Fast Computational GPU Design with GT-Pin

Melanie Kambadur[*], Sunpyo Hong[+], Juan Cabral[+], Harish Patil[+], Chi-Keung Luk[+], Sohaib Sajid[+], Martha A. Kim[*].

[*] Columbia University, New York, NY.

[+] Intel Corporation, Hudson, MA.

GPU Applications

Graphics

- Output image to screen

Computational

- computer vision, finance, data mining

GPU Applications

Graphics

- Output image to screen

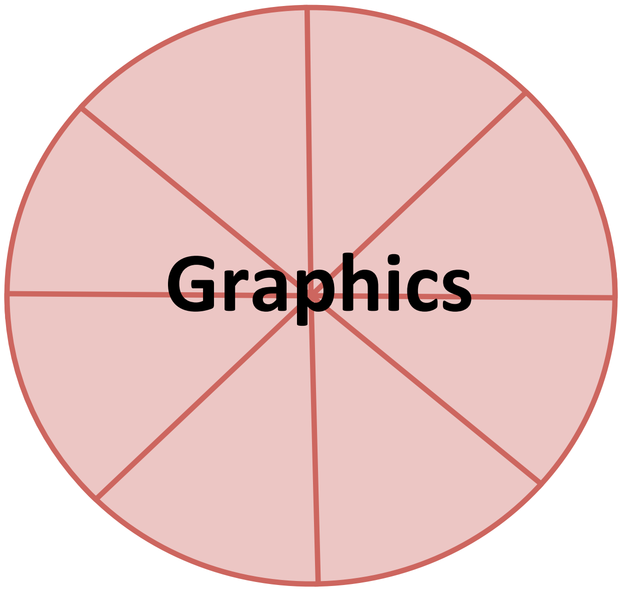
Computational

- computer vision, finance, data mining

Same:

- Highly Parallel
- Potentially low energy

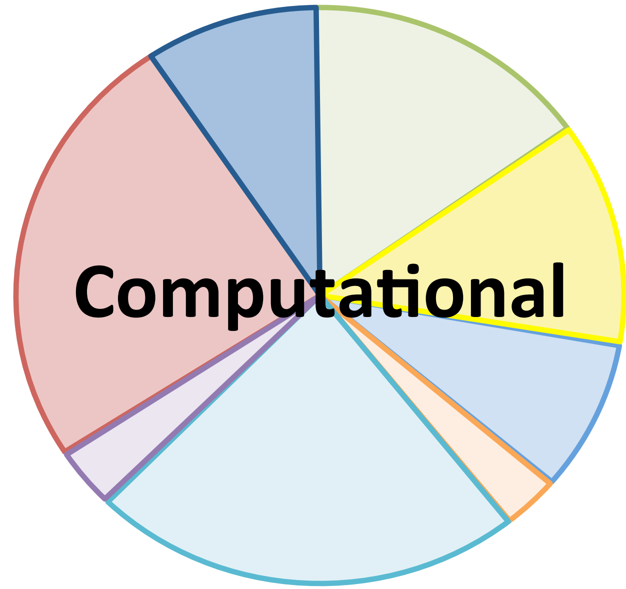
GPU Applications



- Repetitive computations

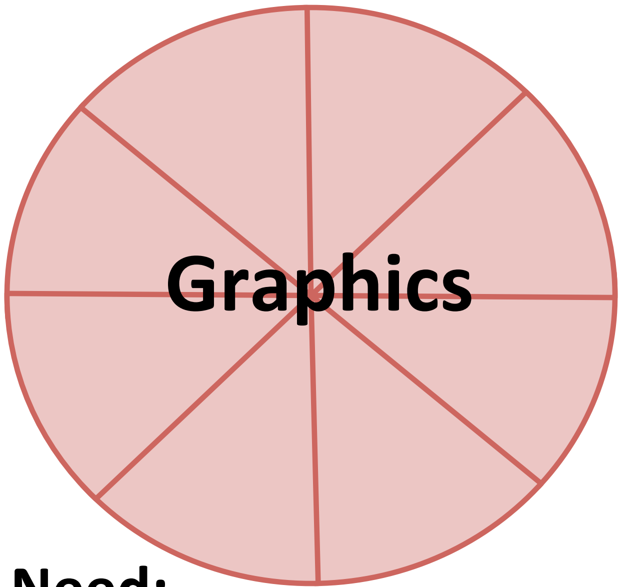
Different:

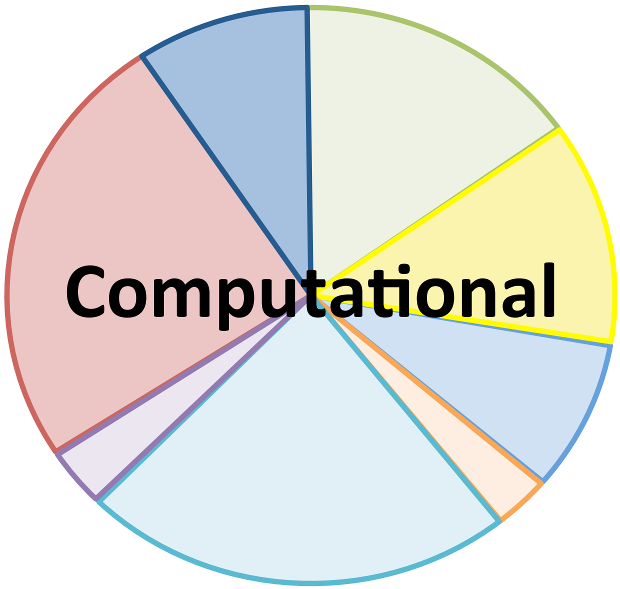
- instruction mix, parallel paradigm, performance (IPC)



- Diverse computations

GPU Applications



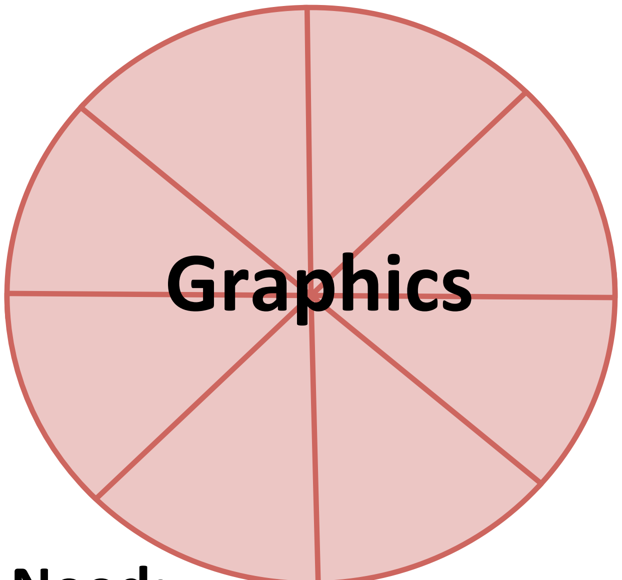


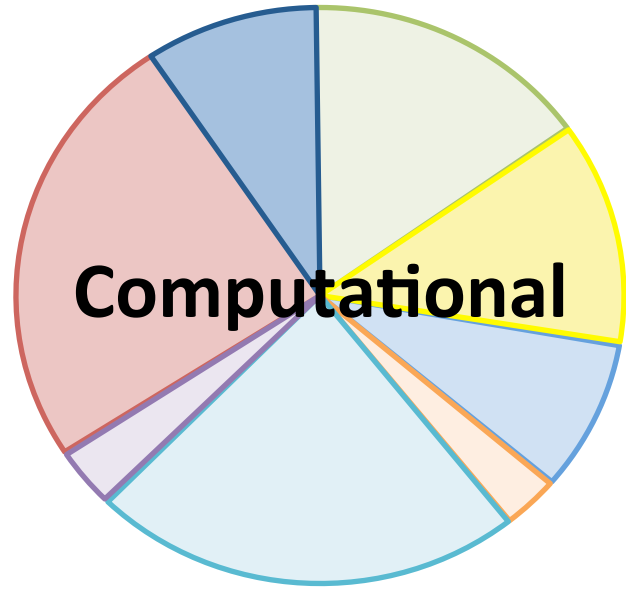
Need:

1. New ways to evaluate computational GPU apps

2. New hardware designs

GPU Applications





Need:

1. New ways to evaluate computational GPU apps
   - ➔ Contribution 1: GT-Pin Tool
   - ➔ Contribution 2: Evaluation of very large computational apps
2. New hardware designs
   - ➔ Contribution 3: Accelerated µarch simulation for GPUs

Talk Overview

- ~~Motivation~~
- GT-Pin Tool
- Sample Measurements
- GPU Simulation Acceleration

GT–Pin Profiling Tool

- Pin for GPUs, i.e. dynamic binary instrumentation for OpenCL programs on Intel GPUs.
- 100K to 1M times faster than simulation
- Provides detailed low-level info:
  - opcode mixes
  - instruction counts
  - basic block counts
  - memory access counts

  … and more
- Custom GT-pin tools

How GT-Pin works (first OpenCL background)

- OpenCL is a language standard for heterogeneous computing (e.g. CPU+GPU)

How GT-Pin works (first OpenCL background)

- OpenCL is a language standard for heterogeneous computing (e.g. CPU+GPU)
- Programs have two parts, a **host** and a **device** (e.g., what runs on CPU vs. GPU)

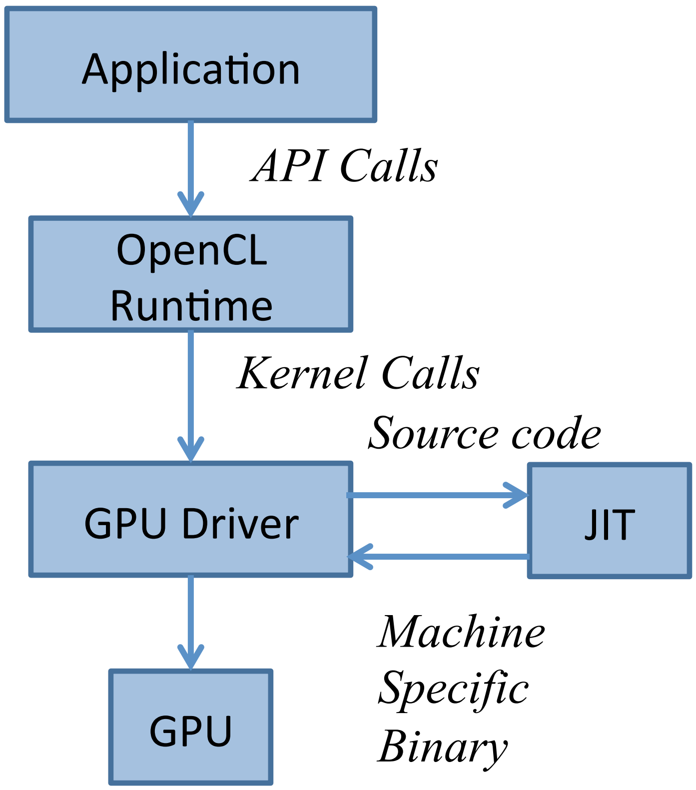
How GT-Pin works (first OpenCL background)

- OpenCL is a language standard for heterogeneous computing (e.g. CPU+GPU)
- Programs have two parts, a **host** and a **device** (e.g., what runs on CPU vs. GPU)
- Host sets up runtime env., organizes program execution (synchronization, distribution of work) through **API calls**

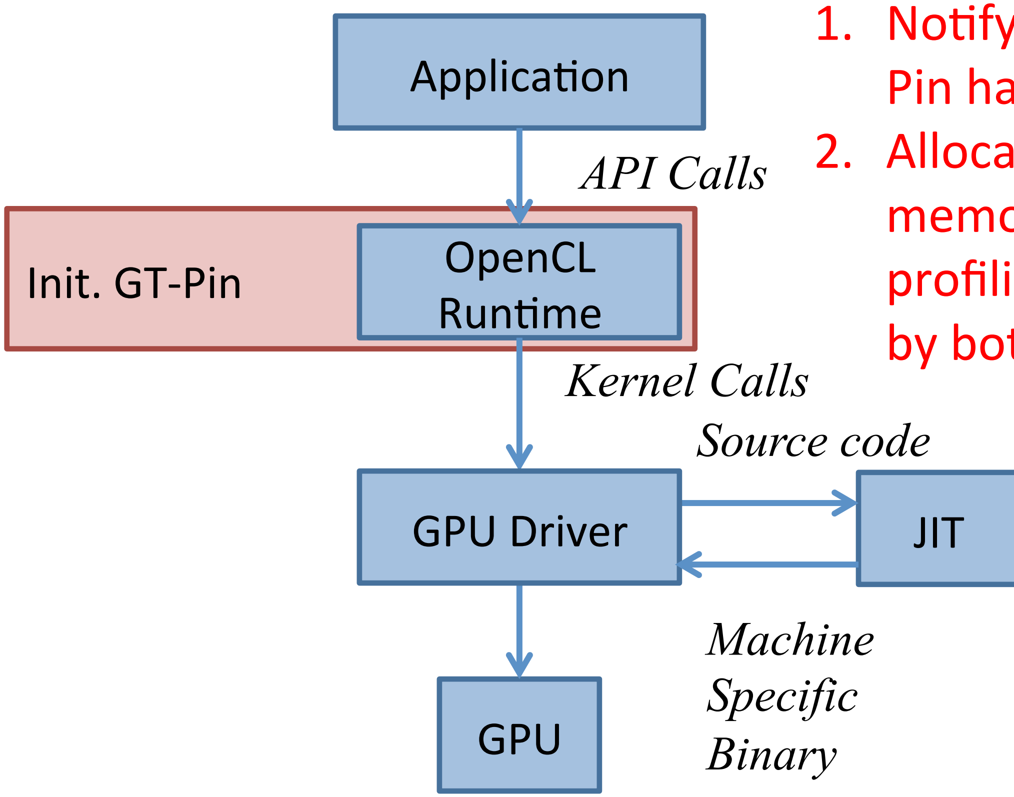
How GT-Pin works (first OpenCL background)

- OpenCL is a language standard for heterogeneous computing (e.g. CPU+GPU)
- Programs have two parts, a **host** and a **device** (e.g., what runs on CPU vs. GPU)
- Host sets up runtime env., organizes program execution (synchronization, distribution of work) through **API calls**
- Device does computational work using **kernels** (like procedures)

Normal OpenCL execution



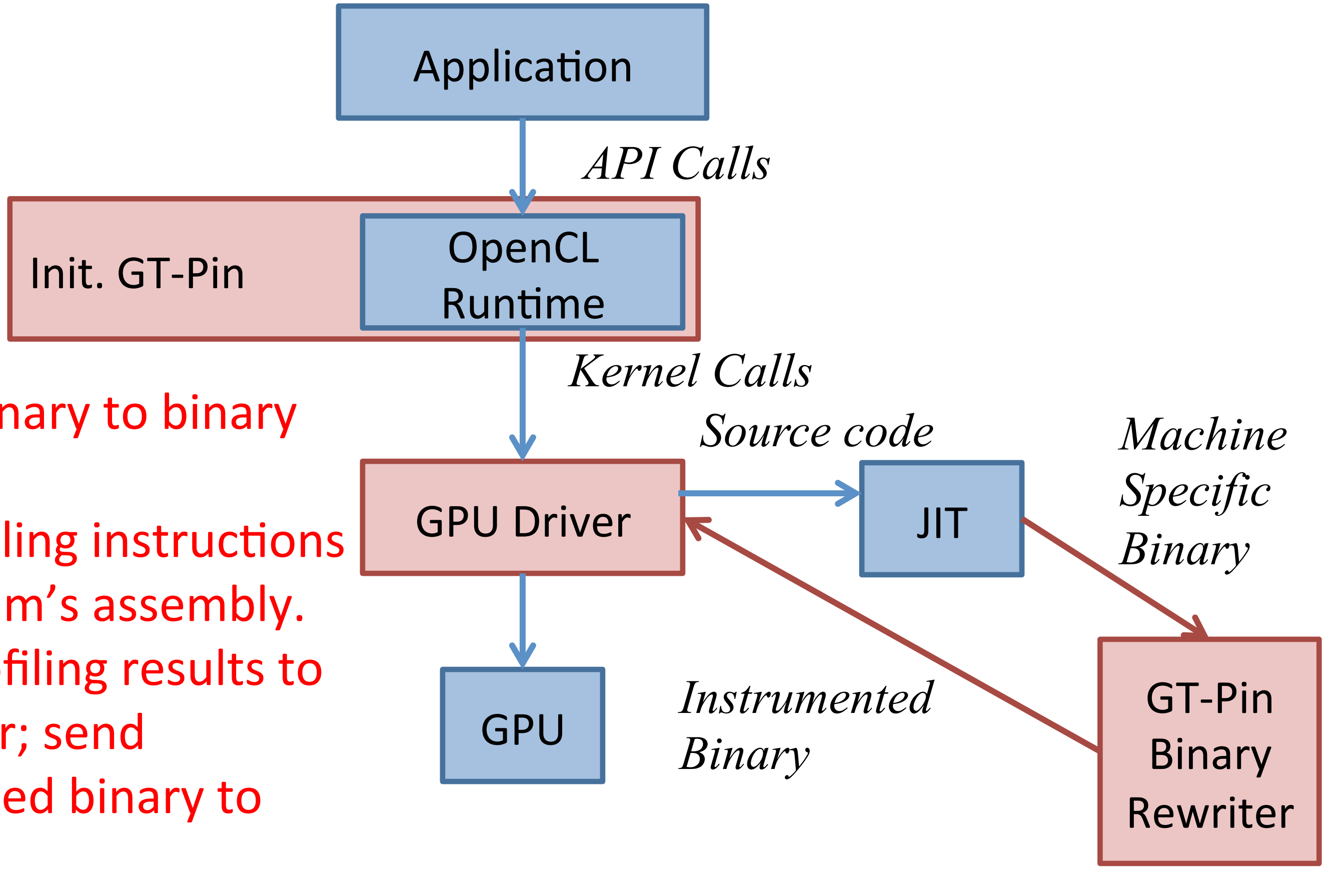
GT-Pin instrumented execution



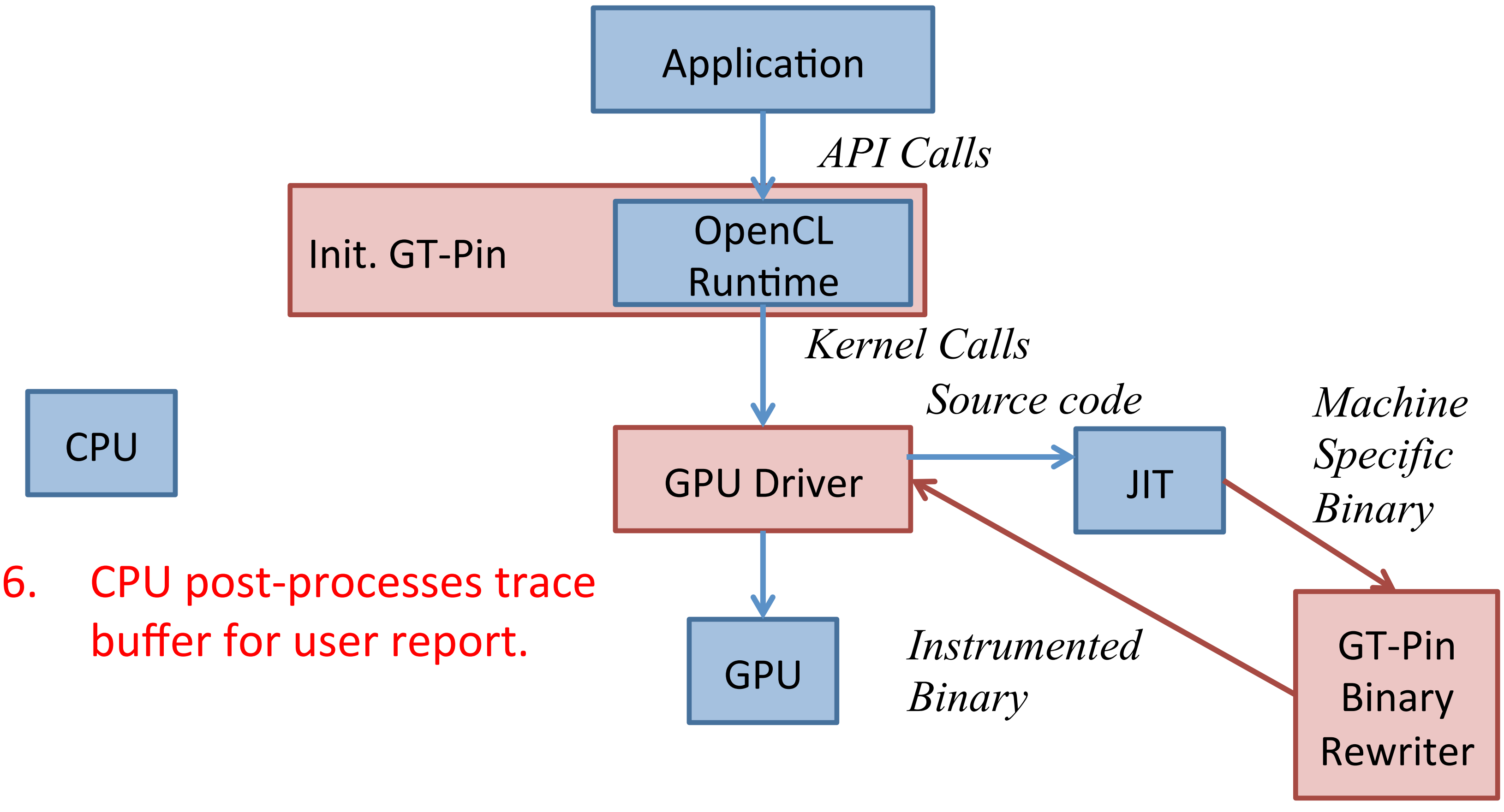
1. Notify GPU driver that GT-Pin has been invoked.
2. Allocate *trace buffer* memory space for profiling results accessible by both CPU & GPU

GT-Pin instrumented execution

3. Redirect binary to binary rewriter
4. Insert profiling instructions into program's assembly.
5. Output profiling results to trace buffer; send instrumented binary to GPU



GT-Pin instrumented execution



6. CPU post-processes trace buffer for user report.

Talk Overview

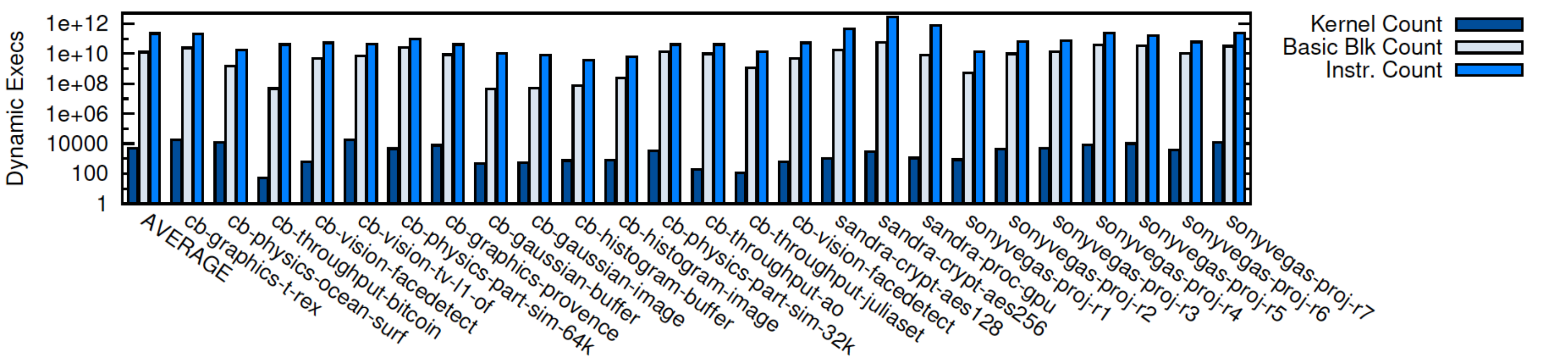
- ~~Motivation~~
- ~~GT-Pin Tool~~
- Sample Measurements
- GPU Simulation Acceleration

Workload Analysis

- 25 OpenCL benchmarks from
  - **CompuBench**
  - **SiSoftware Sandra**
  - **Sony Vegas Pro Press Project**
- Vision, finance, physics, crypto, rendering
- Test Machine: GEN 7 “Ivy Bridge” Intel Core i7-3770 CPU, Intel HD Graphics 4000 GPU, Windows 7 64-bit OS.
- Analyze a variety of metrics in Section IV

Workload Analysis

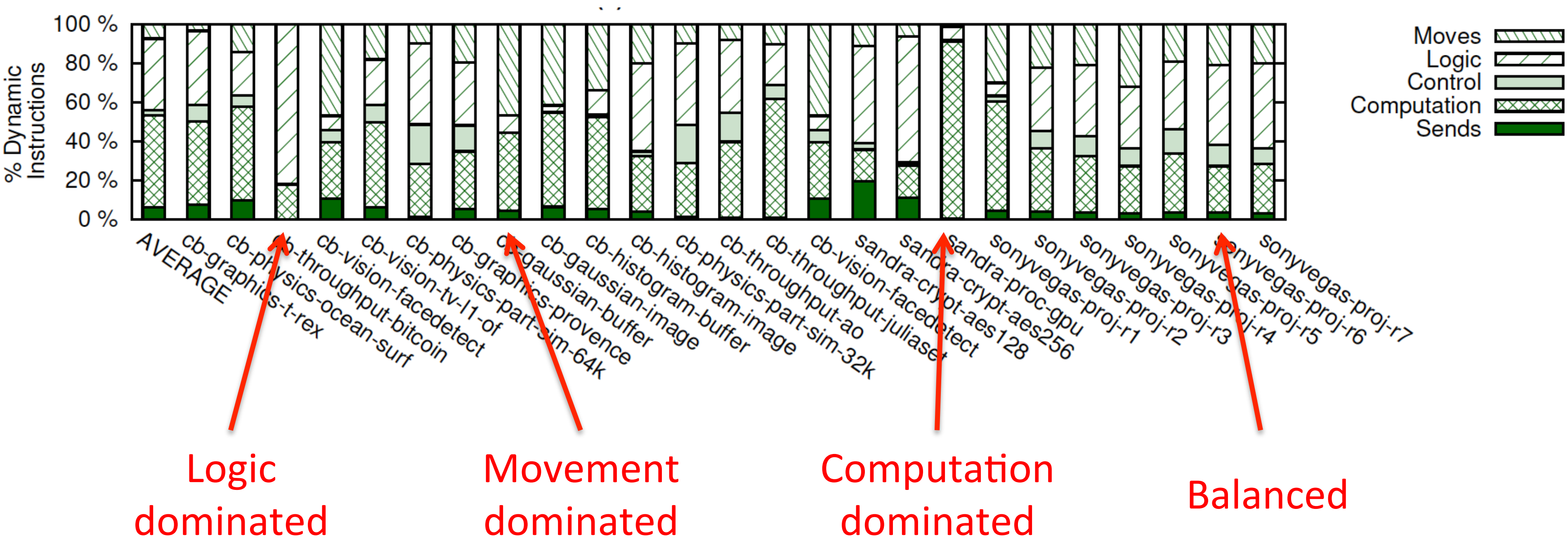
- Large real world applications *not* microkernels
- 6500 to 2 million times more dynamic instructions than benchmarks used in related work.



Avg. kernel calls = 4764
Avg. # of basic blocks = 13 Billion
Avg. GPU instructions = 308 Billion

Workload Analysis

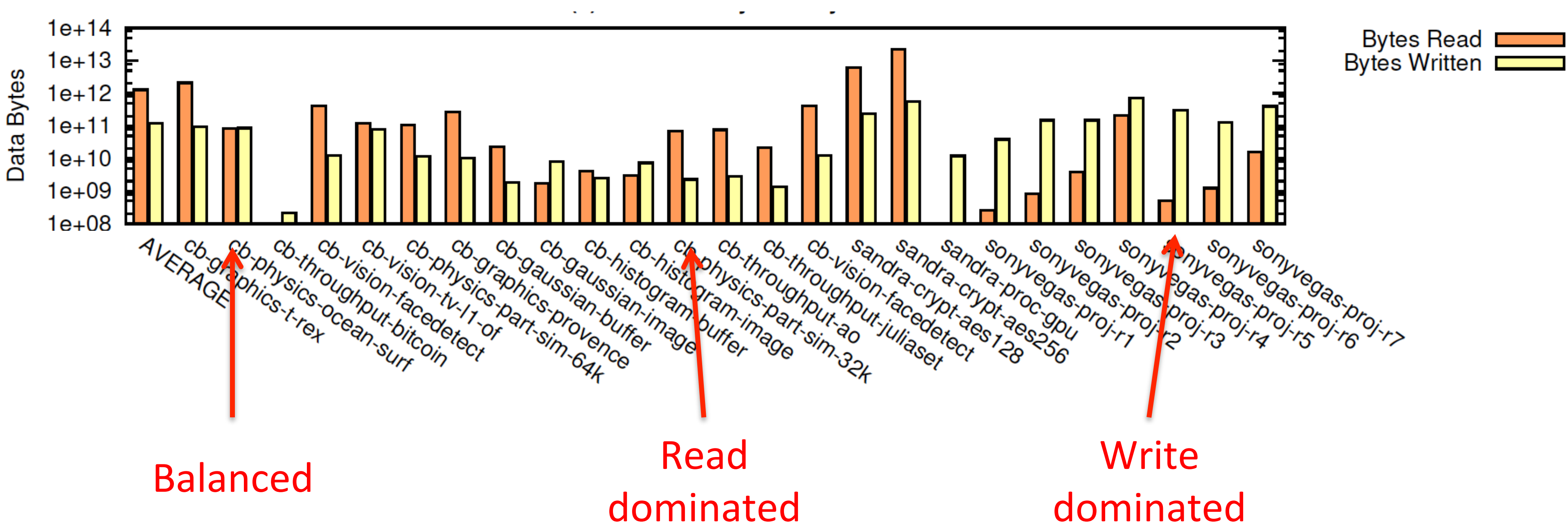
- Instruction mixes vary between applications



- Need to explore multiple apps for good HW design

Workload Analysis

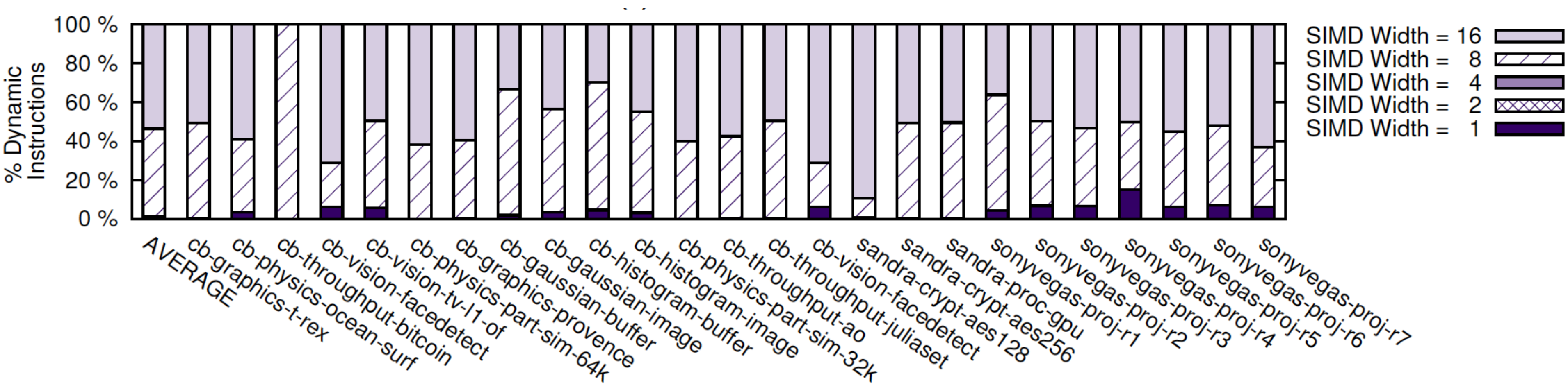
- Applications also vary with respect to read/write ratios:



- Need to explore multiple apps for good HW design

Workload Analysis

- We aren't always taking full advantage of data parallelism:



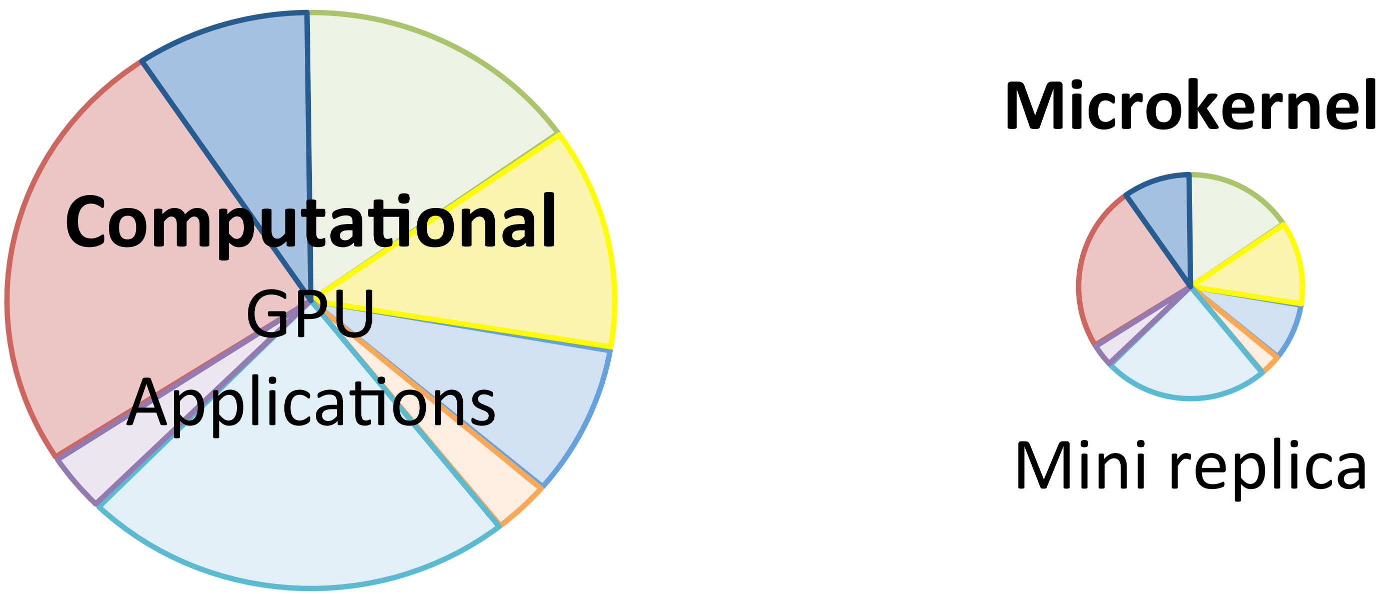
  - 52% instructions have 16-wide SIMD
  - 45% instructions have 8-wide SIMD
  - Remainder 4-wide or less

Talk Overview

- ~~Motivation~~
- ~~GT-Pin Tool~~
- ~~Sample Measurements~~
- GPU Simulation Acceleration

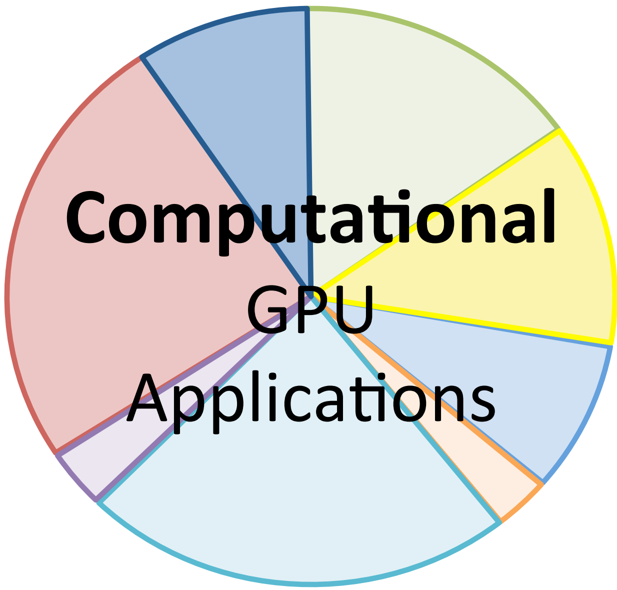
GPU Simulation acceleration

- GPUs are very slow to simulate
- Are microkernels a solution?

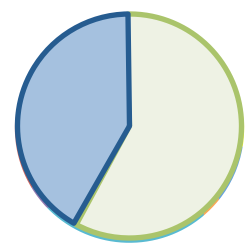


GPU Simulation acceleration

- GPUs are very slow to simulate
- ~~Are microkernels a solution?~~ Probably not.



Microkernel

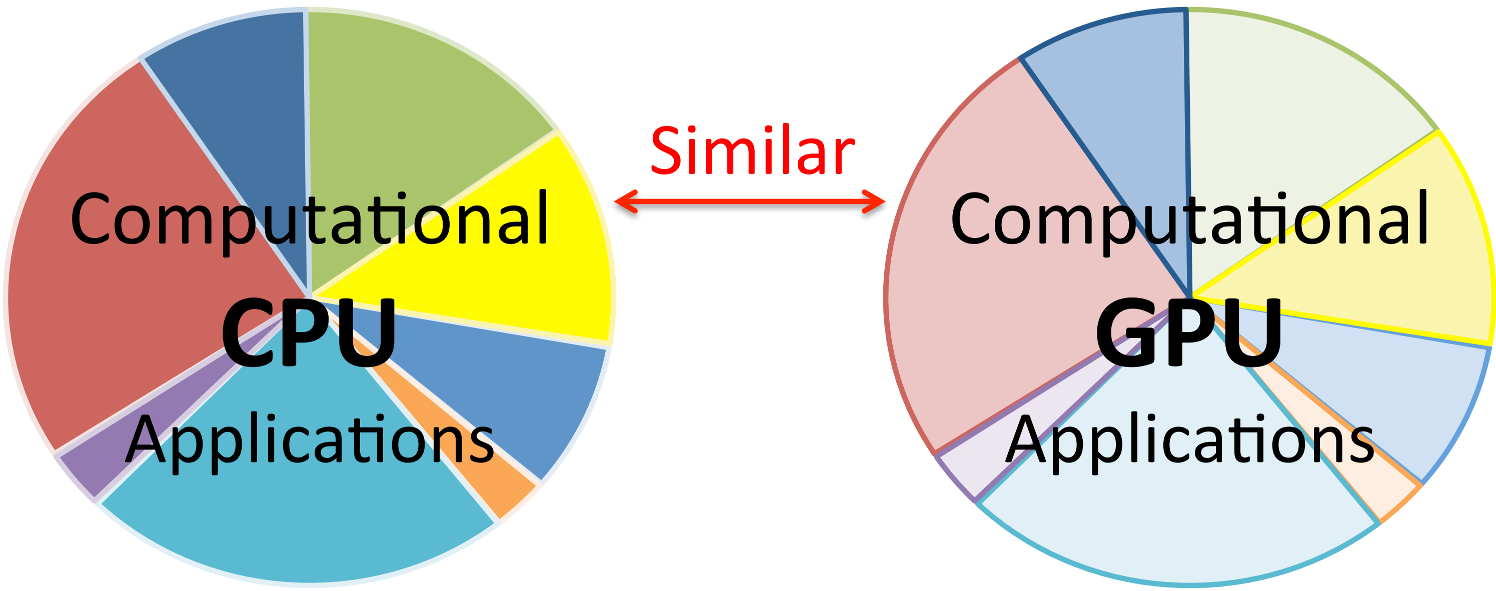


~~Mini replica~~

Not representative!

Solution: Representative Regions

- Use already known CPU region selection techniques, e.g. [Sherwood 2002, Patil 2004]



Background: CPU selection

GOAL: select small but representative regions of current applications so we don't have to simulate full programs when designing future HW.

Background: CPU selection

STEP 1: Trace program execution, gather performance statistics such as instruction & memory access counts

Background: CPU selection

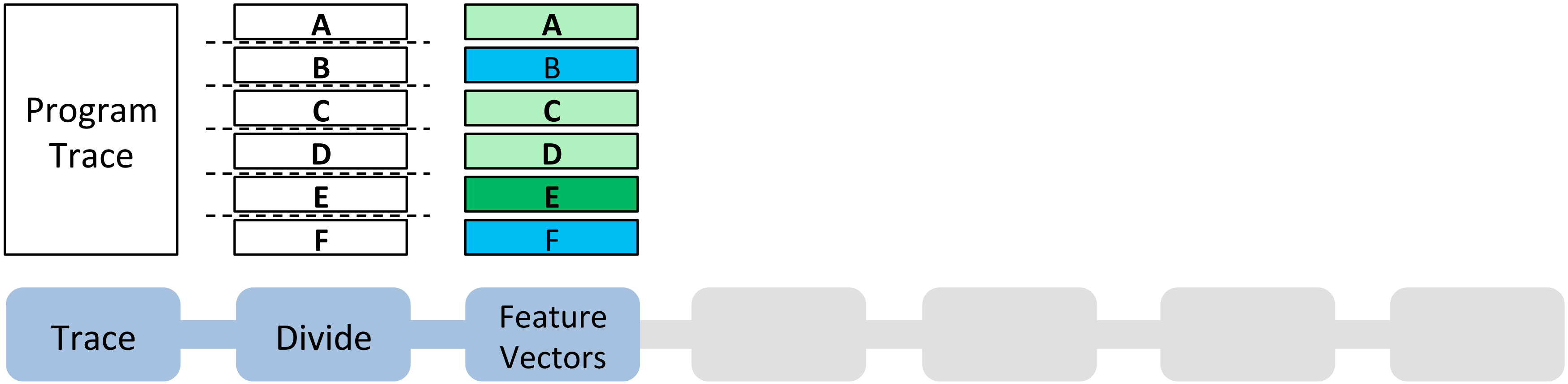
STEP 2: Divide program trace into *intervals,* e.g. break at every 100 million instructions.

Background: CPU selection

STEP 3: Quantify performance behavior with *feature vectors* **per interval**, e.g. *basic block vectors:*

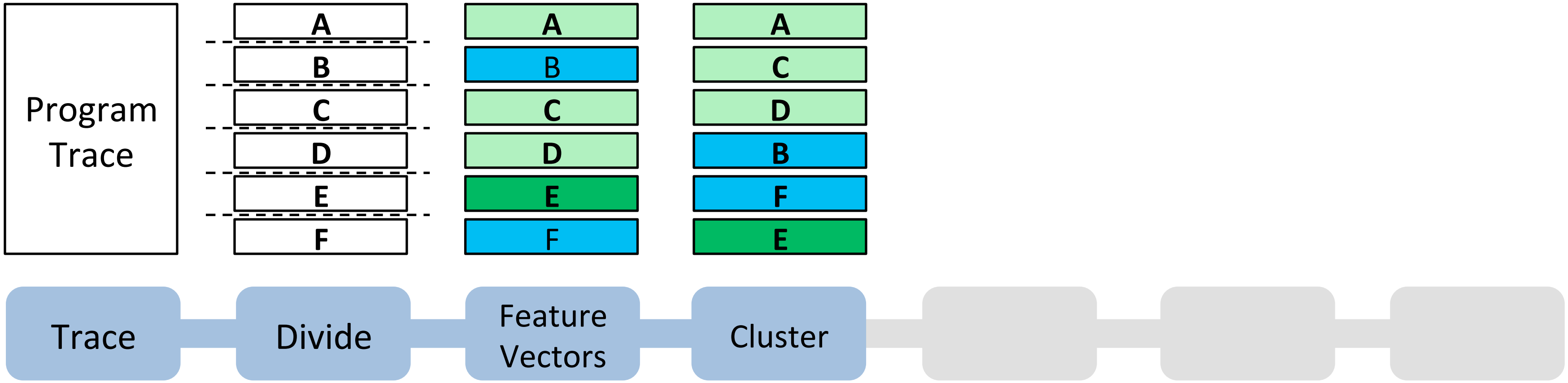
<unique block ID: basic block execs*instr/block>

<BB1:10, BB2:200, BB3:40, BB4:0, BB5:50>



Background: CPU selection

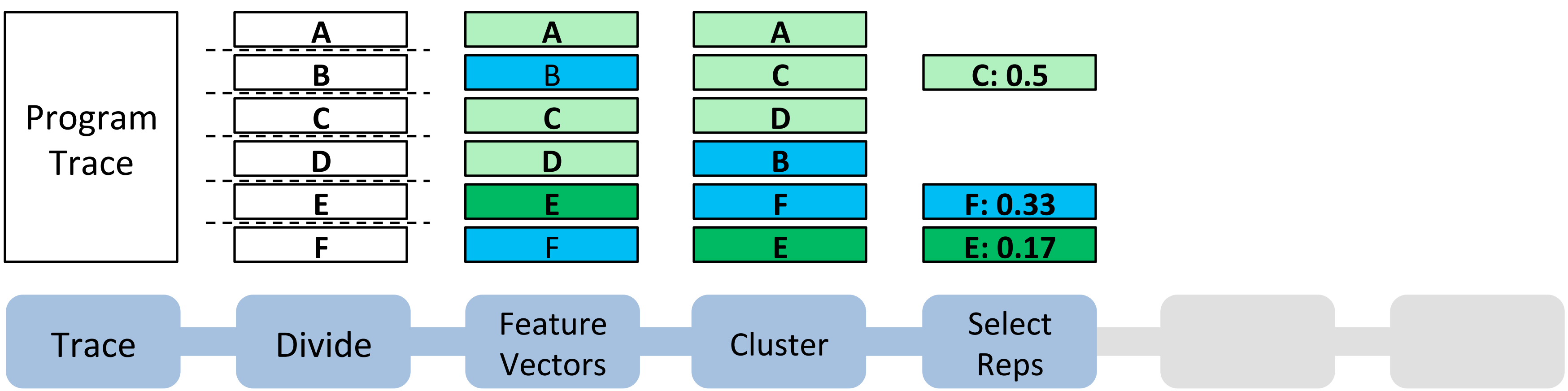
STEP 4: Cluster similar feature vectors. Use machine learning, e.g. k-means or hierarchical clustering.



Background: CPU selection

STEP 5: Select representative intervals per cluster, and compute associated weights per cluster.

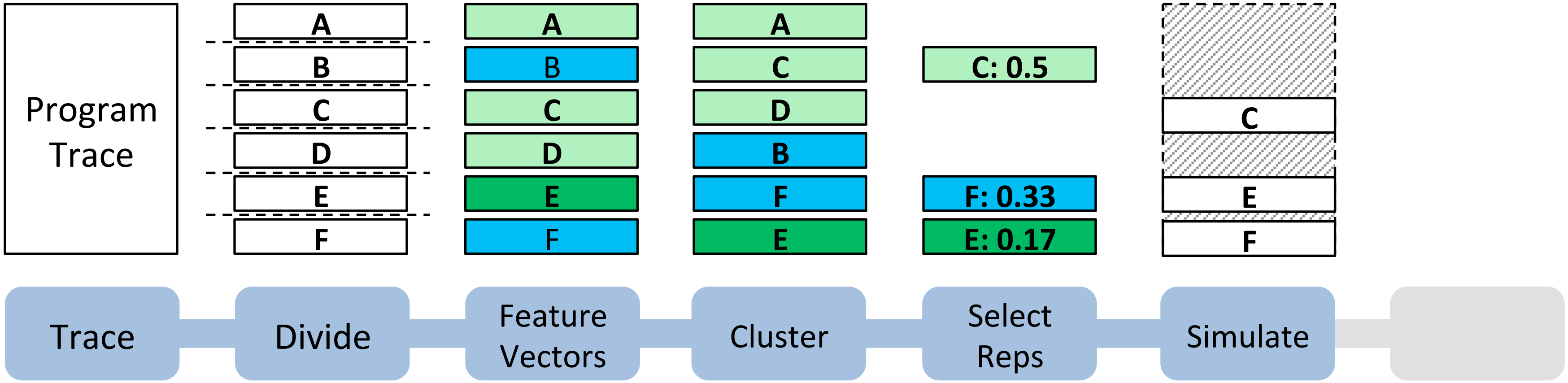
- Weight is a ratio (all weights sum to 1)
- Relative # of instructions in cluster vs. whole program



Background: CPU selection

STEP 6: Simulate the selected intervals in full, FF through the rest of the program. Record performance per selected interval, e.g.

CPI_C=0.5 CPI_E=0.7 CPI_F=0.4

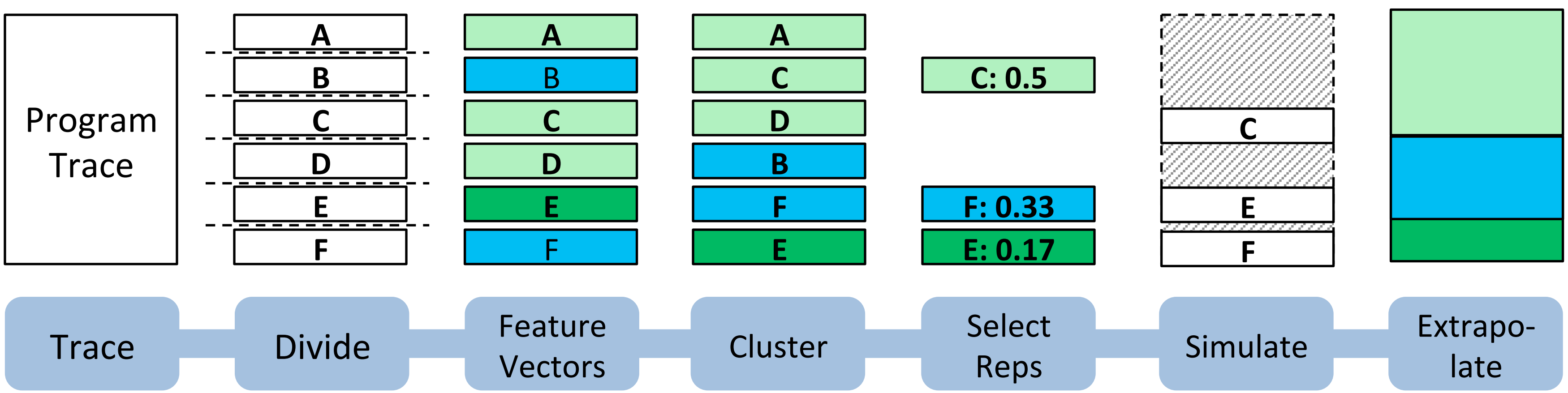


Background: CPU selection

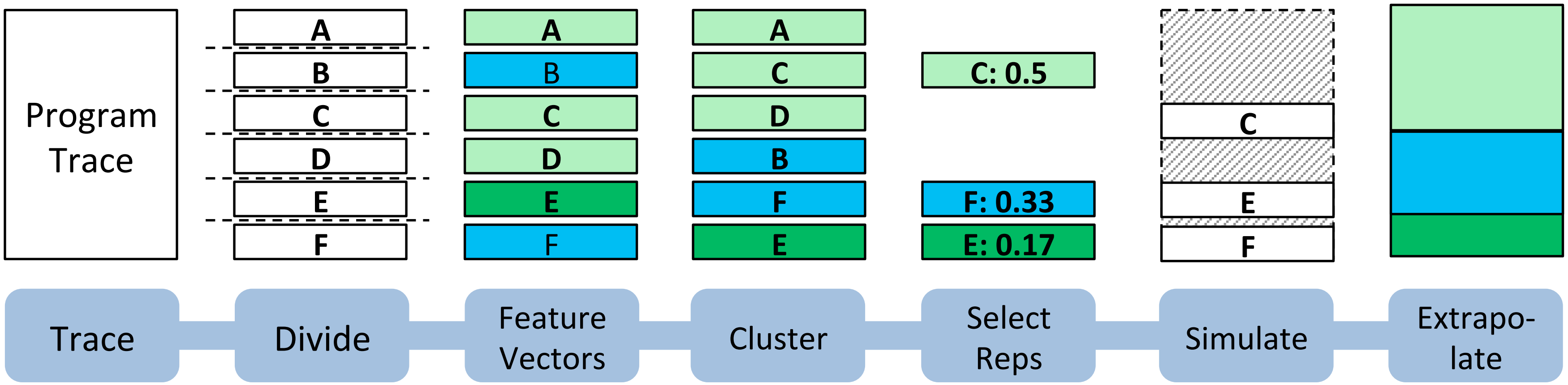
STEP 7: Extrapolate selected performance metrics to calculate whole program performance, e.g.,

$$CPI_C=0.5 \qquad CPI_E=0.7 \qquad CPI_F=0.4$$

$$CPI_{total} = (0.5*0.5) + (0.7*0.17) + (0.4*0.33) = 0.501$$

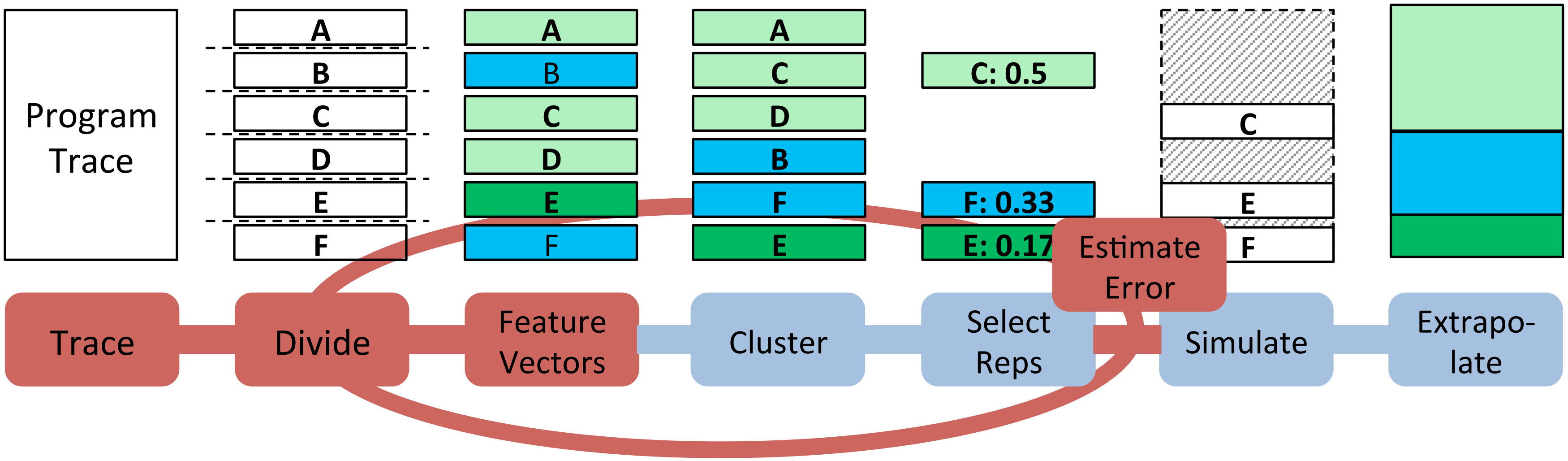


Adapting the CPU algorithm to GPUs



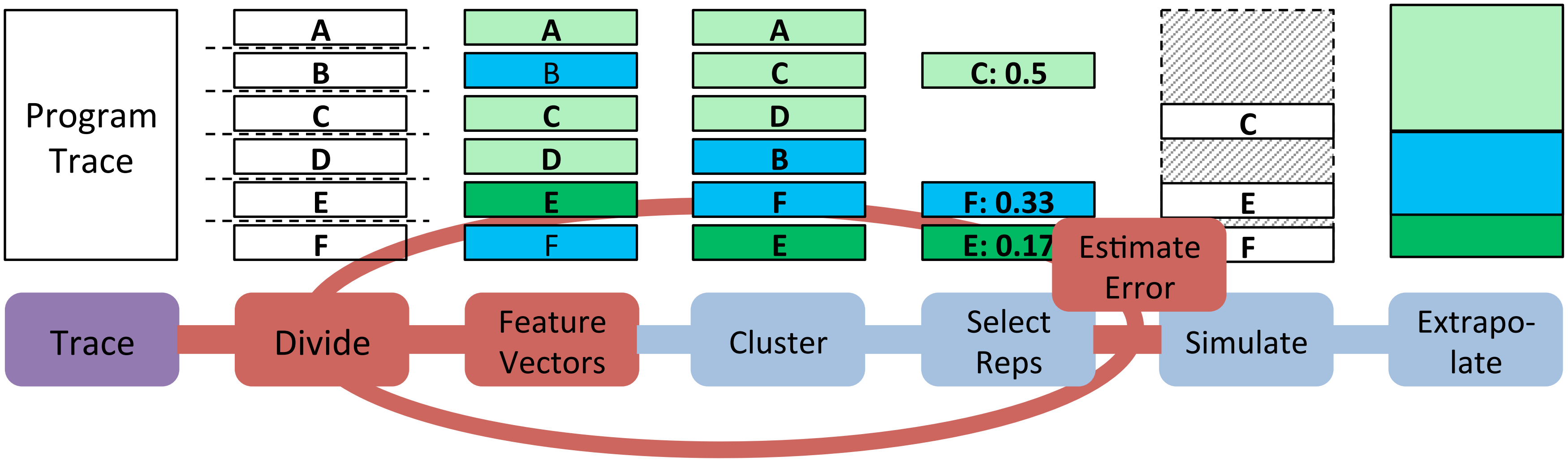
To adapt this process to GPUs, we had to make several adjustments.

Adapting the CPU algorithm to GPUs



To adapt this process to GPUs, we had to make several adjustments.

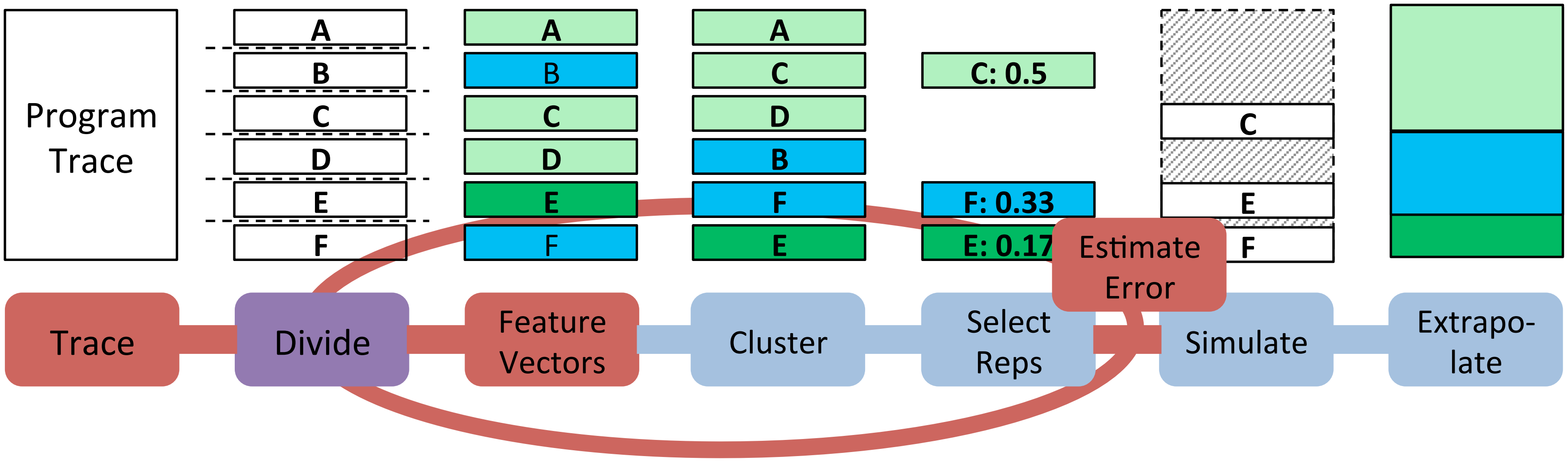
Trace with GT-Pin



In Trace Step, we use GT-Pin to collect:

- Ordered API trace, API call count
- Unique kernel count & frequency
- Dynamic & static instruction count
- Basic block executions
- Bytes read & written per instruction

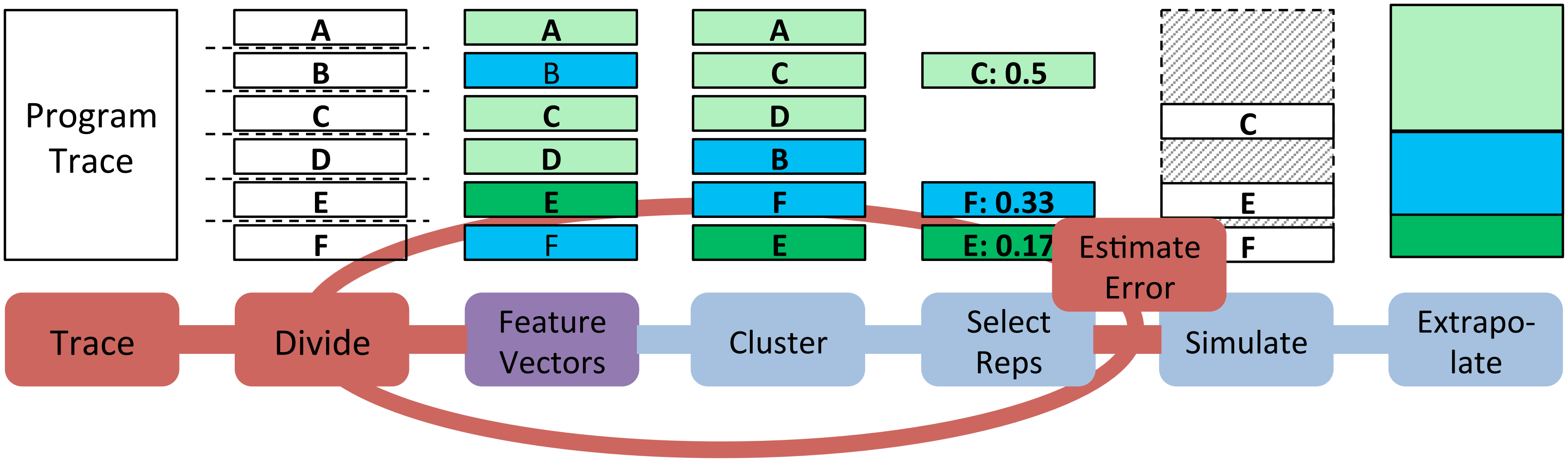
Explore 3 Division Sizes



In Divide Step, we explore multiple interval divisions of API Call trace:

1) Large: divide at *synchronization* calls.
2) Medium: divide approx. every 100M instructions
3) Small: divide at each kernel

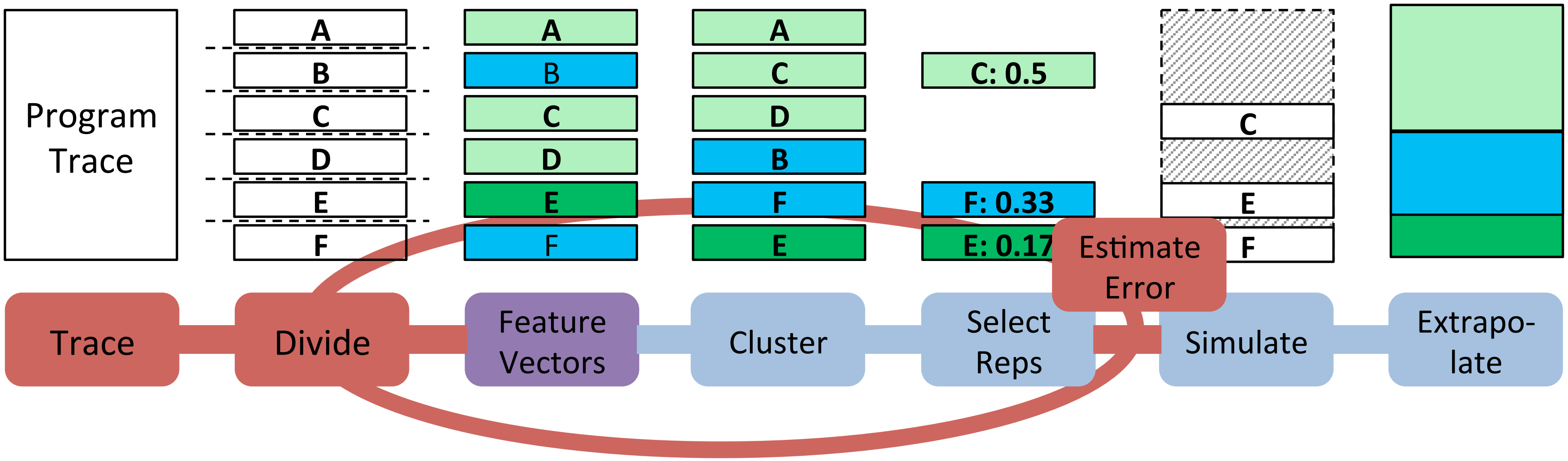
Explore 10 Feature Vectors



Also explore a number of Feature Vectors:

1) Unique kernels [KN]
2) Unique kernels with the same arguments [KN-ARGS]
3) Unique kernels with the same *global work size* [KN-GWS]
4) Unique kernels with same arguments & global work size [KN-ARGS-GWS]
5) Unique basic blocks (i.e. basic block vectors) [BB]

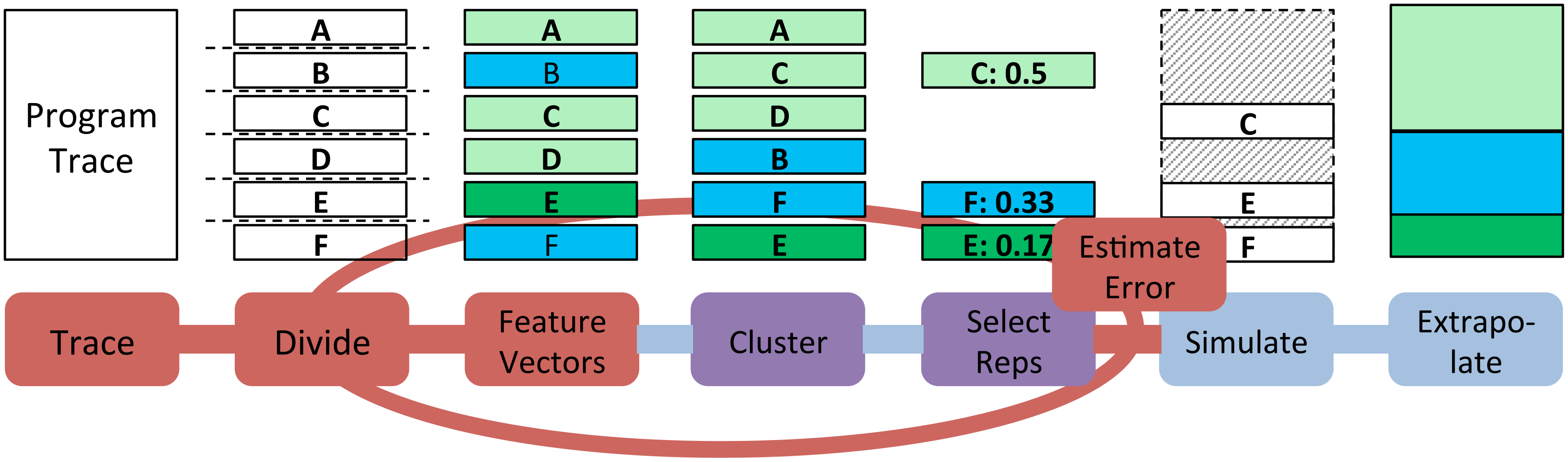
Explore 10 Feature Vectors



Including some Feature Vectors with memory accesses:

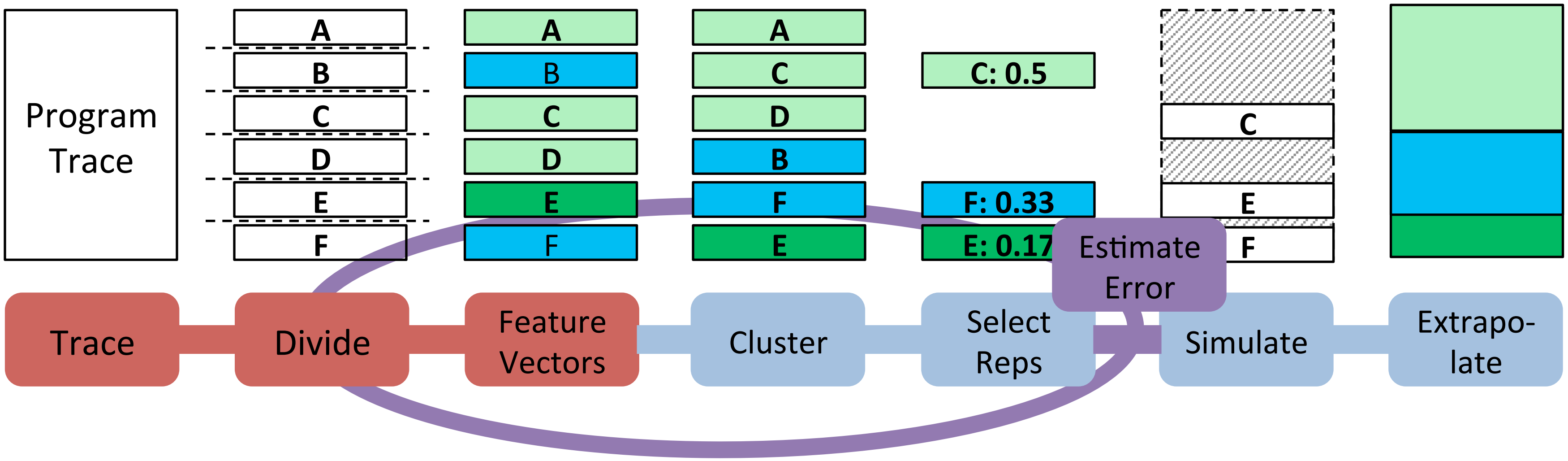
6) Unique BBs and matching bytes written [BB-W]
7) Unique BBs and matching bytes read [BB-R]
8) Unique BBs and matching total bytes (read + written) [BB-R+W]
9) Unique BBs and matching both bytes written & read [BB-RW]
10) Unique kernels and matching both bytes written & read [KN-RW]

Select Representative Regions



We use SimPoint, open source academic software designed for CPU simulation region selection, to group intervals into clusters, and to select representatives & weights

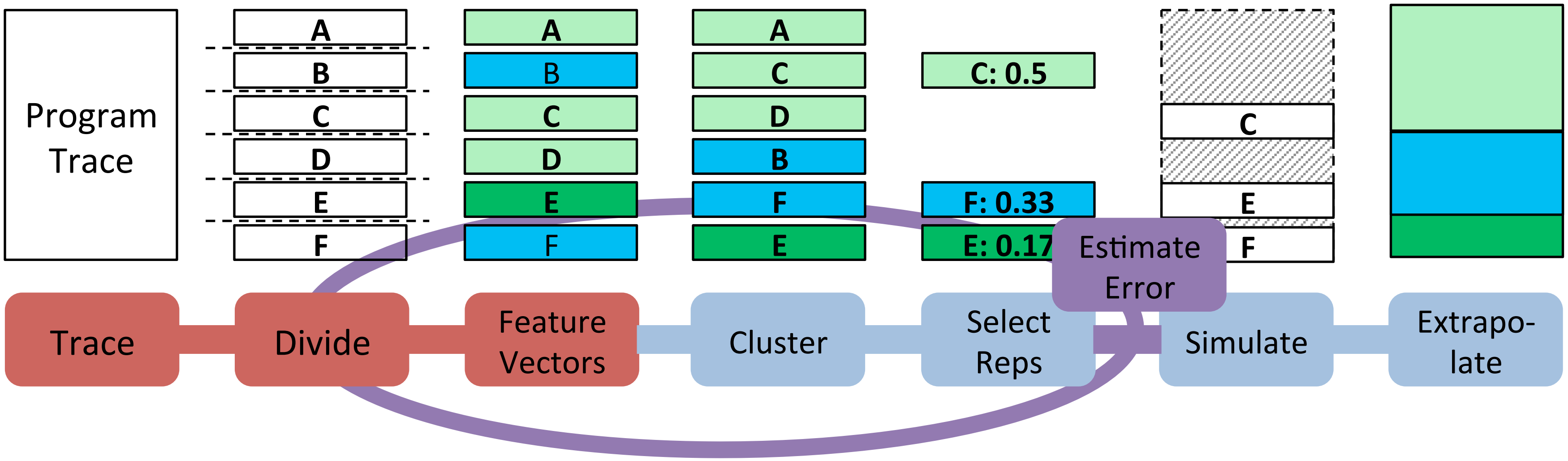
Choose best division size/feature vector combination



To choose best of 30 division size/feature vector combinations, compare performance of extrapolated selection to measured performance of whole program:

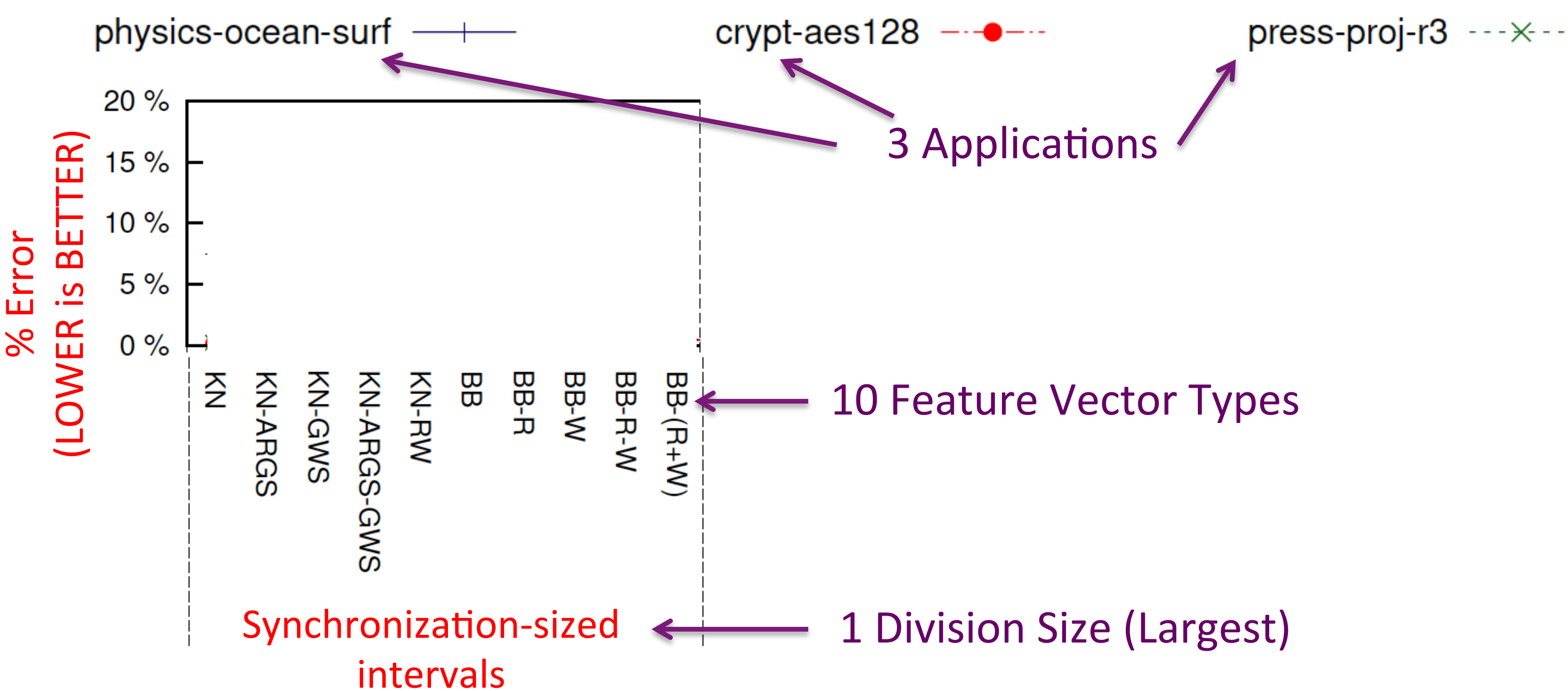
$$error = \left(\frac{Perf_{extrapolated} - Perf_{measured}}{Perf_{measured}} \right) \times 100\%$$

Adapting the CPU algorithm to GPUs



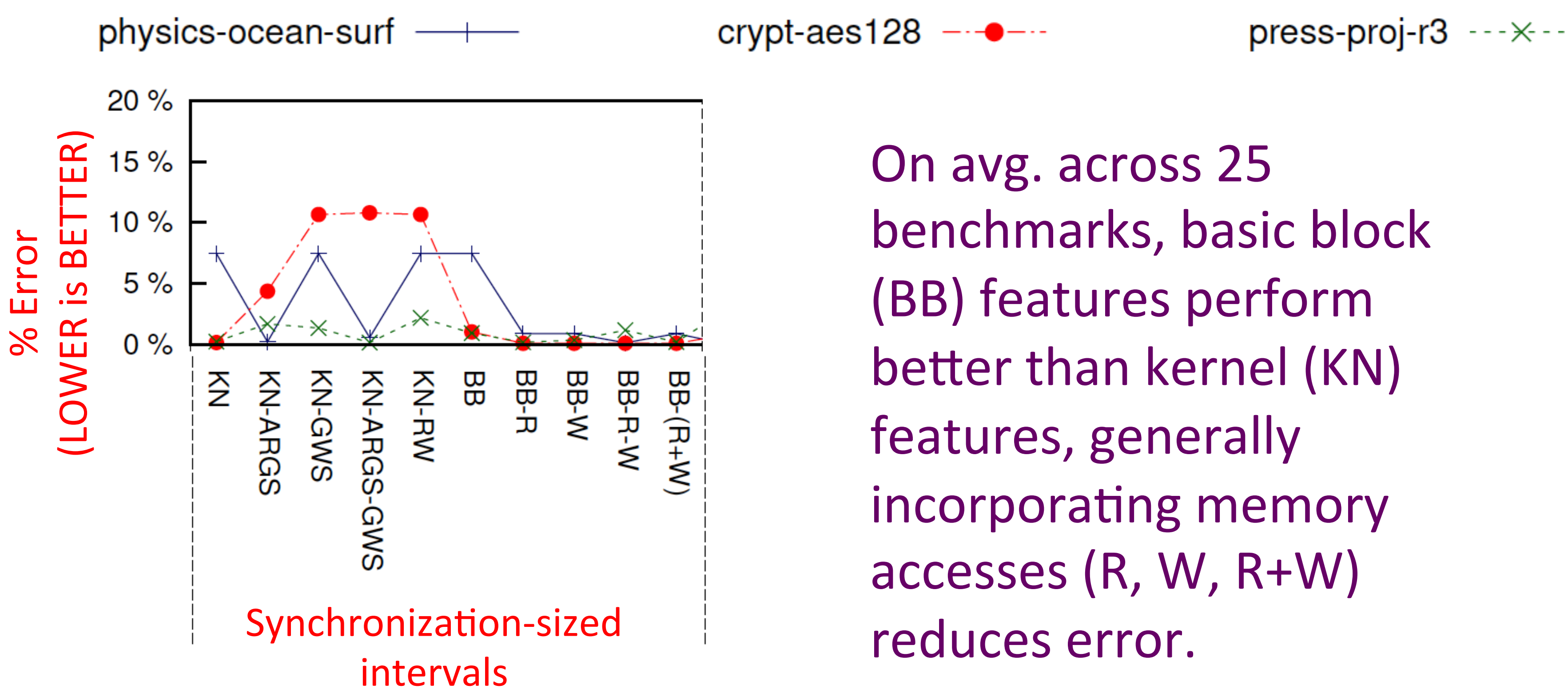
- Typically error performance measurements done via simulation, but this is slow.
- Instead, we use a kernel time measurement tool called Intel CoFluent CPR to validate the selections in native time.

Results



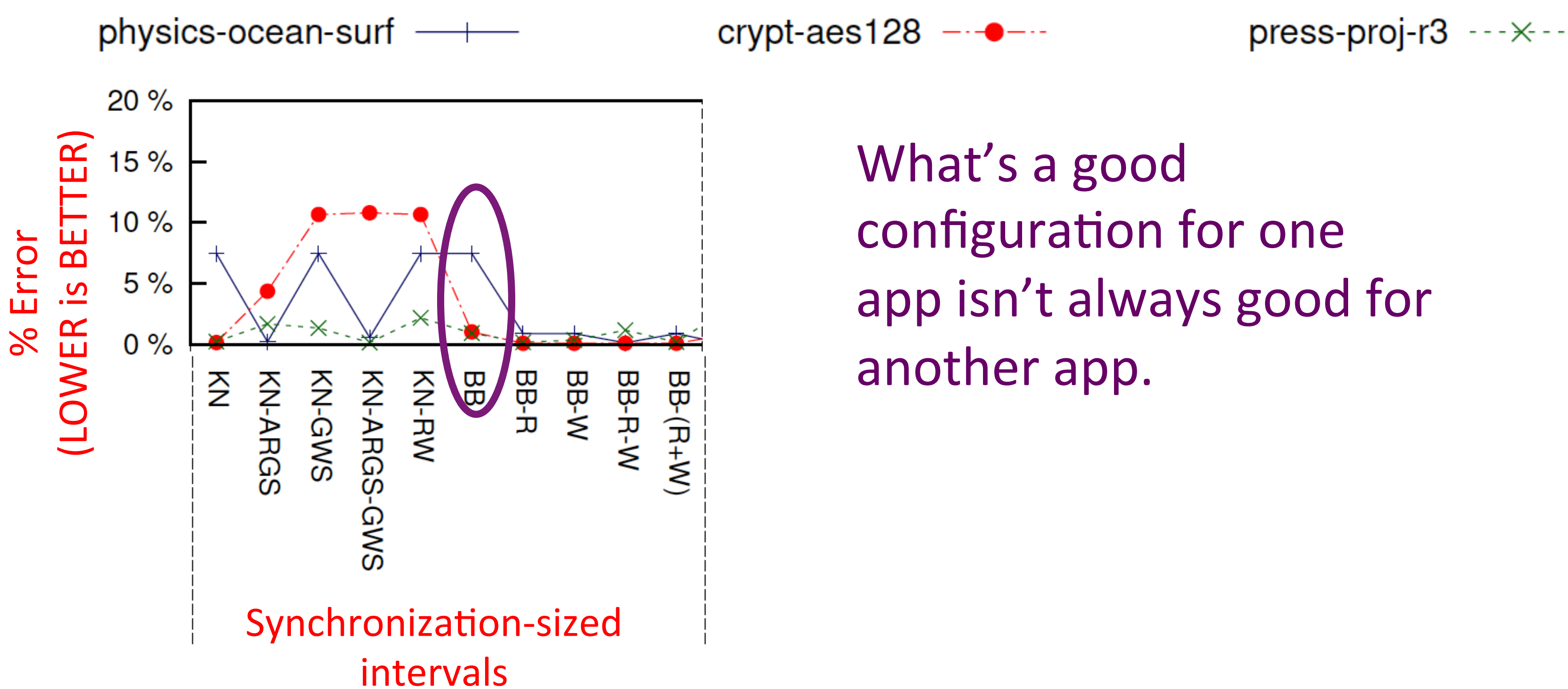
physics-ocean-surf
crypt-aes128
press-proj-r3
3 Applications
20 %
15 %
10 %
5 %
0 %
% Error
(LOWER is BETTER)
KN
KN-ARGS
KN-GWS
KN-ARGS-GWS
KN-RW
BB
BB-R
BB-W
BB-R-W
BB-(R+W)
10 Feature Vector Types
Synchronization-sized intervals
1 Division Size (Largest)

Results



On avg. across 25 benchmarks, basic block (BB) features perform better than kernel (KN) features, generally incorporating memory accesses (R, W, R+W) reduces error.

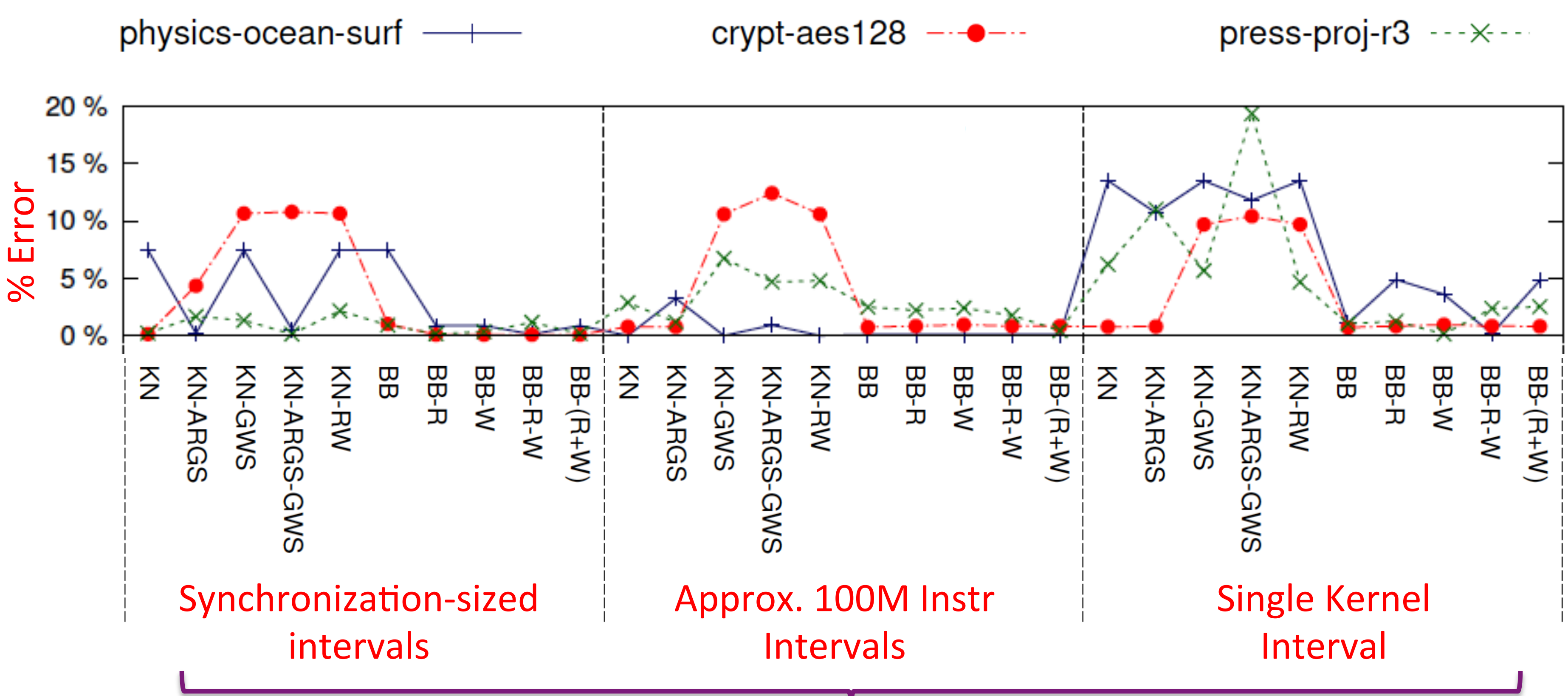
Results



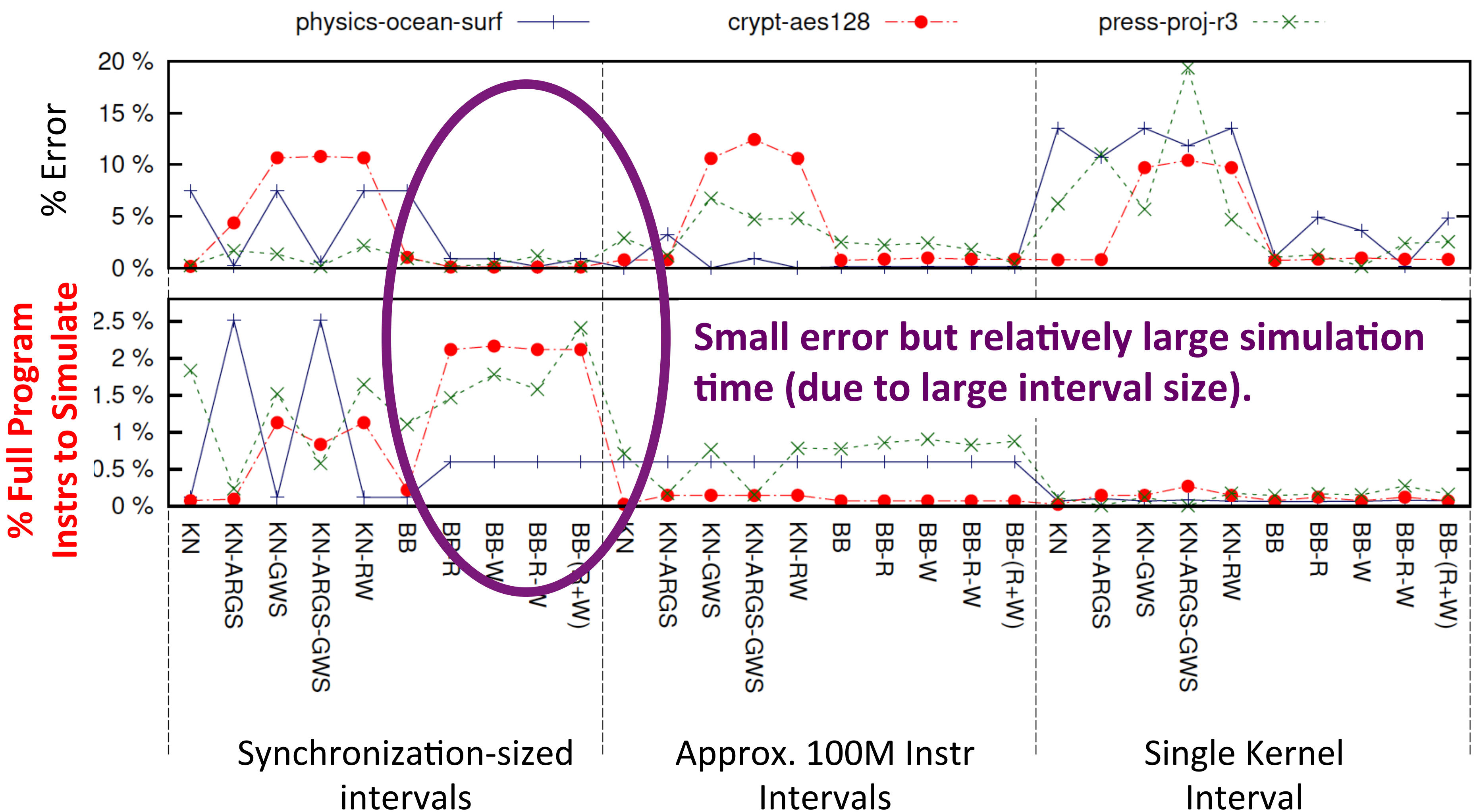
What's a good configuration for one app isn't always good for another app.

Results: Full Exploration Space

"Best configuration" also varies for other interval sizes.



Also consider selection size; “best config” in terms of error is not always best in terms of minimizing simulation time.

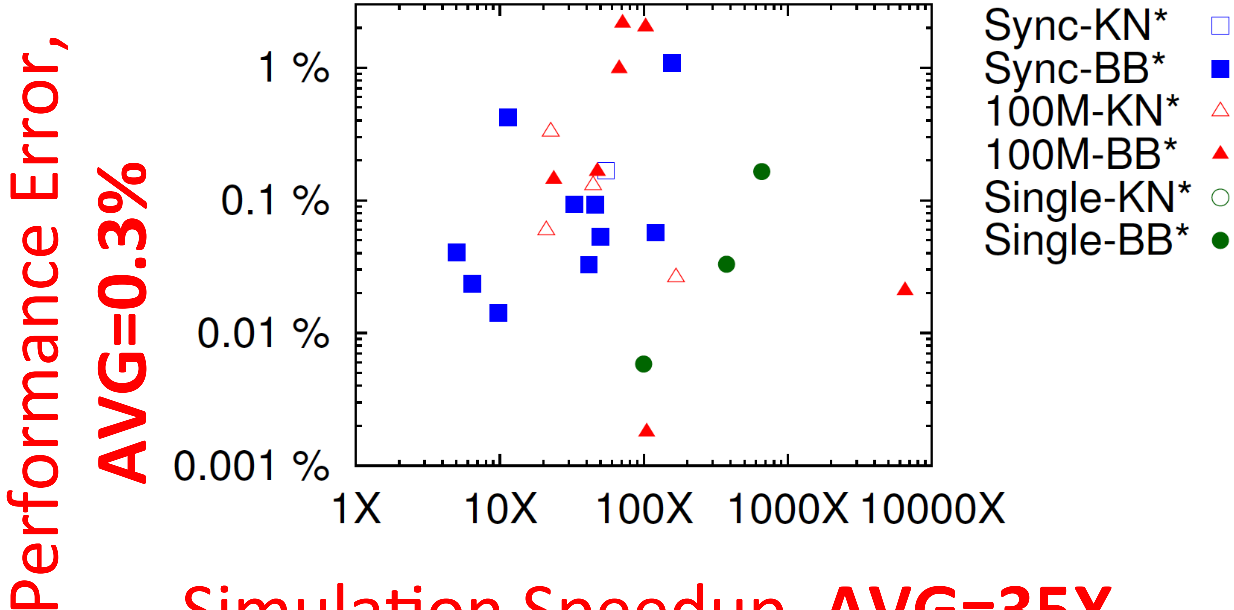


Don’t have to choose one configuration

- Instead of picking best selection size/feature vector for *all* apps, pick best for *each* app.
- Can this because of fast (no simulation) validation

Don’t have to choose one configuration

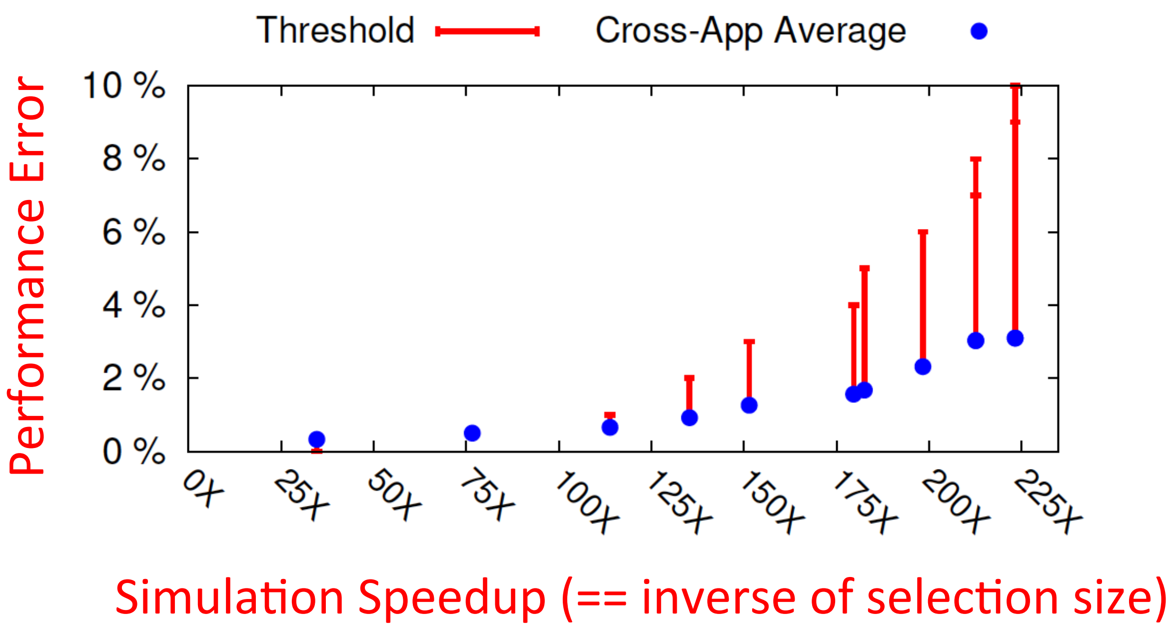
- Instead of picking best selection size/feature vector for *all* apps, pick best for *each* app.
- Can this because of fast (no simulation) validation



Simulation Speedup, **AVG=35X**
(== inverse of selection size)

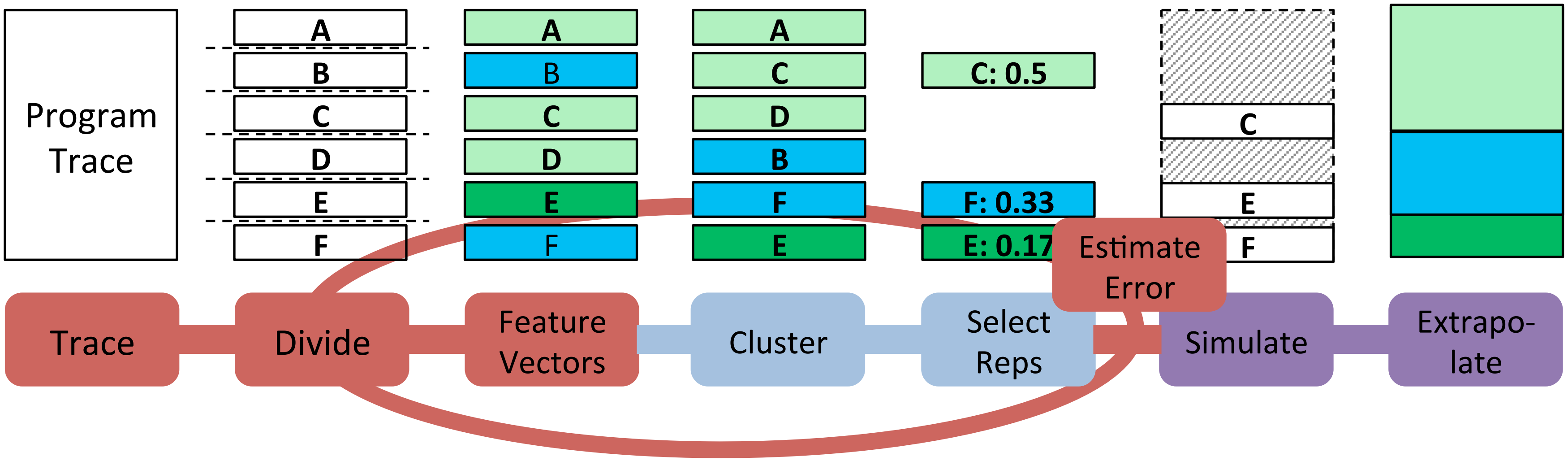
Don't have to choose lowest error

- Instead of choosing lowest error config, can tradeoff between low error and small selection size (i.e., bigger speedup).



- For example, if 3% error is acceptable, average simulation speedup is 223X

Adapting the CPU algorithm to GPUs



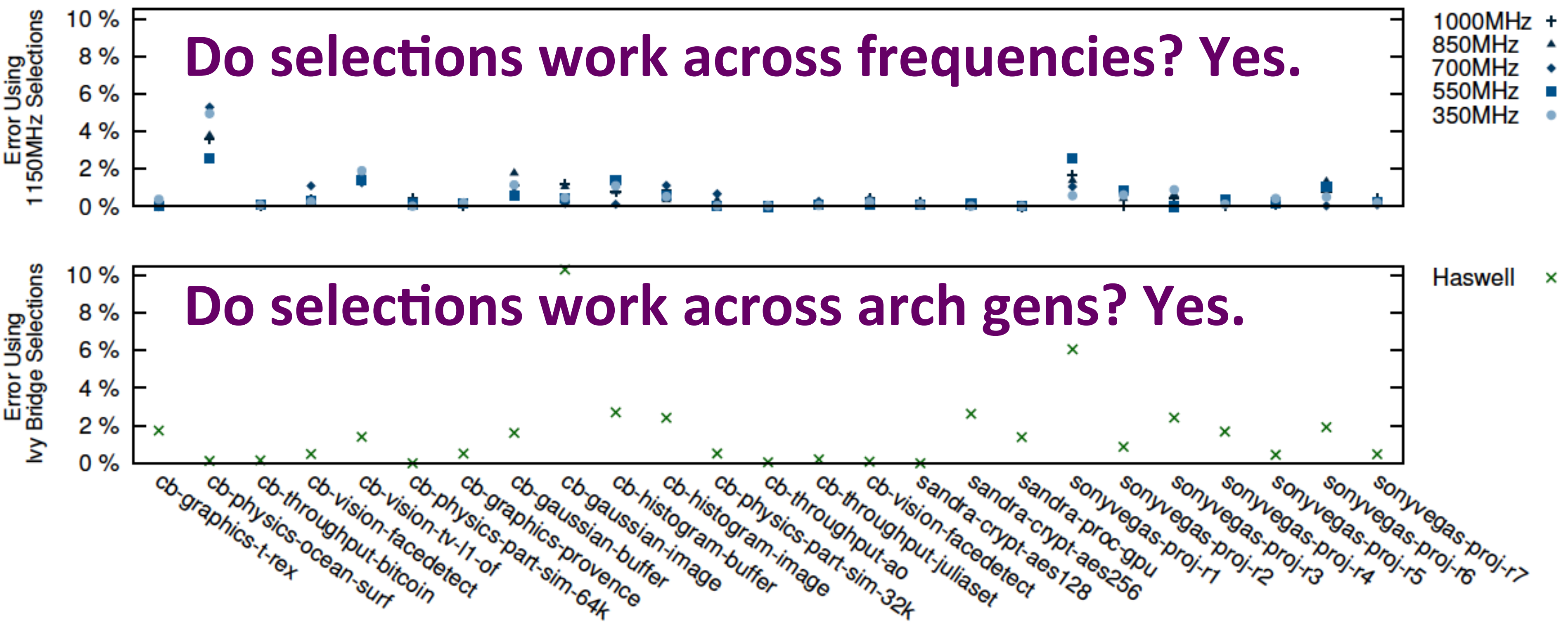
- If selection criteria are good, only need to **select regions on one architecture** for each program
- Can then **use for** simulating/extrapolating **all future architecture** designs' performance

Are selections valid for future HW?

- Does performance *extrapolated* from selections at one frequency (1150 MHz) match *measured* performance of other frequencies?

- Does performance *extrapolated* from selections on one architecture generation (Ivy Bridge) match *measured* performance of future architecture generations (Haswell)?

Are selections valid for future HW?



Summary

Real computational GPU programs are very large and more diverse than graphics apps.

- To evaluate them, we need fast detailed analysis → GT-Pin tool.
- To simulate them, and improve HW design, we need GPU specific region selection methods.

Questions?

Paper Title: *Fast Computational GPU Design with GT-Pin*

melanie@cs.columbia.edu

Intel contacts:

- Chi-Keung (CK) Luk: chi-keung.luk@intel.com
- Sunpyo Hong: sunpyo.hong@intel.com

Notable trace internals

clEnqueueReadBuffer
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clFinish
clEnqueueReadBuffer
clEnqueueWriteBuffer
clEnqueueWriteBuffer
clEnqueueWriteBuffer
clSetKernelArg
…
clSetKernelArg
clEnqueueNDRangeKernel
clEnqueueReadBuffer
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clFinish

- “clEnqueueNDRangeKernel” calls define **GPU** work, (remaining cl* calls work on host device, e.g. **CPU**)
- Synchronization calls (e.g. clEnqueueRead-Buffer, clFinish) coordinate CPU/GPU work

Division 1: Synchronization Intervals

clEnqueueReadBuffer
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel

2 Kernels

---

clFinish
clEnqueueReadBuffer

0 Kernels

---

clEnqueueWriteBuffer
clEnqueueWriteBuffer
clEnqueueWriteBuffer
clSetKernelArg
…
clSetKernelArg
clEnqueueNDRangeKernel

1 Kernel

---

clEnqueueReadBuffer
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel

2 Kernels

---

clFinish

Look at kernels inside consecutive synchronization calls.

Trace → Divide → Feature Vectors → Cluster → Select Reps → Simulate → Extrapo-late

Division 2: ~100M Instr Intervals

clEnqueueReadBuffer
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel

1 Kernel

clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel

1 Kernel

clFinish
clEnqueueReadBuffer

0 Kernels

clEnqueueWriteBuffer
clEnqueueWriteBuffer
clEnqueueWriteBuffer
clSetKernelArg
…
clSetKernelArg
clEnqueueNDRangeKernel

1 Kernel

clEnqueueReadBuffer
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel
clSetKernelArg
clSetKernelArg
clEnqueueNDRangeKernel

2 Kernels

clFinish

Break synch intervals further if sum of kernels' dynamic is instructions is > 100M

Trace — Divide — Feature Vectors — Cluster — Select Reps — Simulate — Extrapo-late

Division 3: Single Kernel Intervals

API calls	Interval
clEnqueueReadBuffer clSetKernelArg clSetKernelArg clEnqueueNDRangeKernel	1 Kernel
clSetKernelArg clSetKernelArg clEnqueueNDRangeKernel	1 Kernel
clFinish clEnqueueReadBuffer	0 Kernels
clEnqueueWriteBuffer clEnqueueWriteBuffer clEnqueueWriteBuffer clSetKernelArg … clSetKernelArg clEnqueueNDRangeKernel	1 Kernel
clEnqueueReadBuffer clSetKernelArg clSetKernelArg clEnqueueNDRangeKernel	1 Kernel
clSetKernelArg clSetKernelArg clEnqueueNDRangeKernel	1 Kernel
clFinish	

Divide until each interval is one kernel.

Trace → Divide → Feature Vectors → Cluster → Select Reps → Simulate → Extrapo-late

Feature vector creation

clEnqueueNDRangeKernel [KernelName = A]

BB #1	BB #2	BB #3	BB #4

From GTPin Traces

clEnqueueNDRangeKernel [KernelName = B]

BB #1	BB #2	BB #3

Feature vector with kernel names:

Kernel_A: 1, Kernel_B: 1

Feature vector with basic blocks

A_1: 1, A_2: 100, A_3: 2, A_4:20, B_1: 4, B_2: 80, B_3:7

Trace → Divide → Feature Vectors → Cluster → Select Reps → Simulate → Extrapo-late

Feature vector creation

clEnqueueNDRangeKernel [KernelName = A]

BB #1	BB #2	BB #3	BB #4
10	2	10	30

clEnqueueNDRangeKernel [KernelName = B]

BB #1	BB #2	BB #3
20	20	10

Then **weight** by static instruction count (*again, get these counts from GTPin traces*).

Feature vector with kernel names:

Kernel_A: 52, Kernel_B: 50

Feature vector with basic blocks:

A_1: 10, A_2: 200, A_3: 20, A_4: 600,

B_1: 80, B_2: 1600, B_3:70

Trace — Divide — Feature Vectors — Cluster — Select Reps — Simulate — Extrapo-late

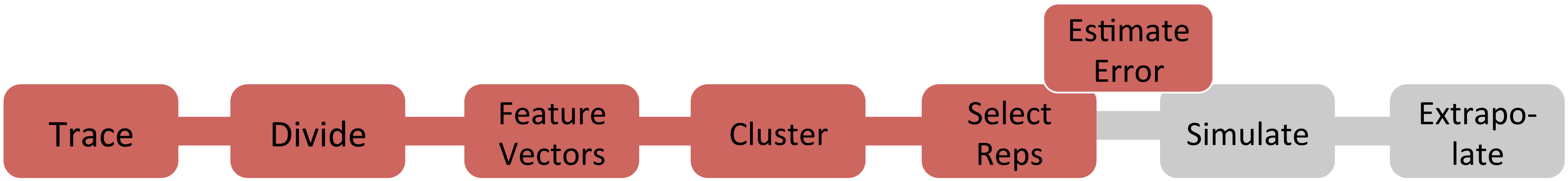
Intel CoFluent CPR

- To supplement GT-Pin profiling data, also use CoFluent CPR
- CoFluent outputs:
  1) Ordered API traces
  2) Seconds per kernel executed
- Use this data for our error feedback and validation.
- Guarantees repeatability through record/replay mechanism: rerun program (on new HW), same API call execution order, same inputs.

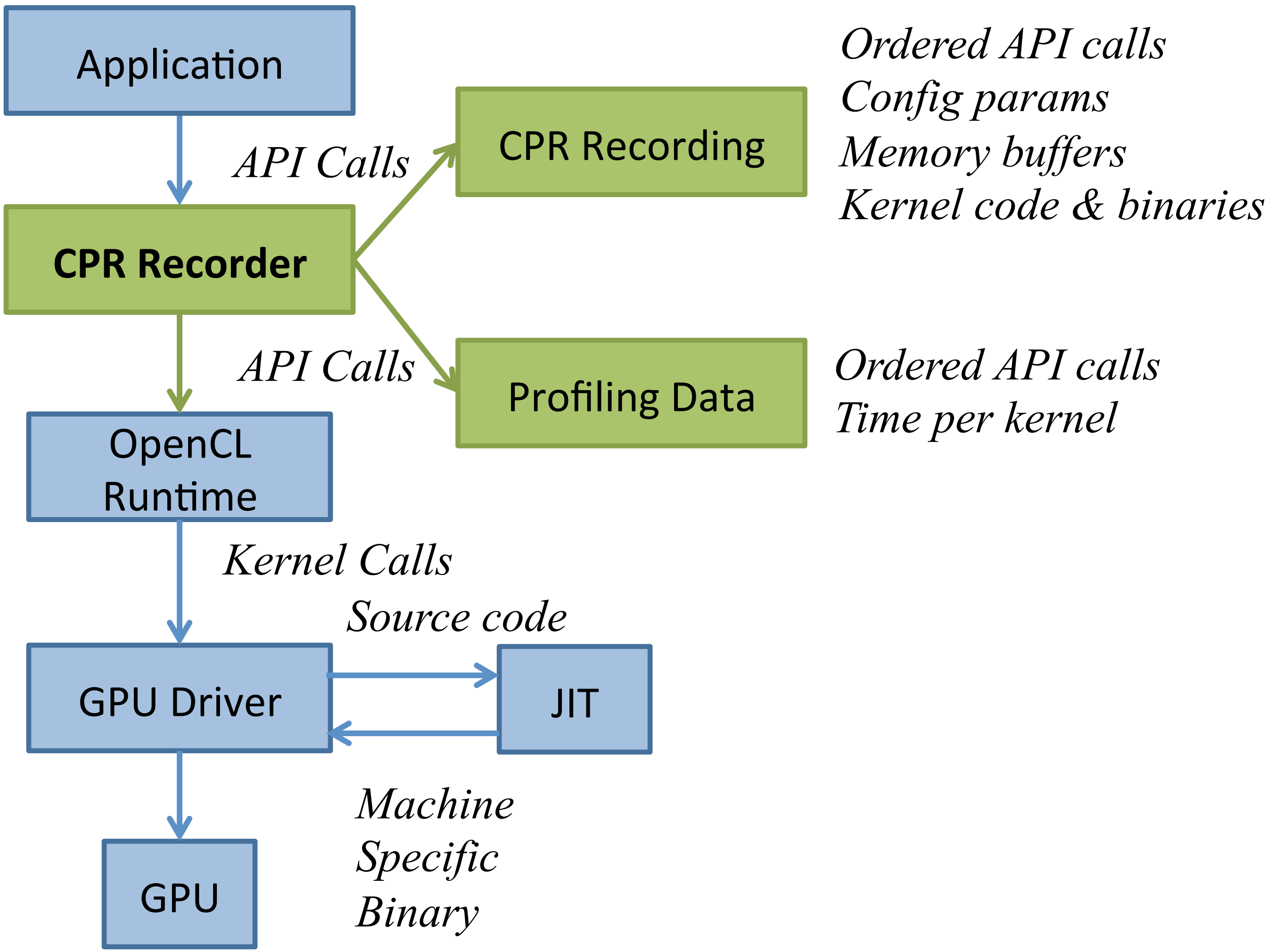
Extrapolate whole-program performance using selections

- To get measured whole program SPI:
  - Divide total seconds (sum of kernel seconds from CoFluent) by total dynamic instrs (from GT-Pin)

- To get projected whole program SPI:
  - **Per selected interval,** calculate seconds/dynamic instructions
  - Multiply interval SPI by SimPoint weight
  - Sum the weighted, selected interval SPIs

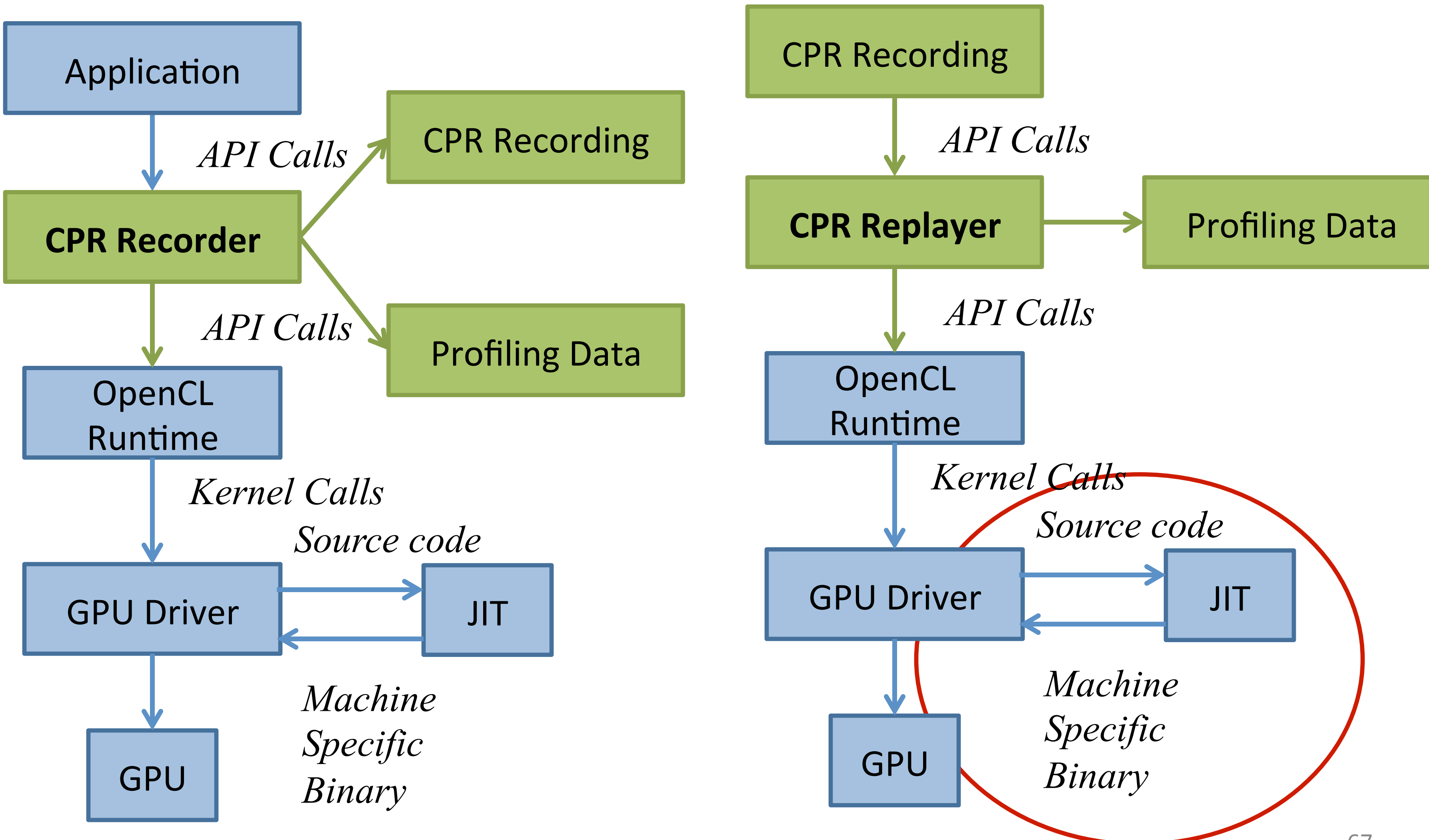


CoFluent one-time recording



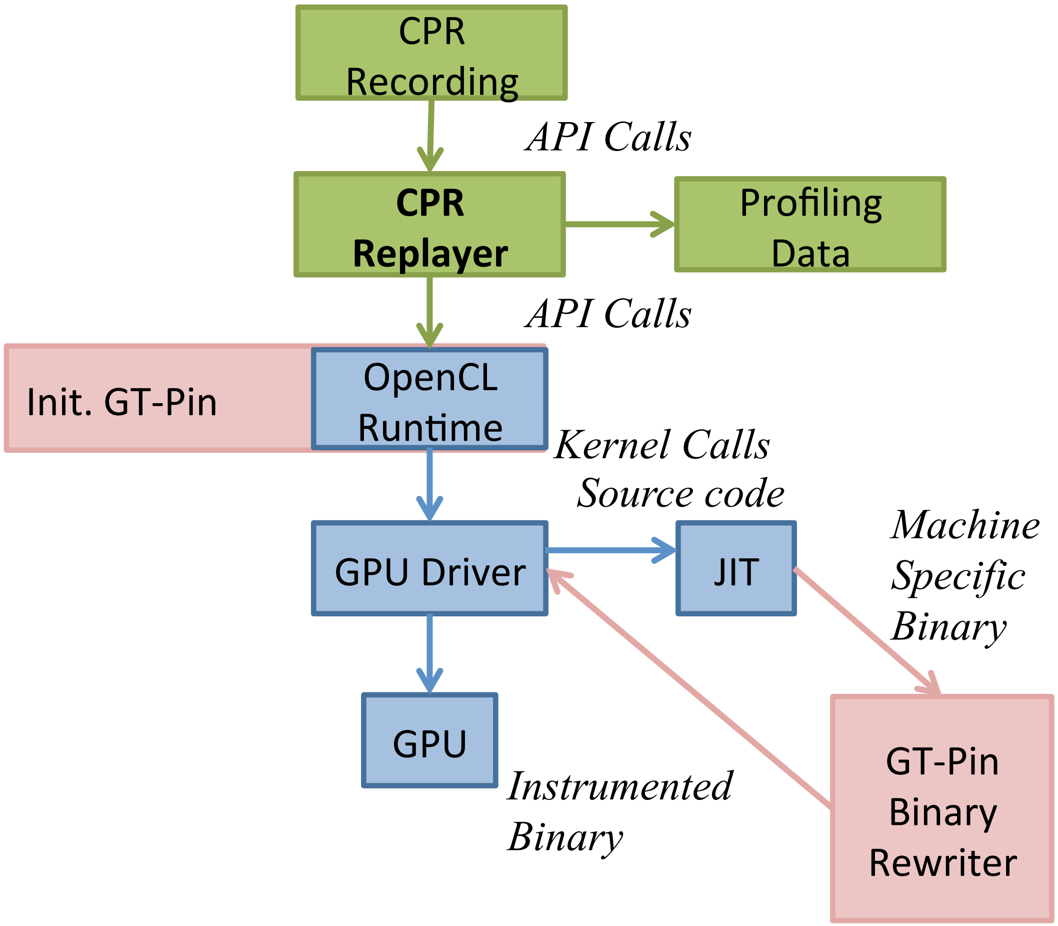
66

Repeatable replay (on any arch)



67

CoFluent + GT-Pin + OpenCL



68