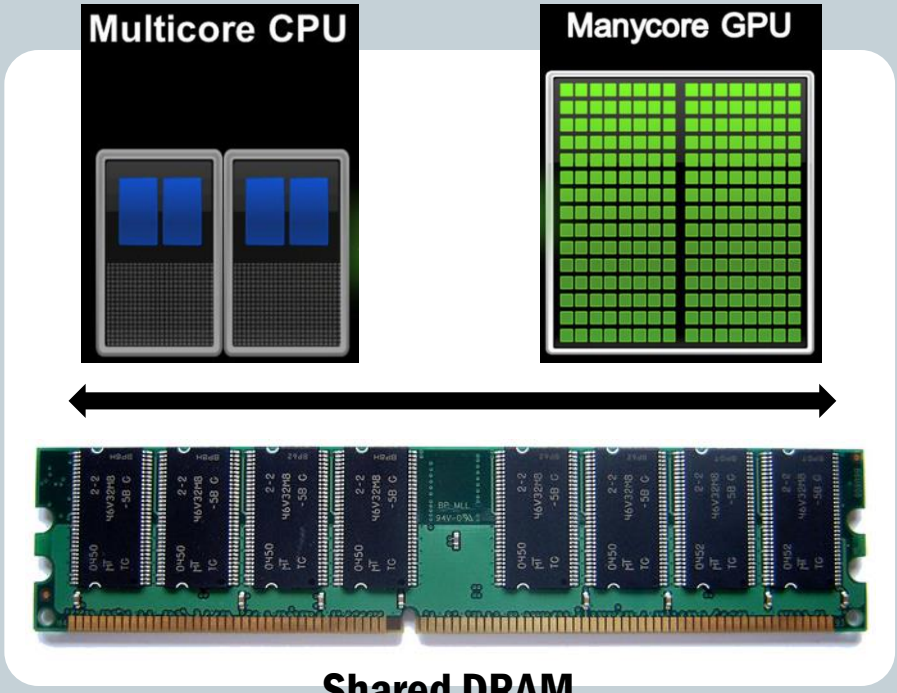
# Towards a Predictable Execution Model for Heterogeneous Systems-on-a-Chip

ANDREA MARONGIU
Università Di Bologna
a.marongiu@unibo.it

# Background

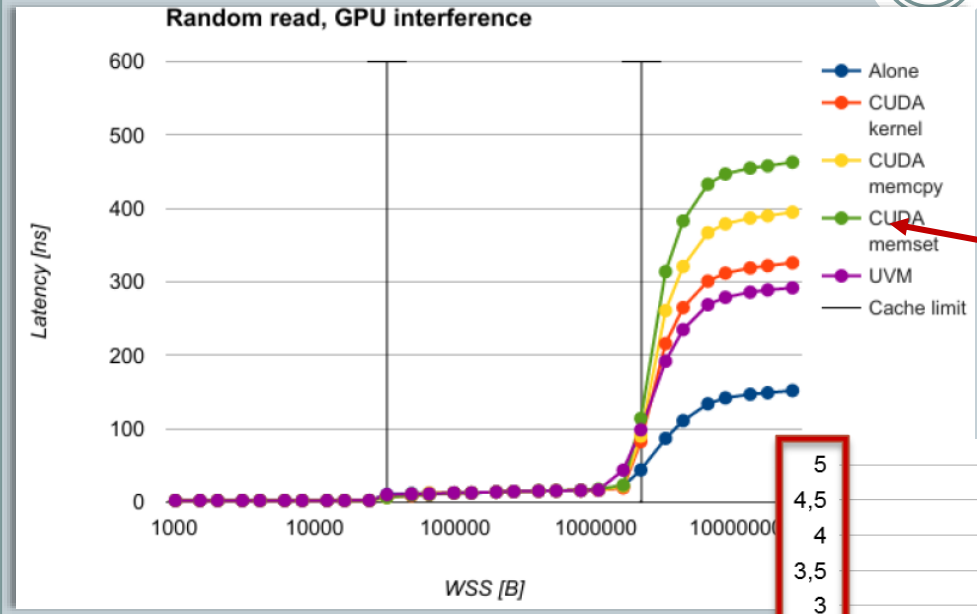





**Strong push for unified memory model in heterogeneous SoCs**

- **Good for programmability**
- **Optimized to reduce performance loss**
- **How about predictability?**

# How large can the interference in execution time among the two subsystems be?

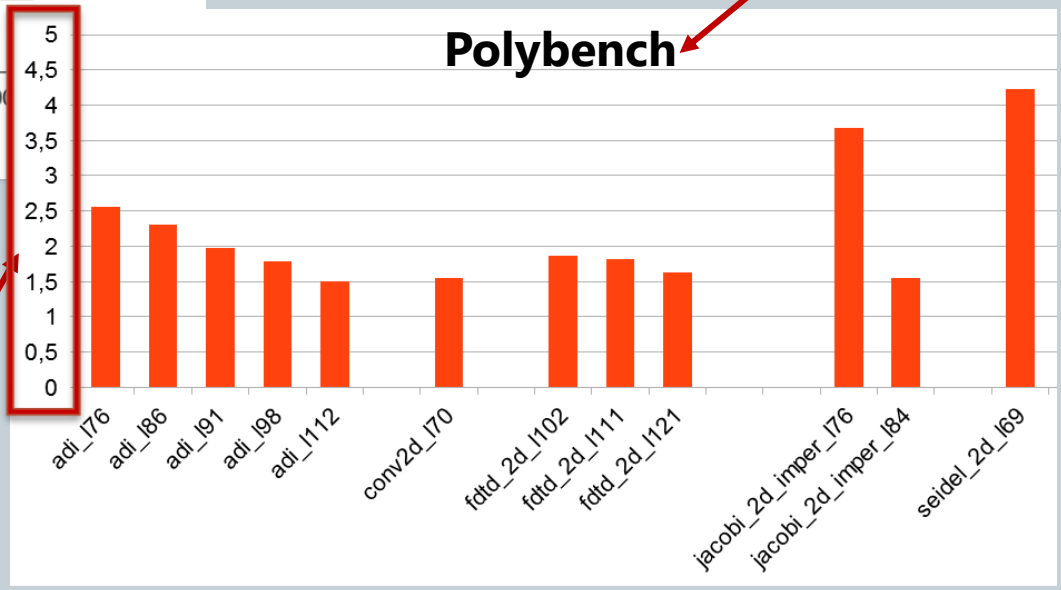



**NVIDIA Tegra X1**

**GPU execution**

**A mix of synthetic workload and real bechmarks**

**Polybench**




**CPU execution**

**Synthetic workload to model high-interference**

slowdown VS
GPU execution in isolation

# How large can the interference in execution time among the two subsystems be?

## HOW ABOUT REAL WORKLOADS?

- **Rodinia benchmarks executing on both the GPU and the CPU (on all 4 A57, one is observed)**
- **Up to 2.5x slower execution under mutual interference**

**NVIDIA Tegra X1**

| CPU/GPU | euler3d | sc_gpu | lud_cuda | srad_v1 | srad_v2 | heartwall | leukocyte | needle | backprop | kmeans | particlefilter_float | pathfinder | lavaMD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| srad_v1 | 1,7 | 1,5 | 1,2 | 1,4 | 1,4 | 1,2 | 1,2 | 1,2 | 1,4 | 1,6 | 1,2 | 1,2 | 1,2 |
| srad_v2 | 1,2 | 1,4 | 1,1 | 1,3 | 1,2 | 1,1 | 1,1 | 1,1 | 1,3 | 1,5 | 1,1 | 1,2 | 1,1 |
| needle | 1,6 | 2,0 | 1,3 | 1,9 | 1,9 | 1,3 | 1,3 | 1,3 | 1,9 | 2,5 | 1,3 | 1,7 | 1,3 |
| euler3d_cpu | 1,2 | 1,4 | 1,1 | 1,1 | 1,3 | 1,1 | 1,1 | 1,1 | 1,3 | 1,5 | 1,1 | 1,2 | 1,1 |
| streamcluster | 1,5 | 1,9 | 1,3 | 1,3 | 1,8 | 1,3 | 1,3 | 1,3 | 2,0 | 2,0 | 1,3 | 1,4 | 1,3 |
| lud_omp | 1,1 | 1,3 | 1,0 | 1,2 | 1,2 | 1,1 | 1,0 | 1,0 | 1,2 | 1,4 | 1,0 | 1,1 | 1,0 |
| heartwall | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,1 | 1,0 |
| particle_filter | 1,2 | 1,2 | 1,2 | 1,3 | 1,1 | 1,1 | 1,2 | 1,1 | 1,1 | 1,1 | 1,1 | 1,3 | 1,1 |
| mummergpu | 1,6 | 2,1 | 1,3 | 1,4 | 1,9 | 1,4 | 1,4 | 1,3 | 2,0 | 2,5 | 1,4 | 1,7 | 1,4 |
| bfs | 2,1 | 1,7 | 1,6 | 1,9 | 1,8 | 1,7 | 1,5 | 1,7 | 1,6 | 1,6 | 1,5 | 1,8 | 1,7 |
| backprop | 1,9 | 2,4 | 1,8 | 2,0 | 2,4 | 1,5 | 1,6 | 1,5 | 2,3 | 2,5 | 1,7 | 2,0 | 1,6 |
| nn | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 |
| hotspot | 1,5 | 1,5 | 1,5 | 1,5 | 1,7 | 1,5 | 1,7 | 1,7 | 1,5 | 1,8 | 1,6 | 1,5 | 1,5 |
| hotspot3D | 1,5 | 2,0 | 1,5 | 1,6 | 1,9 | 1,4 | 1,5 | 1,9 | 2,4 | 1,9 | 1,6 | 1,8 | 1,6 |
| kmeans | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,0 |
| lavaMD | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 |
| myocyte.out | 1,1 | 1,1 | 1,0 | 1,1 | 1,1 | 1,0 | 1,1 | 1,0 | 1,1 | 1,2 | 1,1 | 1,1 | 1,1 |
| pathfinder | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 |
| b+tree | 1,0 | 1,1 | 1,0 | 1,1 | 1,1 | 1,0 | 1,0 | 1,0 | 1,1 | 1,1 | 1,0 | 1,1 | 1,0 |
| leukocyte | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 | 1,1 |
| hotspot | 1,7 | 1,6 | 1,6 | 1,6 | 1,6 | 1,7 | 1,6 | 1,7 | 1,6 | 1,6 | 1,7 | 1,5 | 1,7 |

# How large can the interference in execution time among the two subsystems be?

## HOW ABOUT REAL WORKLOADS?

- **Rodinia benchmarks executing on both the GPU (observed values) and the CPU (on all 4 A57)**
- **Up to 33x slower execution under mutual interference**

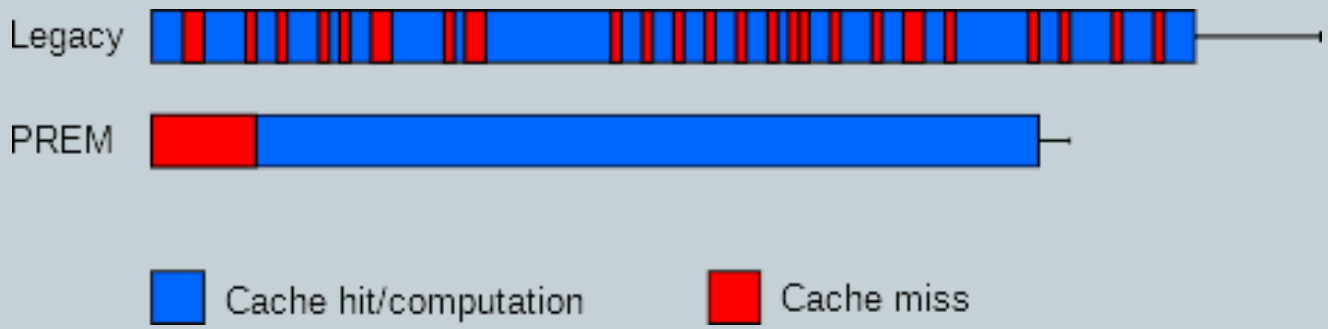
**NVIDIA Tegra X1**

| GPU / CPU | srad_v1 | srad_v2 | needle | euler3d_cpu | sc_omp | lud_omp | heartwall | particle_filter | mummergpu | bfs | backprop | filelist_512k | hotspot | 3D | kmeans_serial | lavaMD | myocyte.out | pathfinder | b+tree.out | leukocyte | hotspot | classification |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| euler3d | 1,2 | 1,2 | 1,3 | 1,1 | 1,2 | 1,2 | 1,0 | 1,0 | 1,2 | 1,1 | 1,4 | 1,0 | 1,0 | 1,1 | 1,1 | 1,0 | 1,1 | 1,1 | 1,0 | 1,0 | 1,0 | 1,2 |
| lud_cuda | 1,5 | 1,5 | 1,3 | 1,1 | 1,2 | 1,1 | 1,0 | 1,0 | 2,2 | 1,1 | 1,6 | 1,0 | 1,0 | 1,2 | 1,1 | 1,0 | 1,2 | 1,1 | 1,0 | 1,0 | 1,0 | 2,4 |
| srad_v1 | 1,2 | 5,2 | 1,3 | 1,3 | 1,4 | 1,2 | 1,1 | 1,3 | 1,4 | 1,6 | 1,5 | 1,2 | 1,2 | 1,2 | 1,2 | 1,1 | 1,1 | 1,2 | 1,1 | 1,1 | 1,1 | 3,1 |
| srad_v2 | 1,4 | 8,9 | 1,5 | 1,1 | 1,1 | 1,9 | 1,0 | 6,5 | 1,3 | 3,6 | 5,6 | 1,2 | 1,8 | 1,1 | 1,1 | 1,0 | 1,2 | 1,5 | 1,3 | 1,0 | 2,9 | 3,6 |
| heartwall | 1,1 | 1,4 | 1,2 | 1,0 | 1,1 | 1,1 | 1,0 | 1,1 | 1,8 | 1,0 | 1,4 | 1,0 | 1,0 | 1,1 | 1,0 | 1,0 | 1,2 | 1,1 | 1,1 | 1,0 | 1,1 | 1,2 |
| leukocyte | 1,4 | 1,5 | 1,6 | 1,0 | 1,0 | 1,0 | 1,0 | 1,6 | 2,6 | 1,2 | 1,6 | 1,3 | 1,2 | 1,1 | 1,0 | 1,1 | 1,3 | 1,1 | 1,1 | 1,0 | 1,4 | 2,3 |
| needle | 2,8 | 4,5 | 2,9 | 2,6 | 2,7 | 2,4 | 1,7 | 13,0 | 4,6 | 5,2 | 2,9 | 3,8 | 6,4 | 2,6 | 2,4 | 2,4 | 3,2 | 2,6 | 2,5 | 1,9 | 4,1 | 14,2 |
| backprop | 3,6 | 32,7 | 2,6 | 2,6 | 2,6 | 16,3 | 2,0 | 3,6 | 9,2 | 2,1 | 3,8 | 7,7 | 2,0 | 2,2 | 2,6 | 2,0 | 3,6 | 2,2 | 2,1 | 2,0 | 2,1 | 2,5 |
| kmeans | 19,3 | 21,0 | 14,4 | 1,2 | 2,2 | 4,4 | 1,0 | 29,8 | 4,2 | 28,9 | 22,7 | 9,5 | 19,9 | 2,9 | 4,4 | 1,0 | 1,9 | 15,7 | 5,4 | 1,2 | 8,3 | 3,8 |
| particlefilter_float | 1,2 | 1,3 | 1,3 | 1,1 | 1,1 | 1,1 | 1,0 | 1,3 | 1,8 | 1,1 | 1,2 | 1,1 | 1,5 | 1,1 | 1,1 | 1,0 | 1,2 | 1,1 | 1,0 | 1,1 | 1,0 | 1,1 |
| pathfinder | 8,0 | 19,3 | 8,2 | 1,1 | 7,5 | 10,5 | 6,4 | 9,3 | 4,1 | 27,2 | 9,2 | 12,1 | 5,4 | 9,6 | 3,4 | 1,6 | 7,4 | 10,8 | 7,8 | 1,1 | 8,2 | 27,7 |
| lavaMD | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,2 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,0 | 1,2 |

# The predictable execution model (PREM)

- **Predictable interval**
  - Memory prefetching in the first phase
  - No cache misses in the execution phase
  - Non-preemptive execution

**Requires compiler support for code re-structuring**

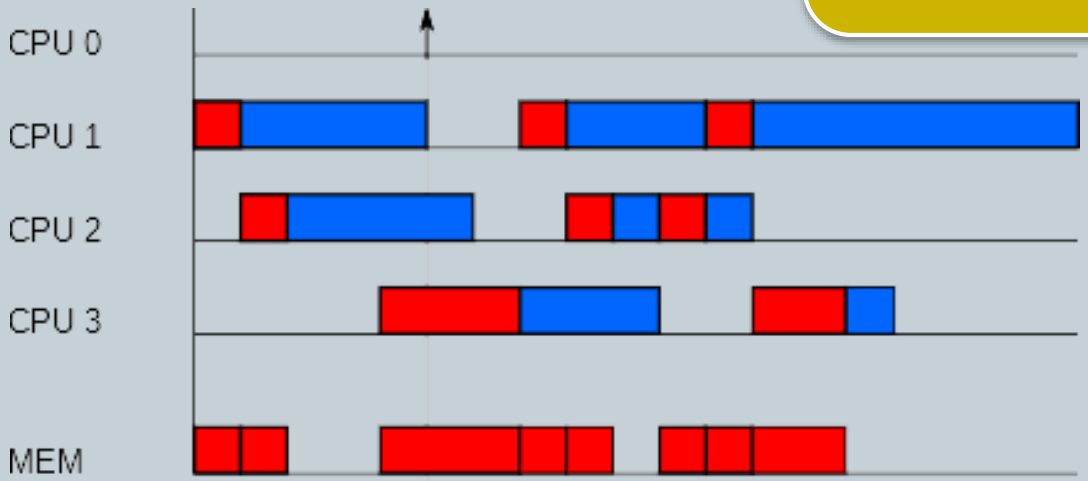



**Originally proposed for (multi-core) CPU.**
**We study the applicability of this idea to heterogeneous SoCs**

# The predictable execution model (PREM)

- **System-wide co-scheduling of memory phases from multiple actors**

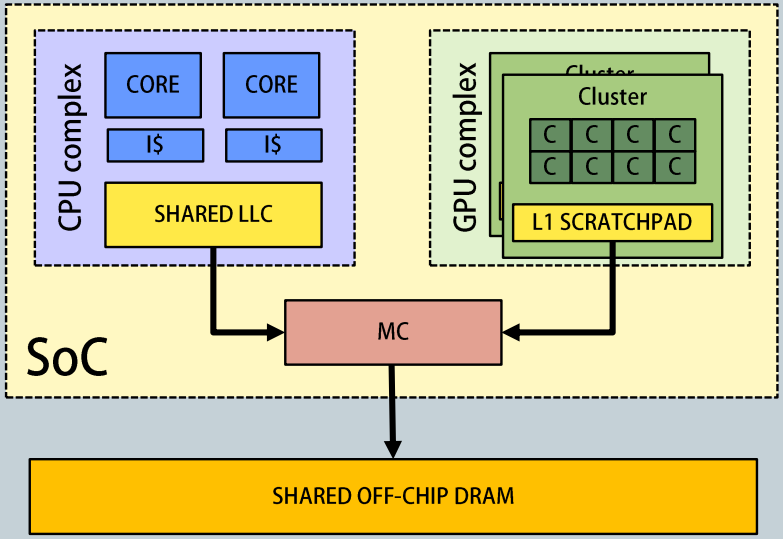
**Requires runtime techniques for global memory arbitration**

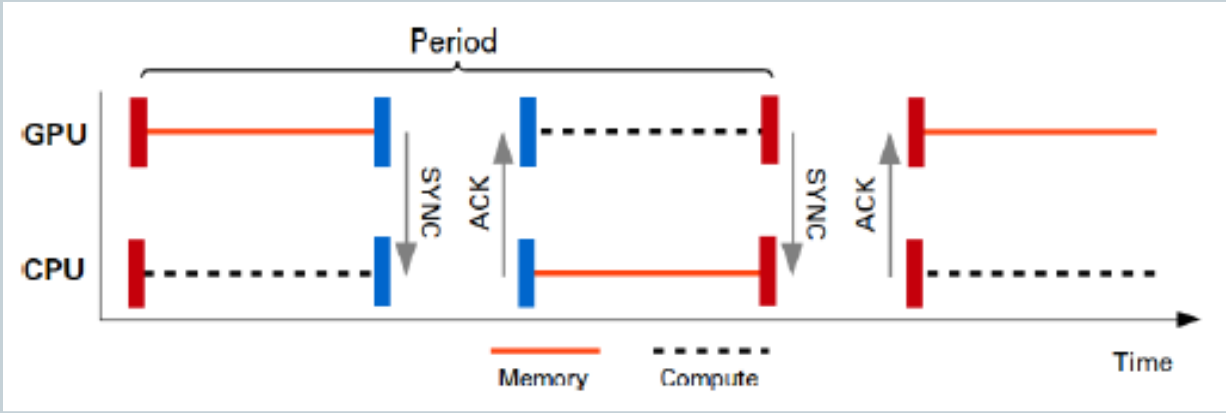



**Originally proposed for (multi-core) CPU.**
**We study the applicability of this idea to heterogeneous SoCs**

# A heterogeneous variant of PREM

- **Current focus on GPU behavior**
  ***(way more severely affected by interference than CPU)***
  - SPM as a predictable, local memory
  - Implement PREM phases within a single offload
    - **Arbitration of main memory accesses via timed interrupts+shmem**
  - Rely on high-level constructs for offloading

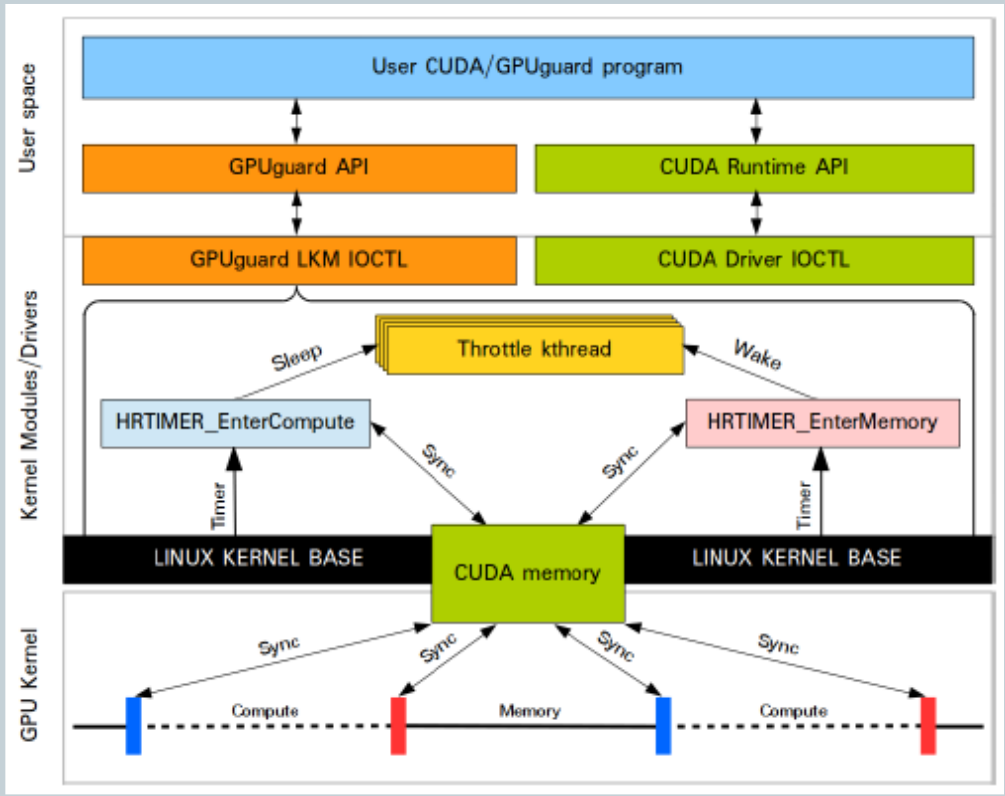


# CPU/GPU synchronization

**Basic mechanism in place to control GPU execution**

- **CPU inactivity forced via the *throttle thread* approach (MEMguard)**
- **GPU inactivity forced via active polling (GPUguard)**



# OpenMP offloading annotations

```
#pragma omp target teams distribute parallel for
for (i = LB; i < UB; i++)
      C[i] = A[i] + B[i]
```

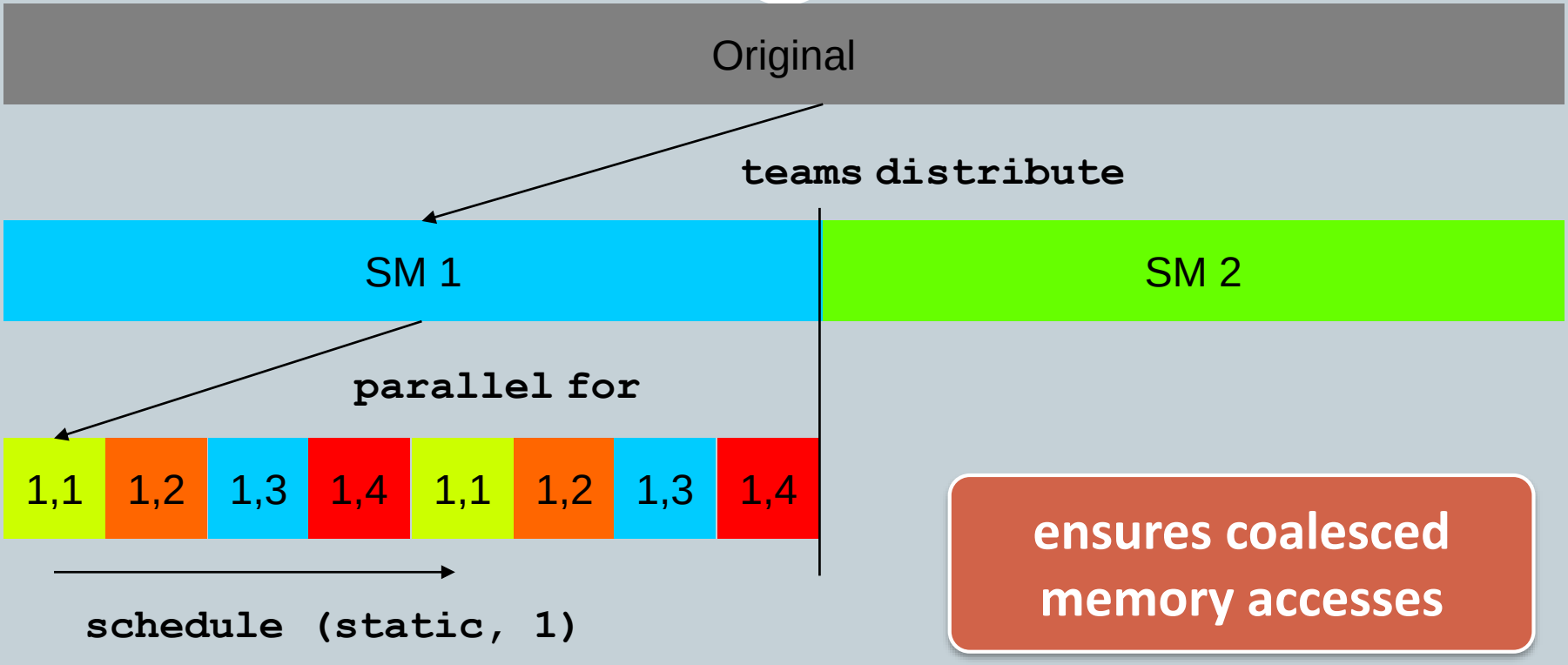
Original loop iteration space as given in the sequential code

The annotated loop is to be offloaded to the accelerator

Divide the iteration space over the available execution groups (SMs in CUDA)

Execute loop iterations in parallel over the available threads

# OpenMP offloading annotations




**ensures coalesced memory accesses**

The OpenMP **schedule** *decides which elements* each GPU thread accesses.

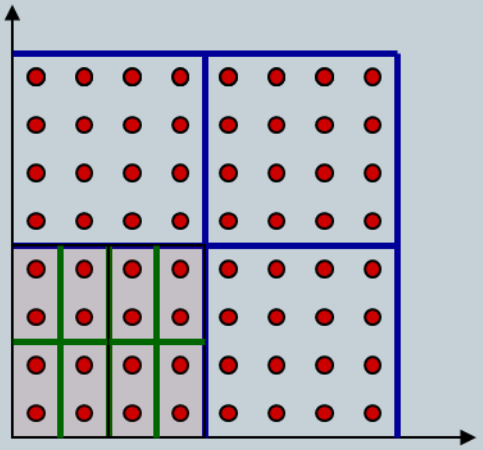
# Mechanisms to analyze/transform the code

- Determine which accesses in offloaded kernel are to be satisfied through SPM
  - programming model abstractions to specify shared data with the host
- Identify allocation regions. Data is brought in upon region entry and out upon region exit
  - We statically know that data is in the SPM for the whole duration of the region
  - The object is available in the SPM independently of the (control flow) path that is taken within the region
- Convenient abstraction to reason on the footprint of data
  - must fit in SPM and use it as much as possible

# Loops

- Apply *tiling* to reorganize loops so as to operate in stripes of the original data structures whose footprint can be accommodated in the SPM
  - Can leverage well-established techniques to prepare loops to expose the most suitable access pattern for SPM reorganization

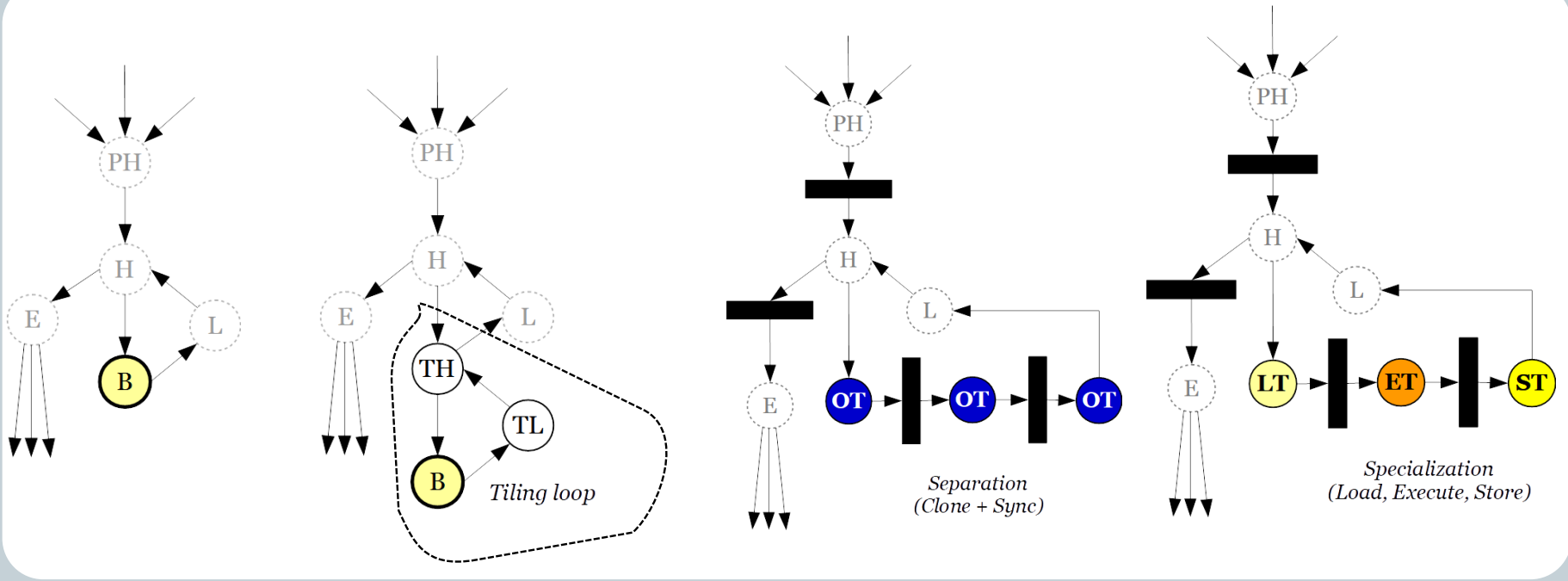
```
FORALL i = 1, Ni
 FORALL j = 1, Nj
  FOR k = 1, WS
   FOR l = 1, WS
    S1
   END FOR
  END FOR
 END FORALL
END FORALL
```



**So far the main transformation applied**

# Hercules Clang OpenMP Compilation

**Loop tiling** → **Loop outlining** → **Separation** → **Specialization**



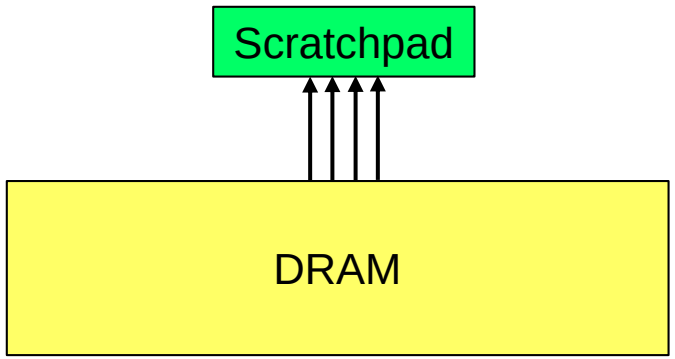
# Hercules Clang OpenMP Compilation

```
void outlined_fun(…) {
  for(…) {
    C[j * TS + i] =
      A[j * TS + i] +
      B[j * TS + i]
  }
}
```

**Specialization**

```
outlined_fun_memory_in(…) {
  for(…) {
    spmA[i] = A[j * TS + I]
    spmB[i] = B[j * TS + i]
  }
}
```

Specialize function for **Scratchpad Load**




- Allocate scratchpad buffers
- Redirect loads into scratchpad

# Hercules Clang OpenMP Compilation

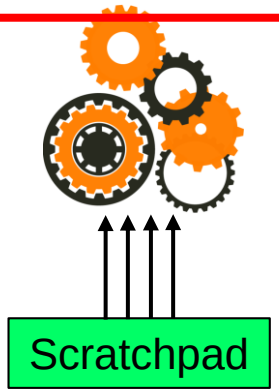
```
void outlined_fun(…) {
  for(…) {
    C[j * TS + i] =
      A[j * TS + i] +
      B[j * TS + i]
  }
}
```

**Specialization**

```
outlined_fun_memory_in(…) {
  for(…) {
    spmA[i] = A[j * TS + I]
    spmB[i] = B[j * TS + i]
  }
}
```

Specialize function for
**In-Scratchpad Compute**

```
outlined_fun_compute(…) {
  for(…) {
    spmC[i] = spmA[i] + spmC[i]
  }
}
```




- Use scratchpad buffer for computations

# Hercules Clang OpenMP Compilation

```
void outlined_fun(…) {
  for(…) {
    C[j * TS + i] =
      A[j * TS + i] +
      B[j * TS + i]
  }
}
```

**Specialization**

```
outlined_fun_memory_in(…) {
  for(…) {
    spmA[i] = A[j * TS + I]
    spmB[i] = B[j * TS + i]
  }
}
```

Specialize function for **Scratchpad Writeback**

```
outlined_fun_compute(…) {
  for(…) {
    spmC[i] = spmA[i] + spmC[i]
  }
}
```

Scratchpad

```
outlined_fun_memory_out(…) {
  for(…) {
    C[j * TS + i] = spmC[i]
  }
}
```

DRAM

- Allocate buffers for output
- Write out data from scratchpad

# Hercules Clang OpenMP Compilation

**Memory phase**

**Compute phase**

```
outlined_fun_memory_in(…) {
  for(…) {
    spmA[i] = A[j * TS + I]
    spmB[i] = B[j * TS + i]
  }
}
```

**SYNC**

**SYNC**

```
outlined_fun_compute(…) {
  for(…) {
    spmC[i] = spmA[i] + spmC[i]
  }
}
```

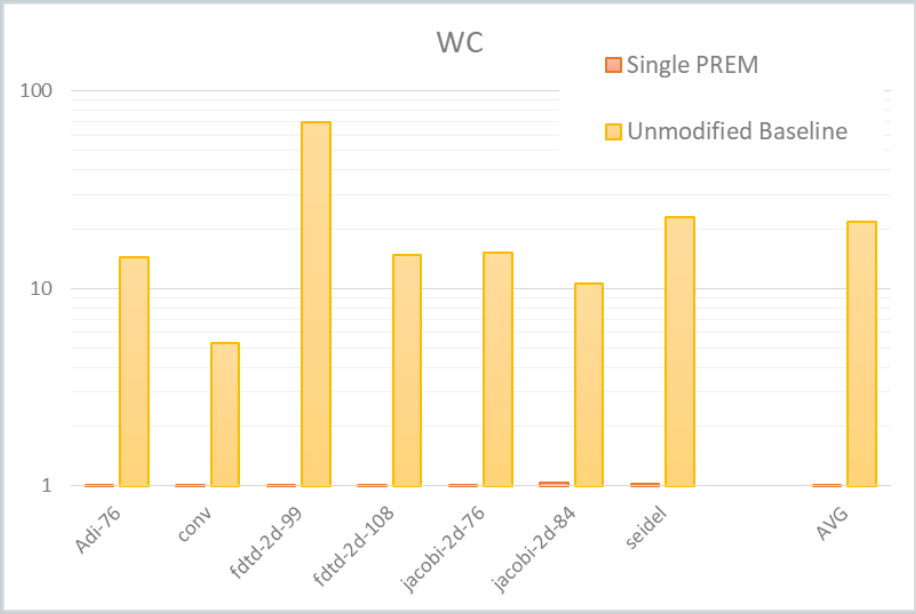
```
outlined_fun_memory_out(…) {
  for(…) {
    C[j * TS + i] = spmC[i]
  }
}
```

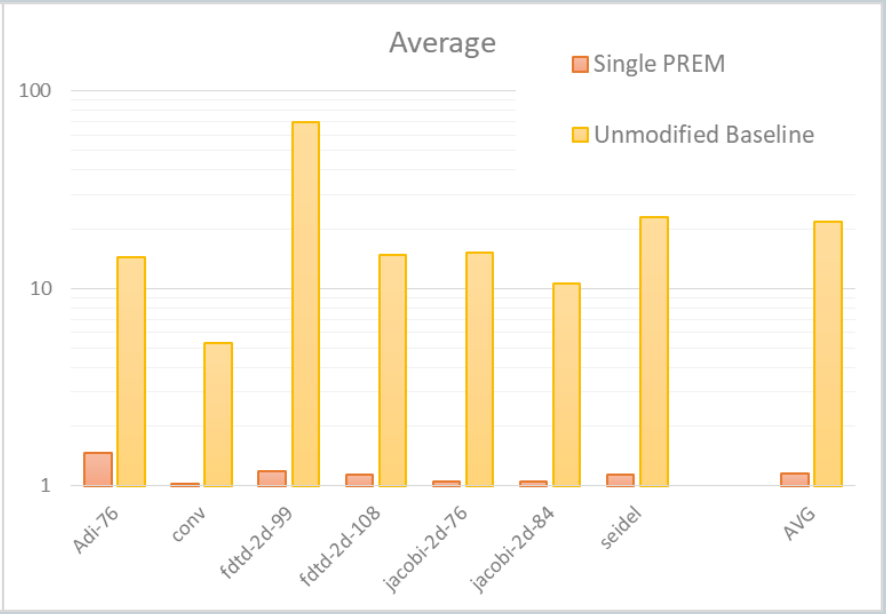
**SYNC**

# EVALUATION

## PREDICTABILITY

1. Near-zero variance when sizing PREM periods for the worst case
2. What's the performance drop?



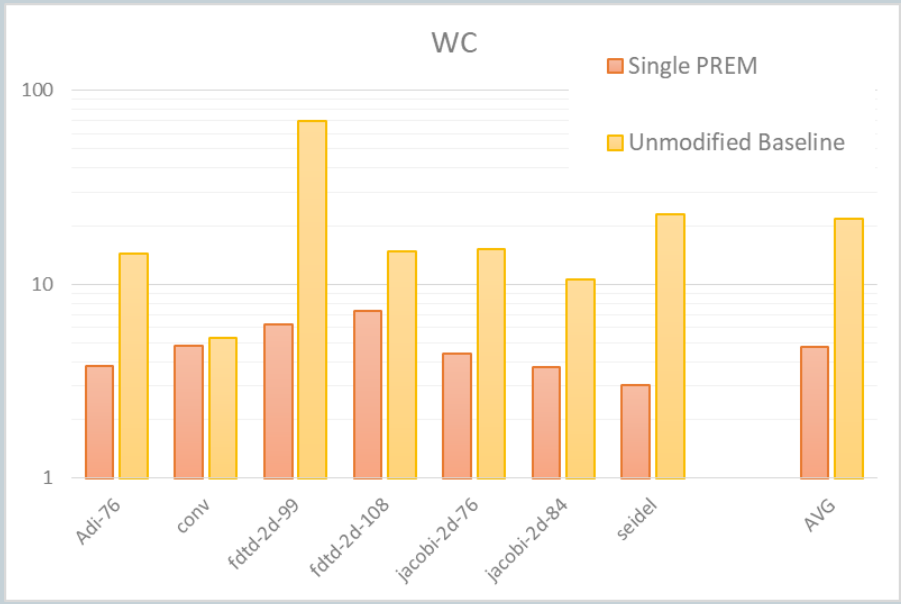



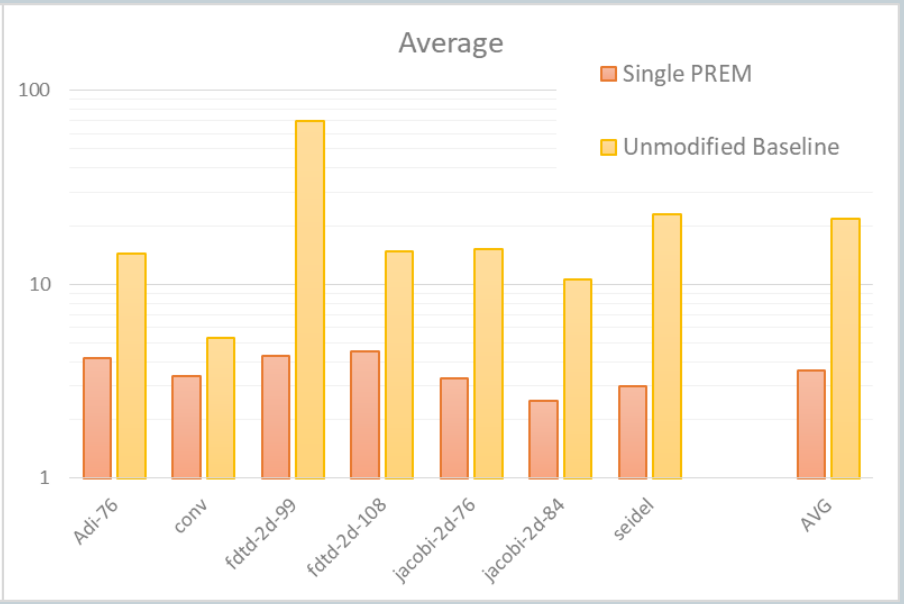
# EVALUATION

## PREDICTABILITY vs PERFORMANCE

1. Up to 11x improvement wrt unmodified program w interference

2. Up to 6x slowdown wrt unmodified program w/o interference

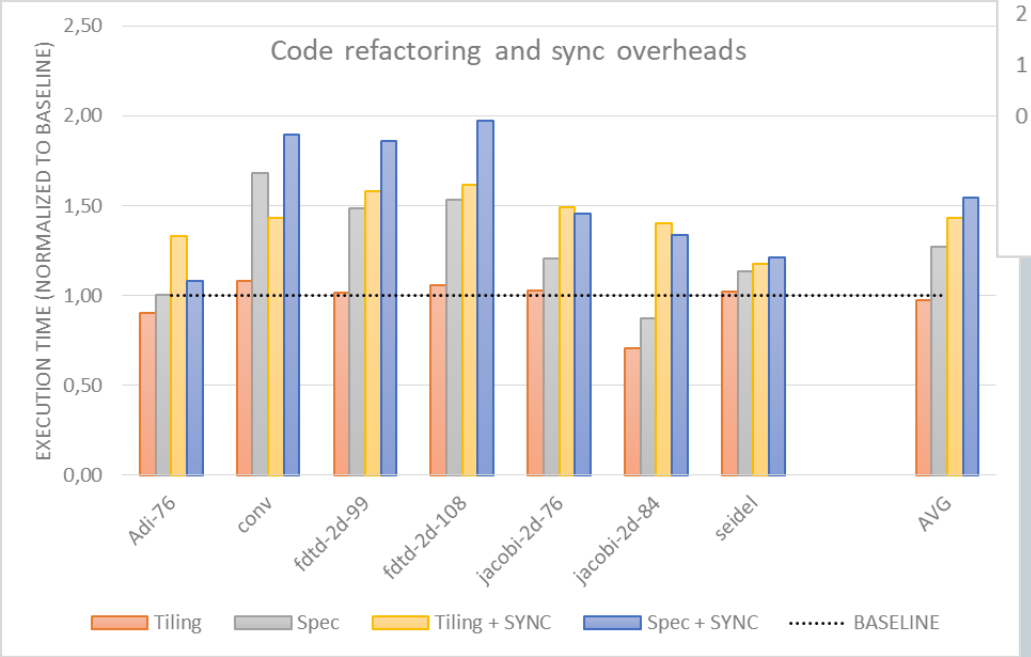


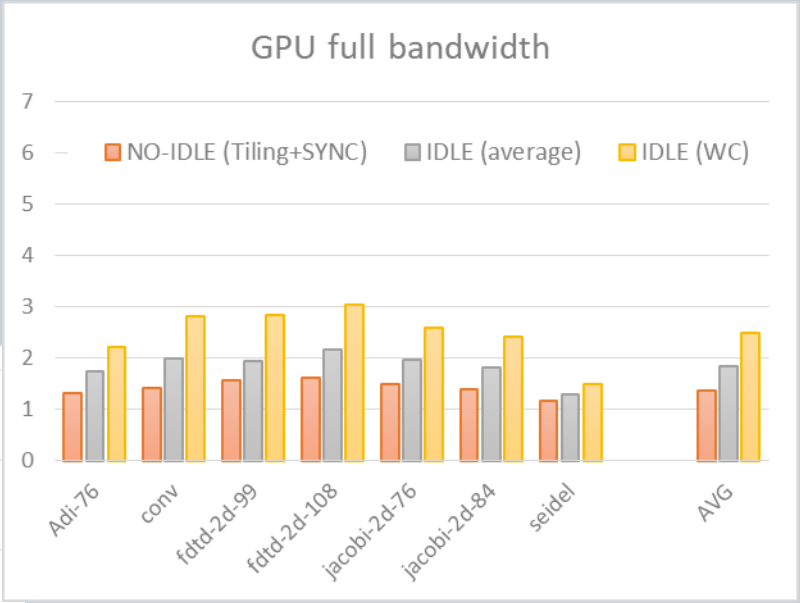



# EVALUATION

## OVERHEADS (1)

1. Code refactoring
2. synchronization

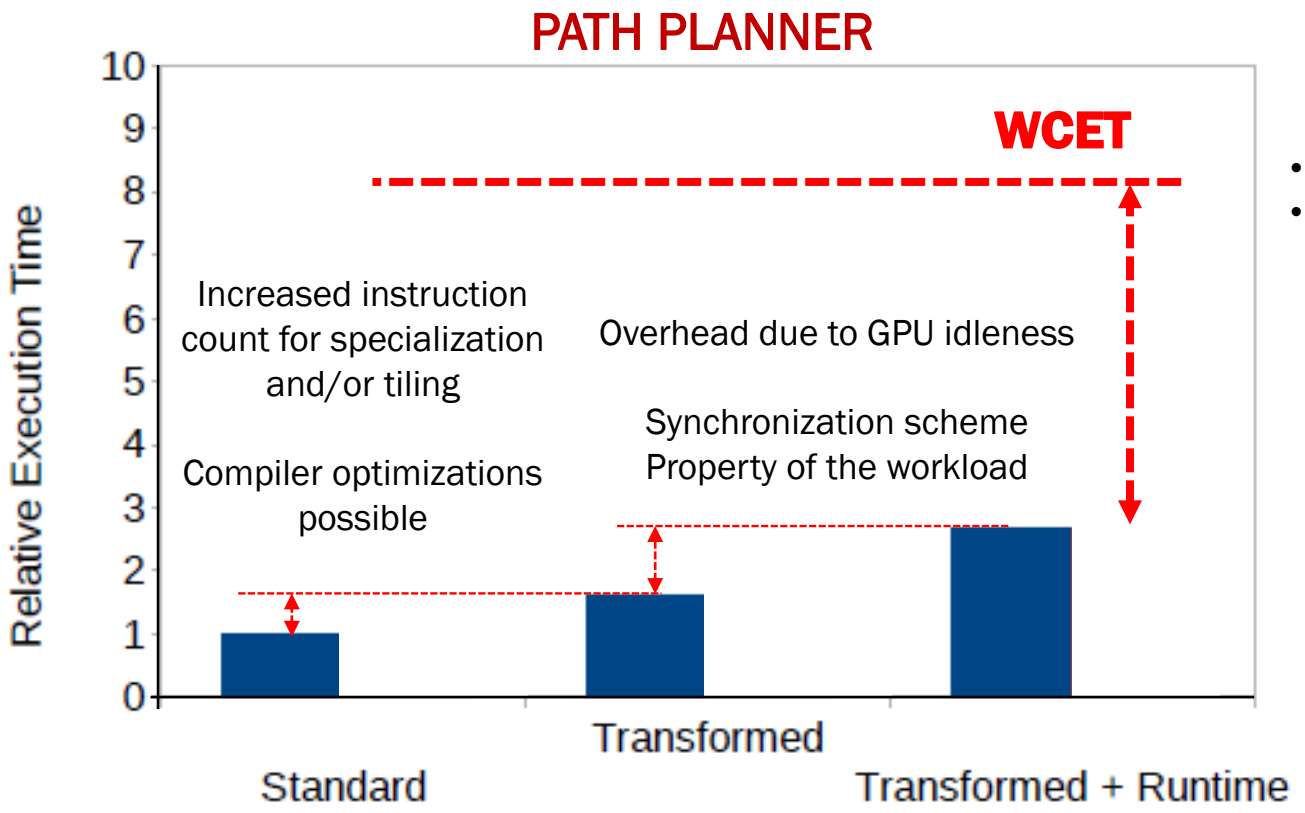



## OVERHEADS (2)

1. GPU idleness

# EVALUATION




- Near zero variance
- 3X reduction in WCET

Isolated run
Memory Contended

Björn Forsberg, Daniele Palossi, Andrea Marongiu, Luca Benini:
*GPU-Accelerated Real-Time Path Planning and the Predictable Execution Model*. **ICCS 2017**

# WORK IN PROGRESS…

- Understanding key functionality/performance issues
- Extensively characterizing performance and overheads on real benchmarks
- PREMization of CPU code
  - Extend scope of analysis to more control-oriented codes

# END!

**THANKS**!

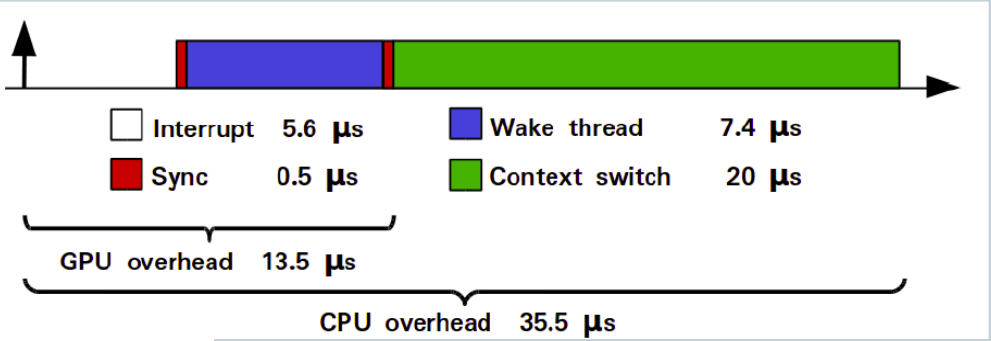
ANDREA MARONGIU
Università Di Bologna
a.marongiu@unibo.it

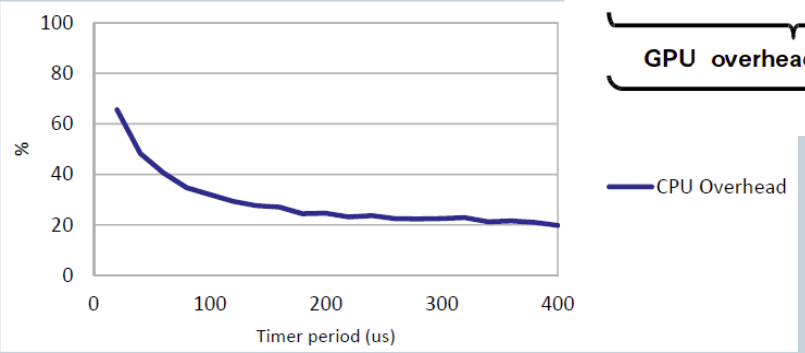
# Known limitations

- No recursion or function pointers. Local objects must have fixed or bounded sizes.
  - GPU codes tend to be much more regular in nature
  - Such assumptions conform with standard conventions for real-time applications (e.g. MISRA)
- All loops in the program must be bounded. Compiler analysis must be able to determine bounds (of loops and arrays)
  - annotations or profiling can also be explored to extend the applicability of the approach
- Limited support for pointers
  - Only memory objects that can be referenced at the entry of the predictable block (no link-based data structures)
    - Such structures are commonly avoided in real-time systems due to the complexity in calculating WCET on data structures whose length is not known at compile time
    - Not common to offload such data structures for GPU operation – When it is done the code undergoes heavy restructuring using regular, dense arrays.

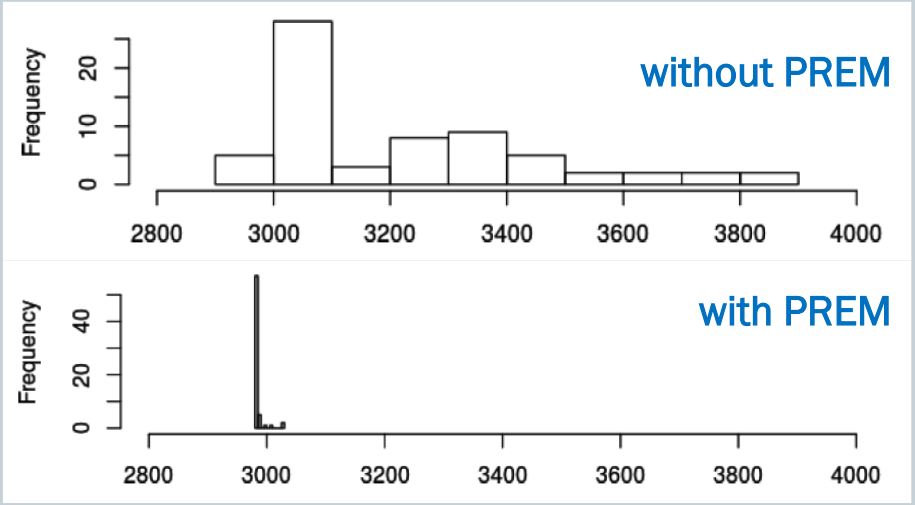
# EVALUATION

## GPUguard synchronization overhead



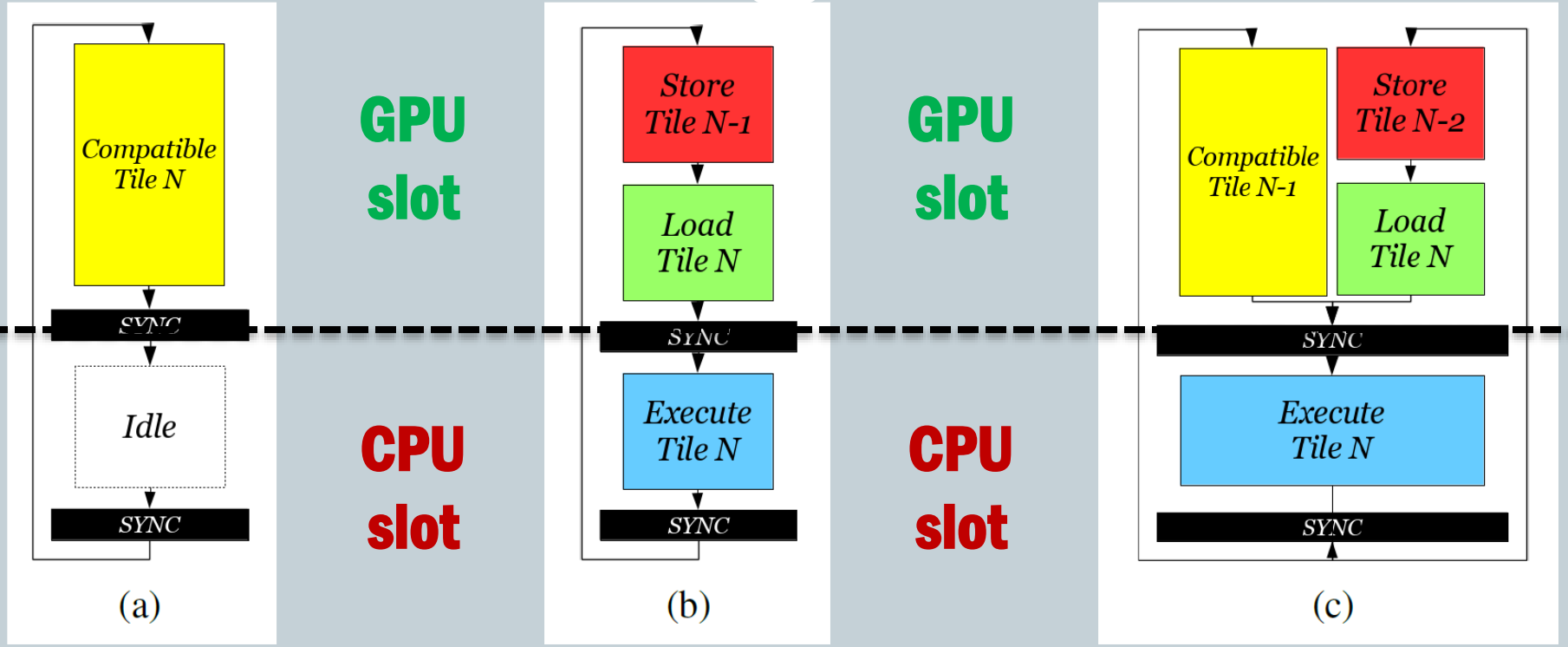

## Matrix multiplication timing variance under memory contention




- Early evaluation demonstrates effectiveness of the approach

Björn Forsberg, Andrea Marongiu, Luca Benini:
*GPUguard: Towards supporting a predictable execution model for heterogeneous SoC*. **DATE 2017**

# Deployment policies




**Compatible interval**

**PREM**

**Combined**