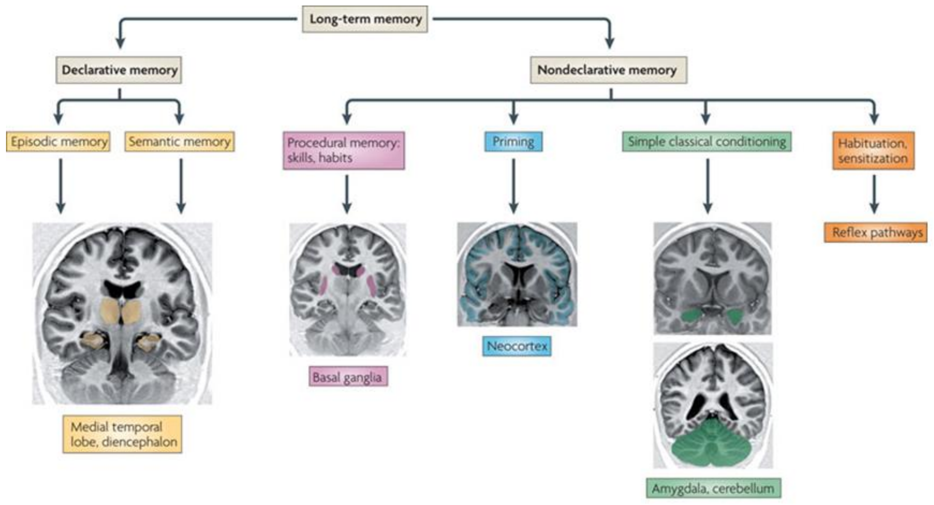
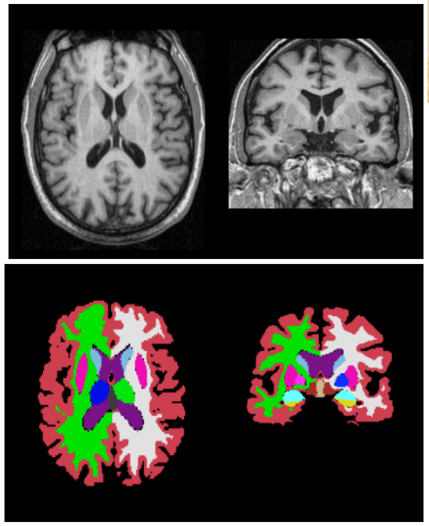
A new perspective on processing-in-memory architecture design

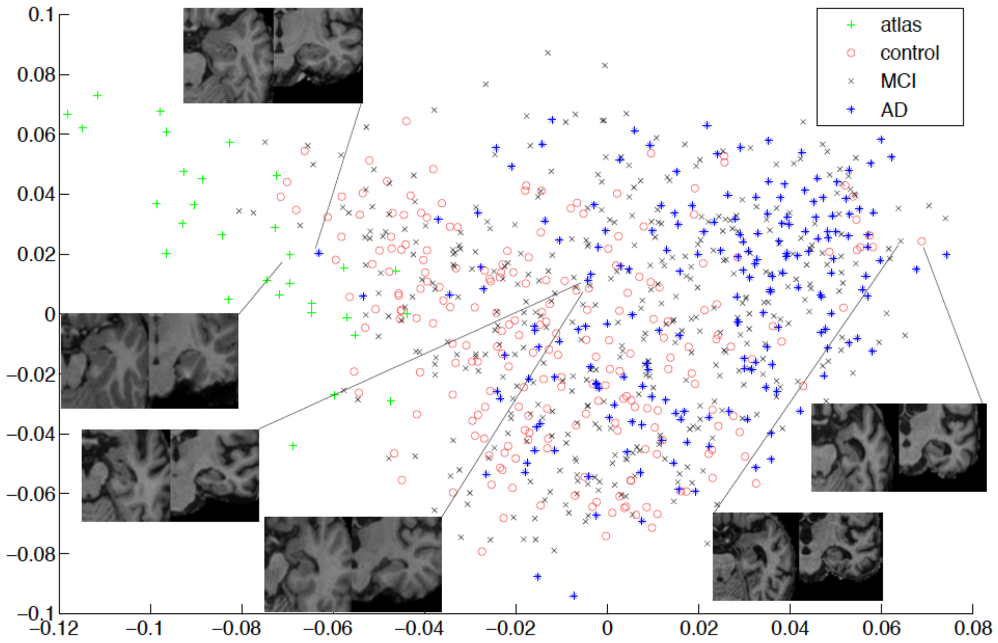
Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joe Greathouse, Mitesh Meswani, Mark Nutter, Mike Ignatowski

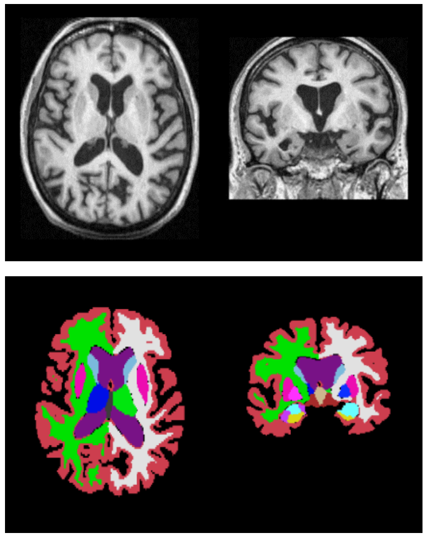
AMD Research

PROCESSING-IN-MEMORY?



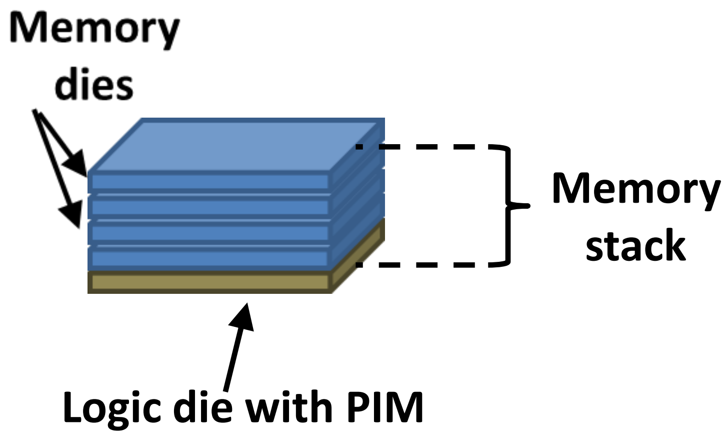






HIGHLIGHT

- Memory system is a key limiter
  - At 4TB/s, vast majority of node energy could be consumed by the memory system
- Prior PIM research constrained by
  - Implementation technology
  - Non-traditional programming models
- Our focus
  - 3D die stacking
  - Use base logic die(s) in memory stack
    - General-purpose processors
    - Support familiar programming models
    - Arbitrary programs vs a set of special operations



PROCESSING-IN-MEMORY (PIM) --- OVERVIEW (I)

- Moving compute close to memory promises significant gains
  - Memory is a key limiter (performance and power)
  - Exascale goals of 4TB/s/node and <5pj/bit
- Prior research
  - Integration of caches and computation
    - “A logic-in-memory computer” (1970)
    - No large scale integration was possible.
  - Logic in DRAM processes
    - In-memory processors with reduced performance or highly specialized for a limited set of operations.
    - Reduced DRAM due to the presence of compute logic.
  - Embedded DRAM in logic processes
    - Not cost-effectively accommodate sufficient memory capacity
    - Reduced density of embedded memory

PROCESSING-IN-MEMORY (PIM) --- OVERVIEW (II)

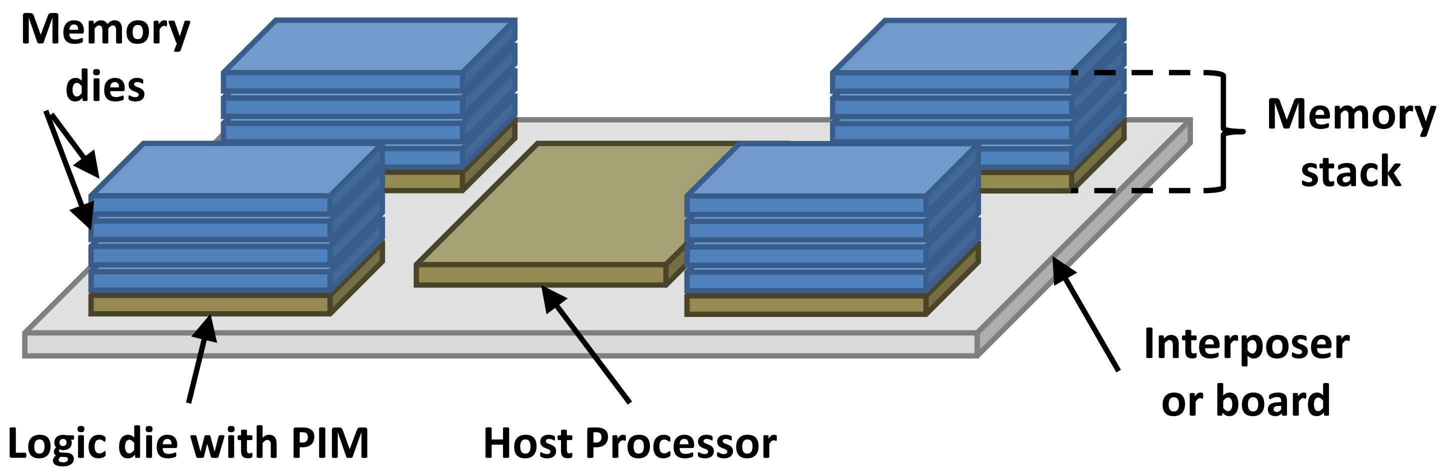
- New opportunity: logic die stacked with memory
  - Logic die needed anyway for signal redistribution and integrity
  - Potential for non-trivial compute
- Key benefits
  - Reduce bandwidth bottlenecks
  - Improve energy efficiency
  - Increase compute for a fixed interposer area
  - Processor can be optimized for high BW/compute ratio
- Challenges
  - Programming models and interfaces
  - Architectural tradeoffs
  - Application refactoring

OUTLINE

- PIM architecture baseline
- API specification
- Emulator and performance models
- Application studies

NODE HARDWARE ORGANIZATION

- Single-level of PIM-attached memory stacks
- Host has direct access to all memory
  - Non-PIM-enabled apps still work
- Unified virtual address space
  - Shared page-tables between host and PIMs
- Low-bandwidth inter-PIM interconnect



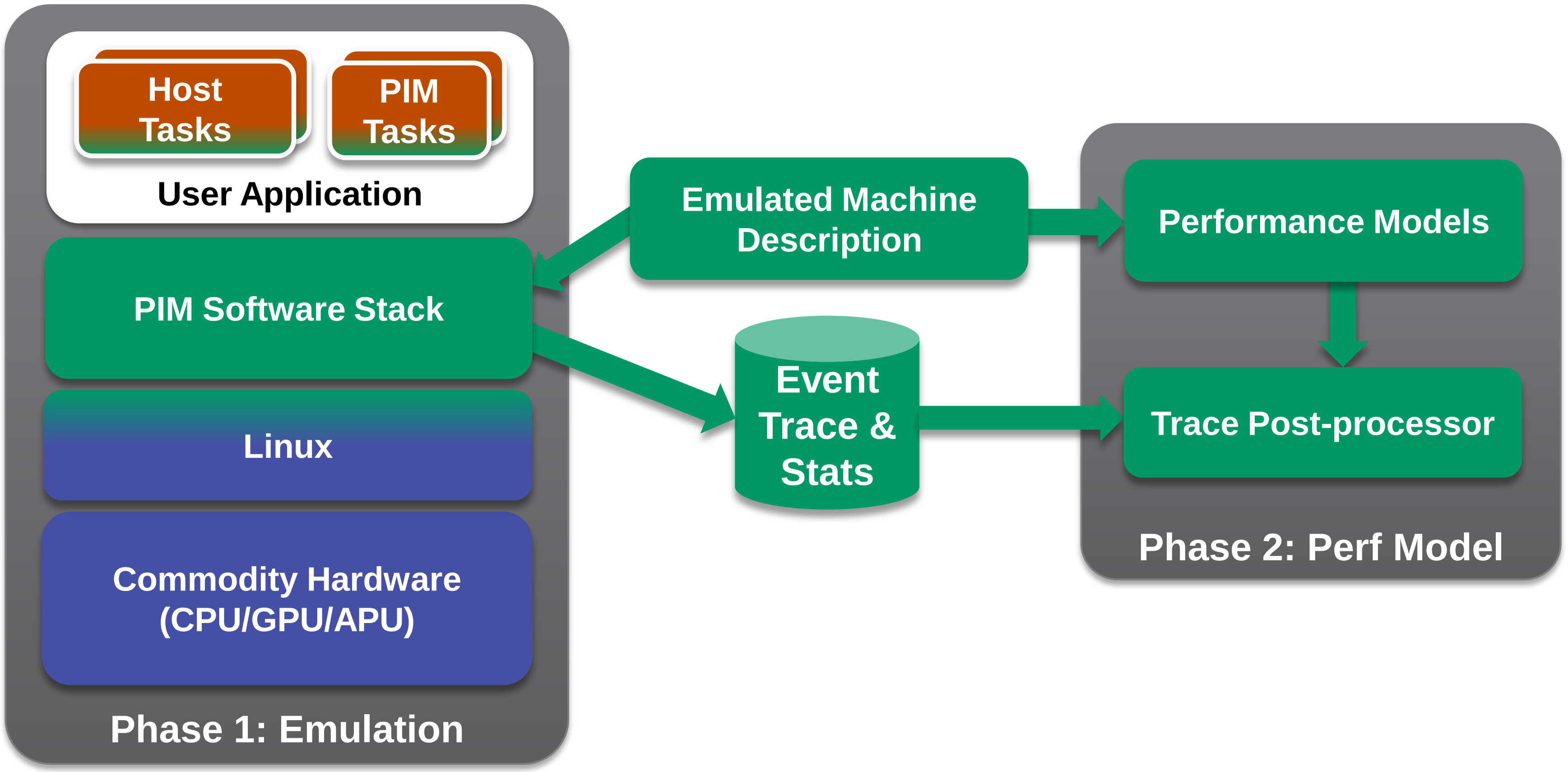
PIM API OVERVIEW

- Current focus is on single PIM-enabled node
  - Current PIM hardware baseline is general-purpose processor
- Key goals of API
  - Facilitate data layout control
  - Dispatch compute to PIMs using standard parallel abstractions
- A “convenient” level of abstraction for early PIM evaluations
  - Aimed at PIM feasibility studies
  - annotation of data and compute for PIM with reasonable programmer effort
- Key API features:
  - discover device
  - query device characteristics
  - manage locality
  - dispatch compute

PIM PTHREAD EXAMPLE: PARALLEL PREFIX SUM

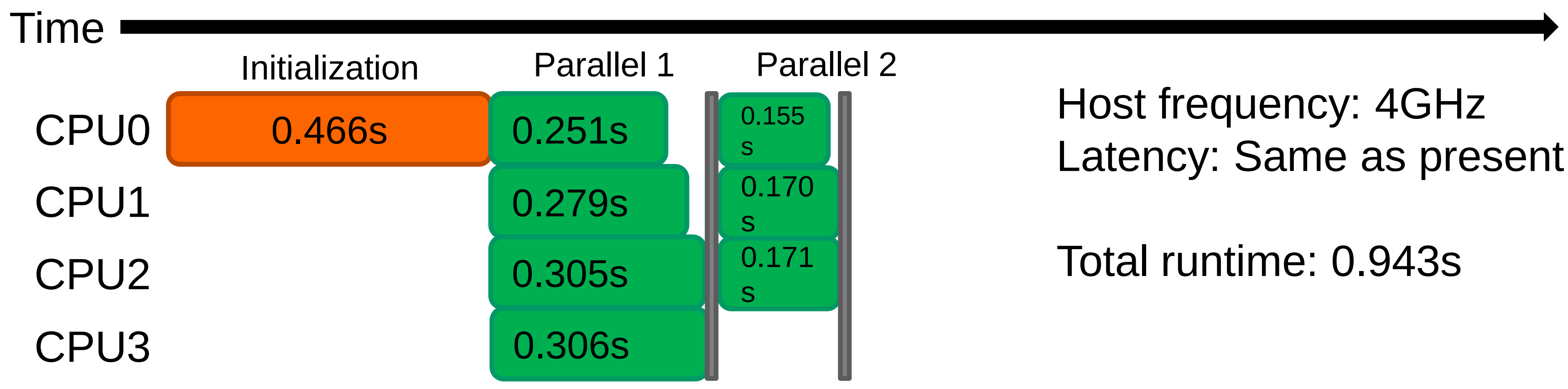
```
…
list_of_pims = malloc(max_pims * sizeof(pim_device_id));
failure = pim_get_device_ids(PIM_CLASS_0, max_pims, list_of_pims, &num_pims);
for (i = 0; i < num_pims; i++) {
    failure = pim_get_device_info(list_of_pims[i], PIM_CPU_CORES, needed_size, device_info, NULL);
    …
}
…
for (i = 0; i < num_pims; i++) {
    parallel_input_array[i] = pim_malloc(sizeof(uint32_t) * chunk_size, list_of_pims[i],
                                          PIM_MEM_DEFAULT_FLAGS, PIM_PLATFORM_PTHREAD_CPU);
    parallel_output_array[i] = pim_malloc(sizeof(uint64_t) * chunk_size, list_of_pims[i],
                                           PIM_MEM_DEFAULT_FLAGS, PIM_PLATFORM_PTHREAD_CPU);
}
…
for (i = 0; i < num_pims; i++) {
    pim_args[PTHREAD_ARG_THREAD] = &(pim_threads[i]); // pthread_t
    arg_size[PTHREAD_ARG_THREAD] = sizeof(pthread_t);
    pim_args[PTHREAD_ARG_ATTR] = NULL; // pthread_attr_t
    arg_size[PTHREAD_ARG_ATTR] = sizeof(pthread_attr_t);
    pim_args[PTHREAD_ARG_INPUT] = &(thread_input[i]); // void * for thread input
    arg_size[PTHREAD_ARG_INPUT] = sizeof(void *);
    pim_function.func_ptr = parallel_prefix_sum;
    spawn_error = pim_spawn(pim_function, pim_args, arg_size, NUM_PTHREAD_ARGUMENTS,
                            list_of_pims[i], PIM_PLATFORM_PTHREAD_CPU);
}
for (i = 0; i < num_pims; i++) {
    pthread_join(pim_threads[i], NULL);
}
…
```

PIM EMULATION AND PERFORMANCE MODEL



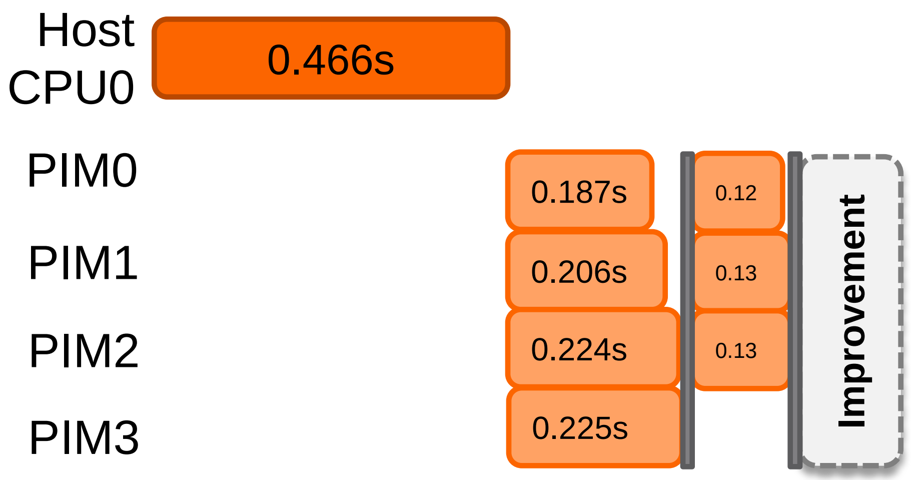
- Phase 1: Native execution on commodity hardware
  - Capture execution trace and performance stats
- Phase 2: Post-process with performance models
  - Predict overall performance on future memory and processors

PARALLEL PREFIX SUM – HOST-ONLY VS HOST+PIM



Host frequency: 4GHz
Latency: Same as present

Total runtime: 0.943s

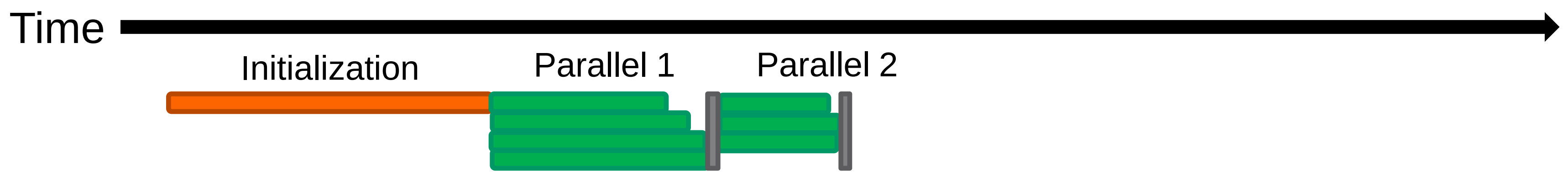


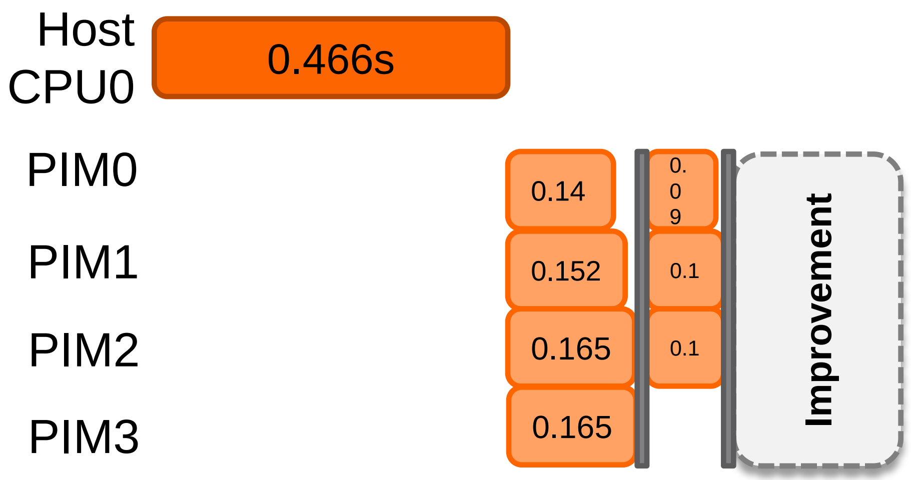
Host frequency: 4GHz
PIM frequency: 2GHZ
Host latency: current
PIM latency: 30% reduction

Total runtime: 0.820s
15% Faster

PIM computation benefits from reduced latency

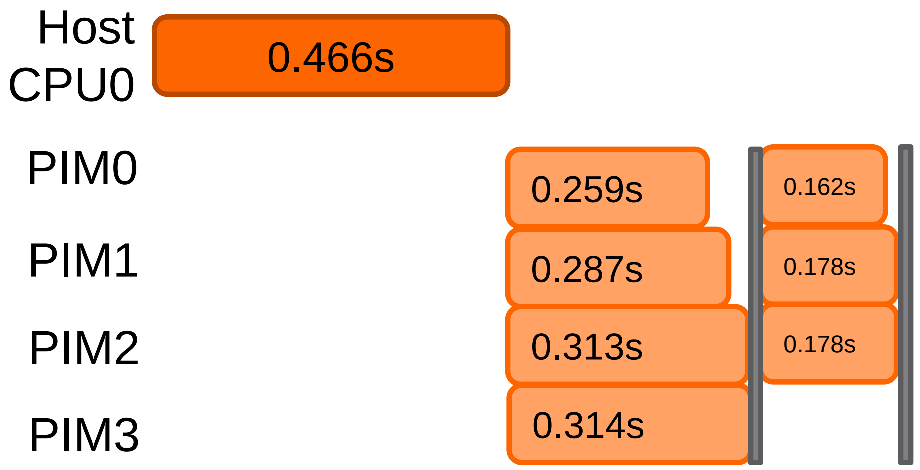
PARALLEL PREFIX SUM – HOST-ONLY VS HOST+PIM





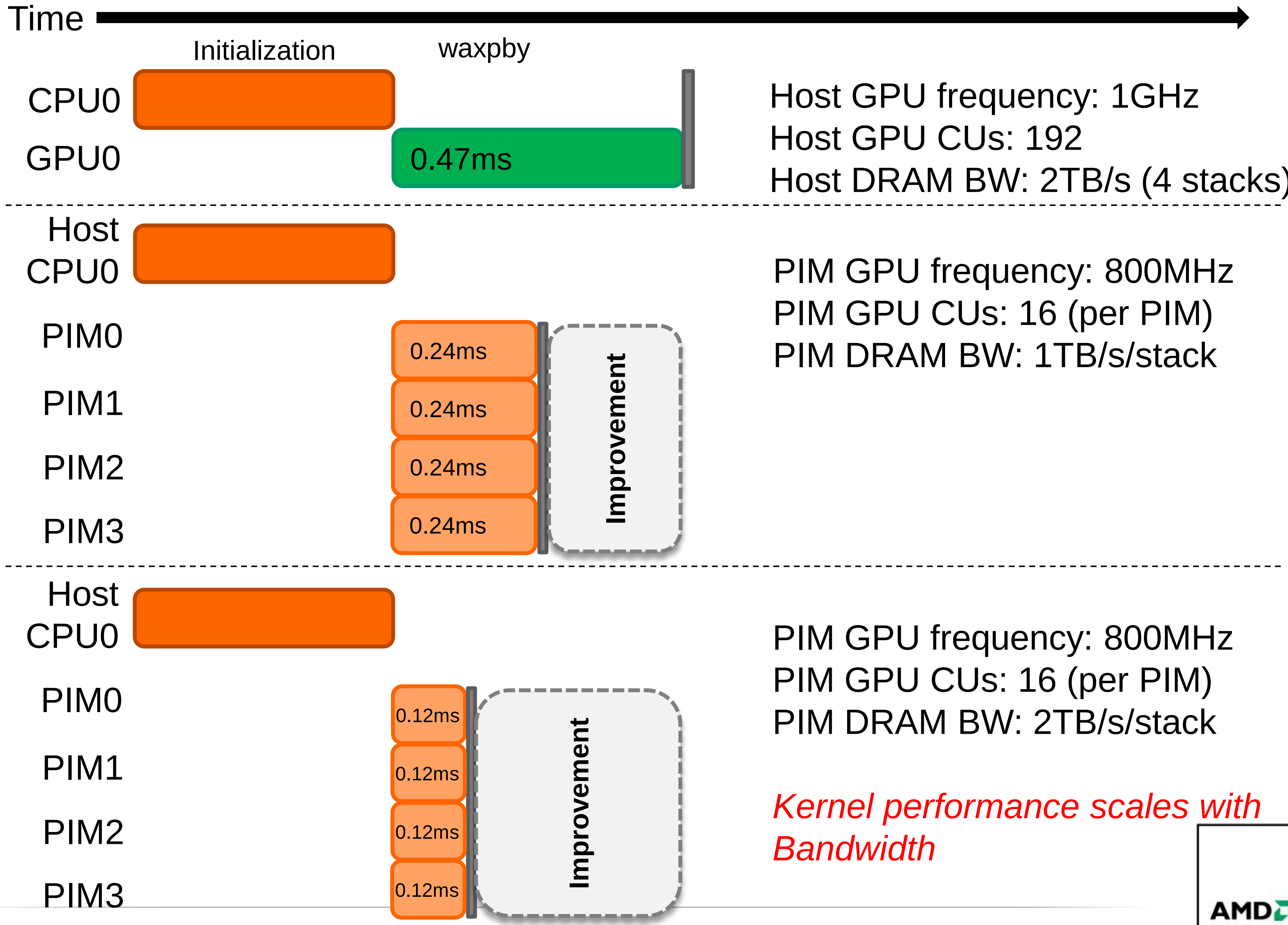
Host frequency: 4GHz
PIM frequency: 2GHZ
Host latency: current
PIM latency: 50% reduction

Total runtime: 0.728s
30% Faster



Host frequency: 4GHz
PIM frequency: 2GHZ
Host latency = PIM latency
Total runtime: 0.958s
2% Slower

WAXPBY – HOST-ONLY VS HOST+PIM



Host GPU frequency: 1GHz
Host GPU CUs: 192
Host DRAM BW: 2TB/s (4 stacks)

PIM GPU frequency: 800MHz
PIM GPU CUs: 16 (per PIM)
PIM DRAM BW: 1TB/s/stack

PIM GPU frequency: 800MHz
PIM GPU CUs: 16 (per PIM)
PIM DRAM BW: 2TB/s/stack

Kernel performance scales with Bandwidth

SUMMARY AND FUTURE WORK

- PIM architecture baseline
- API specification
- Emulator and performance models
- Application studies

- Design space study
- Evaluation of the performance models
- Further work on API, execution model, applications etc.

ACKNOWLEDGEMENT:

Lee Howes

Gabe Loh

QUESTIONS?

FURTHER DISCUSSIONS:

Dongping.zhang@amd.com