A Symbolic Emulator for Shuffle Synthesis on the NVIDIA PTX Code

Kazuaki Matsumura*, Simon Garcia de Gonzalo**, Antonio J. Peña*

*Barcelona Supercomputing Center (BSC), **Sandia National Laboratories

kmatsumura@nvidia.com, simgarc@sandia.gov, antonio.pena@bsc.es

25/2/23 @ CC 2023

Funded by the Ministerio de Ciencia e Innovación-Agencia Estatal de Investigación (PID2019-107255GB-C21/AEI/10.13039/501100011033). Carried out on the ACME cluster owned by CIEMAT and funded by the Spanish Ministry of Economy and Competitiveness project CODEC-OSE (RTI2018-096006-B-I00).

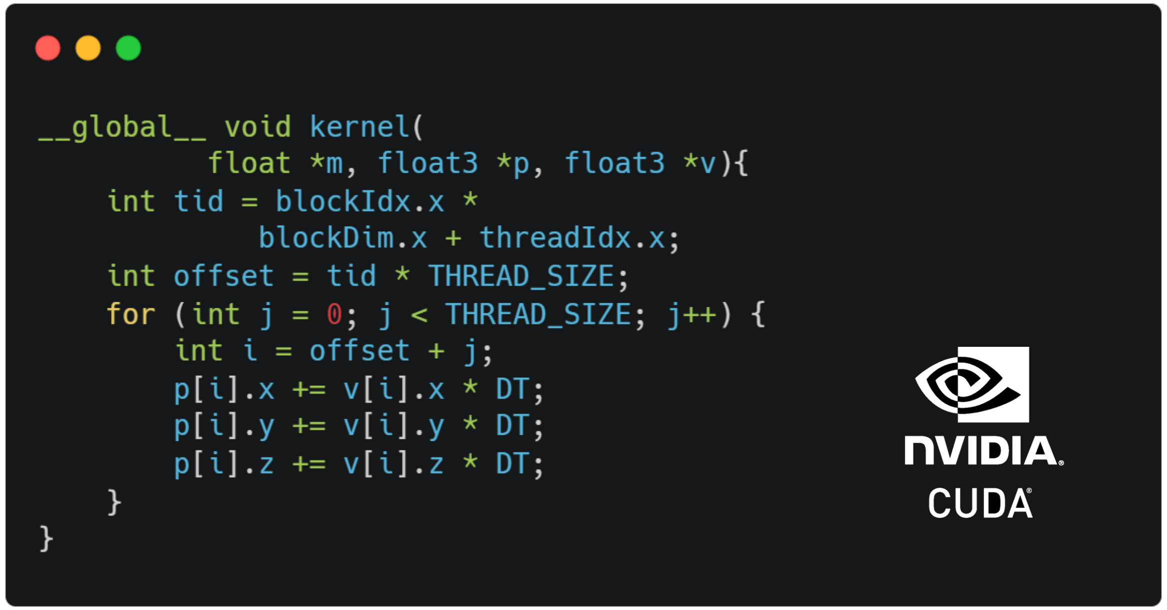
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The EPEEC project has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement N° 801051

Introduction

- **Modern supercomputers employ heterogeneous designs**
  - Execution performed on CPUs & *Accelerators* (e.g. GPUs, FPGAs)
- Example: **BSC’s MareNostrum 4**
  - One part consists of IBM POWER9 processors + NVIDIA Volta GPUs
  - Another part: AMD Rome processors + AMD Radeon Instinct MI50

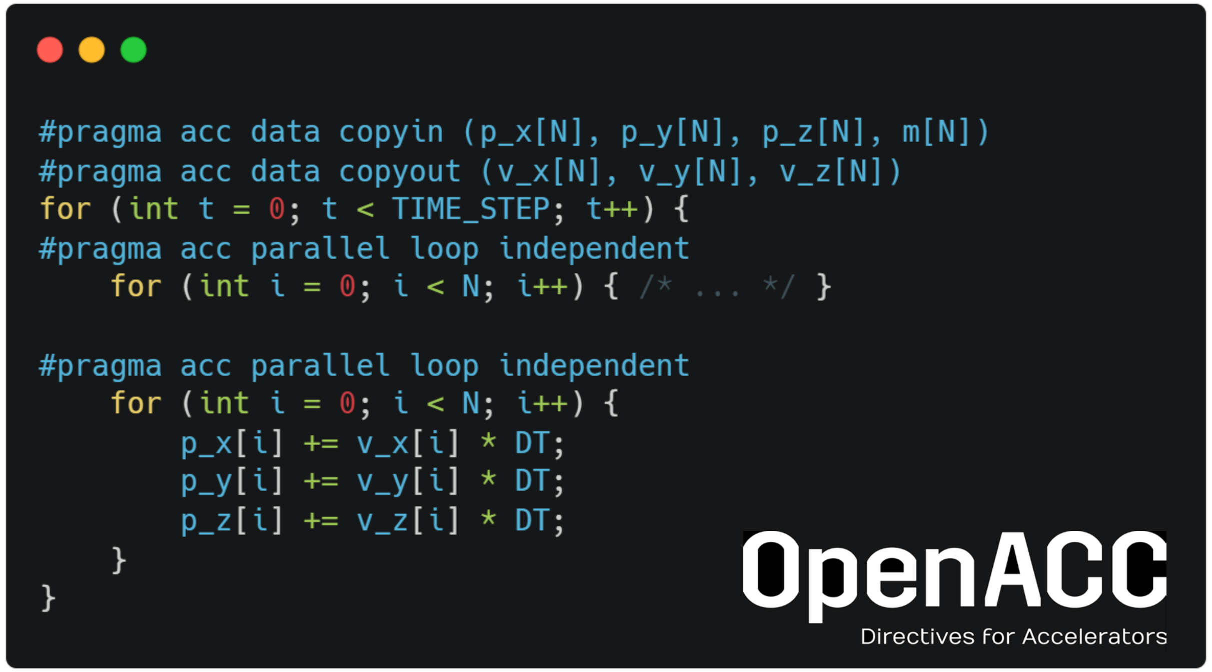
Introduction



```
__global__ void kernel(
          float *m, float3 *p, float3 *v){
    int tid = blockIdx.x *
             blockDim.x + threadIdx.x;
    int offset = tid * THREAD_SIZE;
    for (int j = 0; j < THREAD_SIZE; j++) {
        int i = offset + j;
        p[i].x += v[i].x * DT;
        p[i].y += v[i].y * DT;
        p[i].z += v[i].z * DT;
    }
}
```

- Utilizing accelerators poses additional programming cost
  - Through primitives (CUDA, OpenCL) or *Abstract Models* (DSL, Directive)
  - The latter introduces less engineering efforts w/ limited interfaces ⟶ *Less Efficiency*

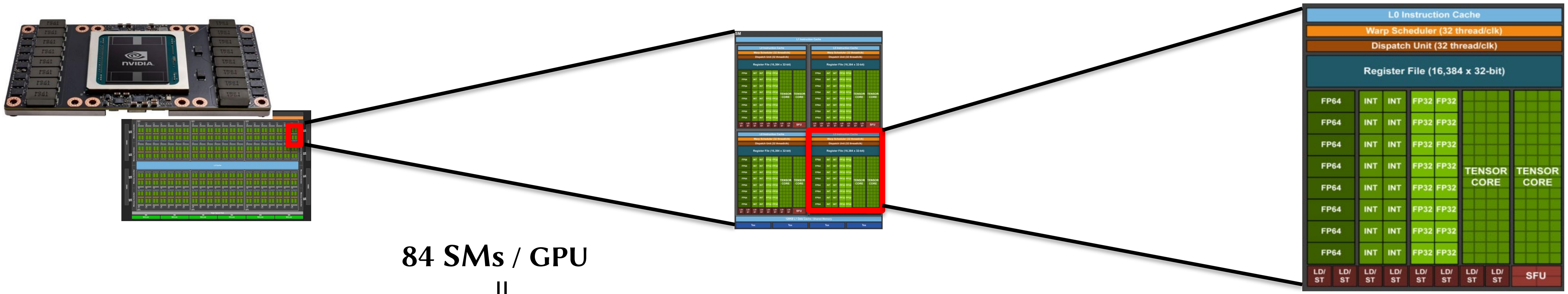
OpenACC



```
#pragma acc data copyin (p_x[N], p_y[N], p_z[N], m[N])
#pragma acc data copyout (v_x[N], v_y[N], v_z[N])
for (int t = 0; t < TIME_STEP; t++) {
#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) { /* ... */ }

#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) {
        p_x[i] += v_x[i] * DT;
        p_y[i] += v_y[i] * DT;
        p_z[i] += v_z[i] * DT;
    }
}
```

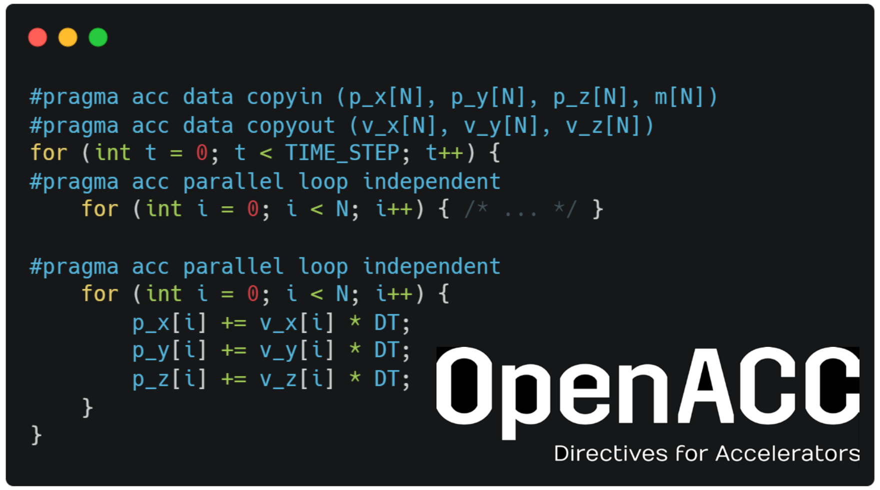
- OpenACC offers compiler directives to program accelerators in existing languages
- Without introducing vendor-specific languages such as CUDA, users are allowed to parallelize their code and rely on the compiler for generating device-specific application code
- *No access to the lower-level operation*

The NVIDIA GPU Architecture


L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU

84 SMs / GPU
||
Thread Block

~16 Warps / SM

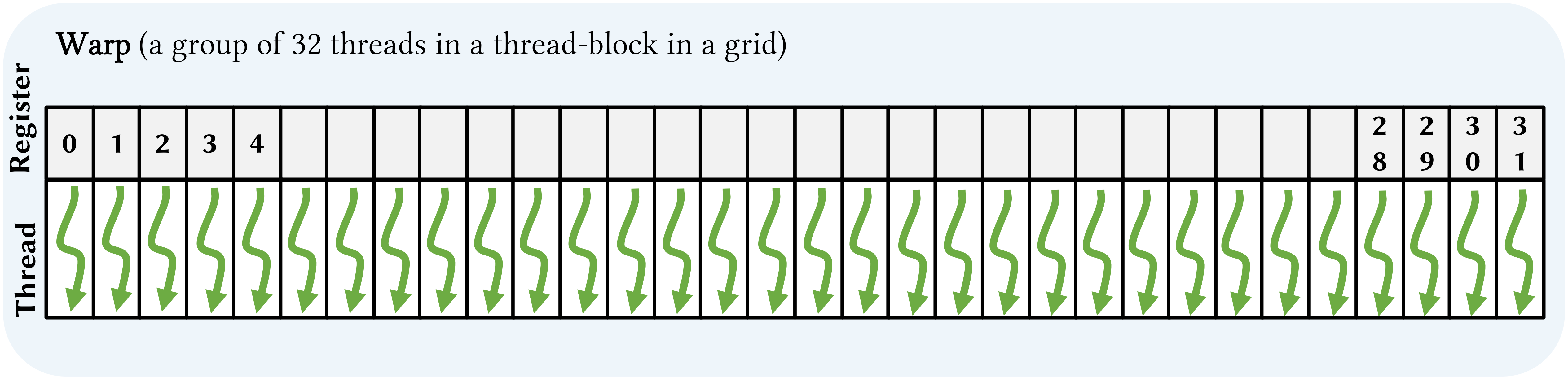


```
#pragma acc data copyin (p_x[N], p_y[N], p_z[N], m[N])
#pragma acc data copyout (v_x[N], v_y[N], v_z[N])
for (int t = 0; t < TIME_STEP; t++) {
#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) { /* ... */ }

#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) {
        p_x[i] += v_x[i] * DT;
        p_y[i] += v_y[i] * DT;
        p_z[i] += v_z[i] * DT;
    }
}
```

Grid > Thread Block > Warp > Thread

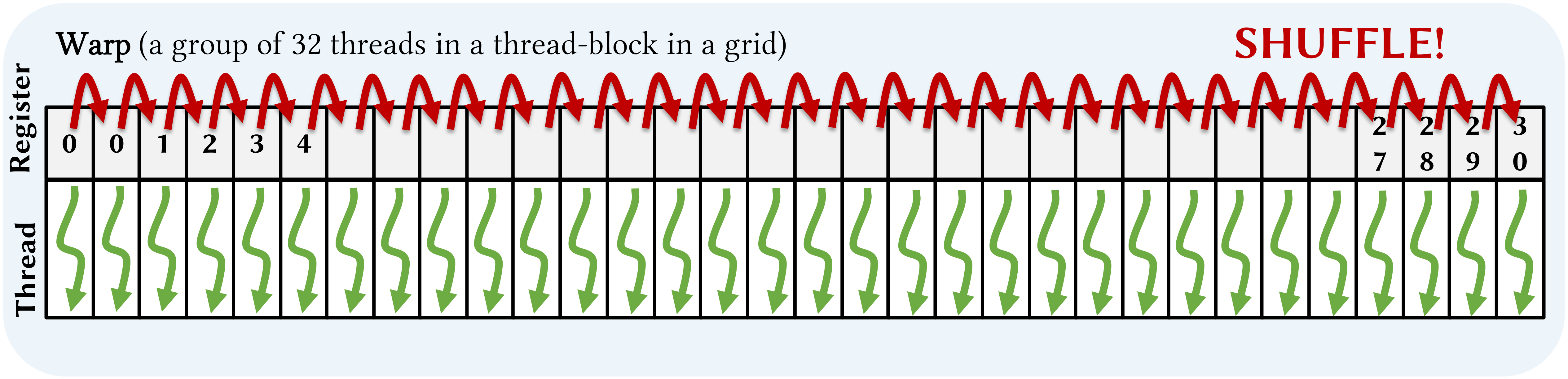
- OpenACC splits loops into thread blocks, of which the grid consists
- Each thread block can be from at most *sequential* 1024 threads.
- The unit of 32 threads is called "**Warp**"
- Threads in the same warp work together, allowing cooperation

Shuffle Operation


Warp (a group of 32 threads in a thread-block in a grid)
Register
Thread
0
1
2
3
4
28
29
30
31

- The shuffle operation is a communication means among threads in a warp

Shuffle Operation



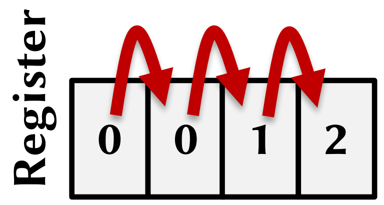
- The shuffle operation is a communication means among threads in a warp
  - ***Significant performance benefits in a lot of literature.*** Limited to CUDA or lower level code

Difficulty of Shuffle

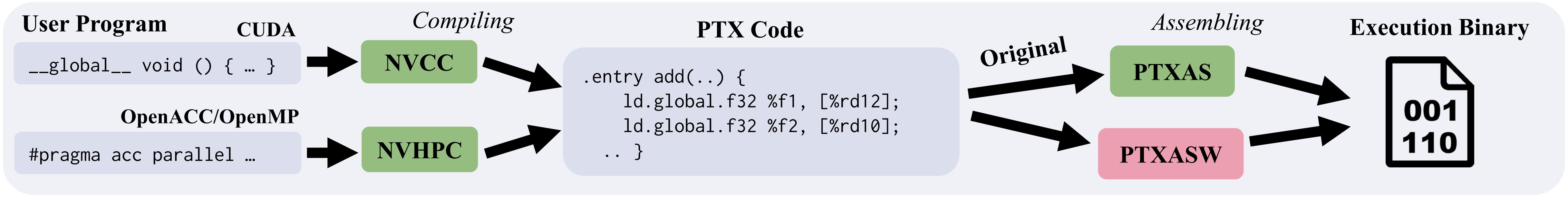
- The shuffle operation is not low cost
  - Comparable to shared memory reads

name	Shuffle (up)	SM Read	L1 Hit
Kepler	24	26	35
Maxwell	33	23	82
Pascal	33	24	82
Volta	22	19	28

- Deeply involved in the algorithm design
  - Non-trivial modification in the fundamental part of codes
- Otherwise, it is required to support corner cases
  - Edge threads, or insufficient number of threads (< 32)

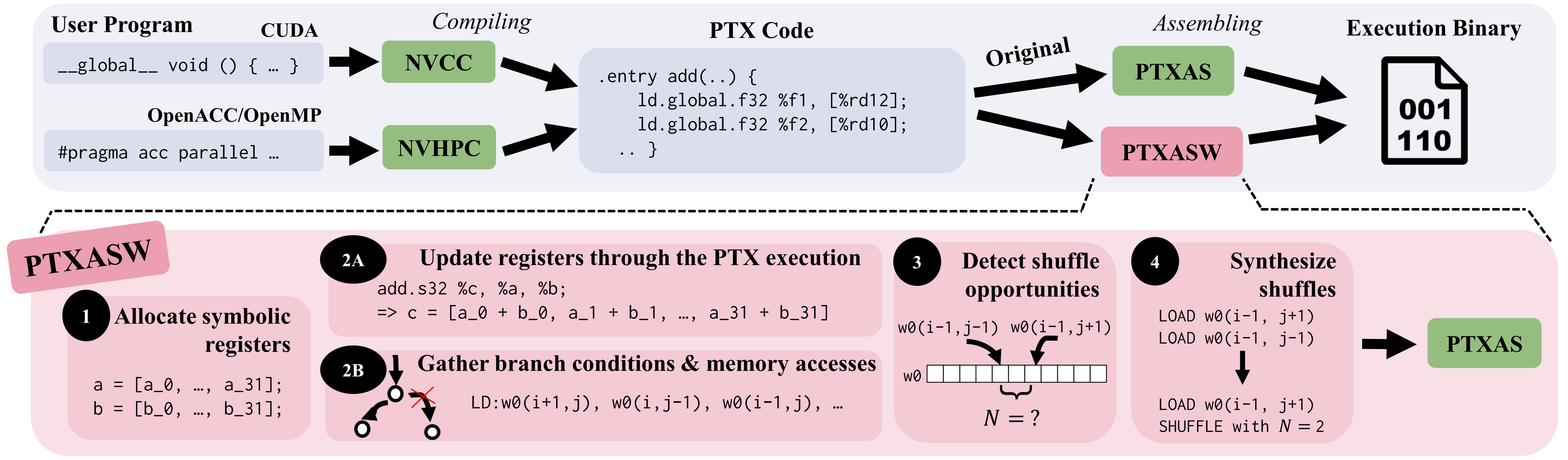


Proposal: PTXAS-Wrapper



- **Our work automates the shuffle operation with whatever applications**
  - Introduce a wrapper/optimizer for NVIDIA PTX (Assembly used for NVIDIA GPUs)
    - Supporting code generated by NVIDIA HPC Compiler (C/C++/Fortran w/ OpenACC/OpenMP) & CUDA Compiler

Proposal: PTXAS-Wrapper



- Implement *a PTX emulator* for the shuffle opportunity detection
- *Synthesize shuffle* as register caches to avoid unnecessary memory loads

○ Experiments on four generations of GPUs

- Performance improvement up to 132% on Maxwell; Analyze the use case of shuffle for each GPU

PTX Emulation (Instruction Encoding)

- First, PTXASW recognizes variable declarations and ***prepares a symbolic bitvector for each register***

```
.reg .16 a; →  a = [ a_0 , a_1 , .. , a_14 , a_15 ];
```

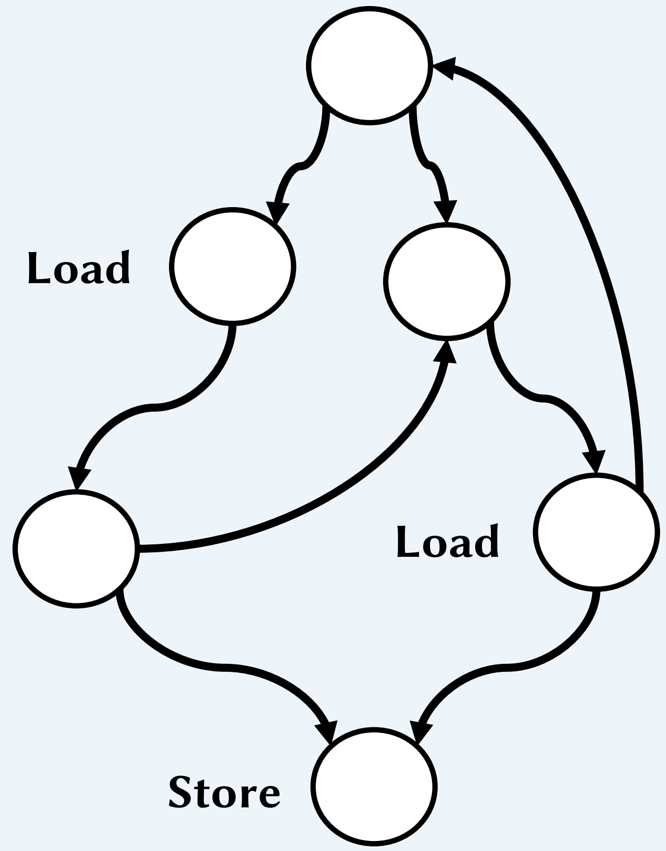
- We ***encode each PTX instruction as the computation over vectors***

```
            add.u16 %c, %a, %b; // dst: %c; src: %a, %b

                              ↓

c = a + b
  = [a_0 + b_0, a_1 + b_1, .., a_14 + b_14, a_15 + b_15];
```

PTX Emulation (Branching)

- As inputs and several parameters are unknown at compilation, unsolvable values of predicates are often observed leading to undetermined execution flows where computation is boundless

- We continue each branching while duplicating the register environment for succeeding flows. All the flows finish at the re-entry to iterative blocks or the end of instructions



PTX Emulation (Branching)

- At the entry to the iterative code block, we wrap each iterator of the block with uninterpreted functions
  - This produces incomparable values
  - Therefore, we clip the initial values out and add them to registers containing uninterpreted functions at the block entry for better accuracy in the case of incremental iterators to be found by induction variable recognition

```
i = init + loop(0, N)
```

- Accumulate the symbolic expressions in predicates used at the prior divergence
  - If the destination of a new branch can be determined by a SMT solver, unrealizable paths are pruned

- Also, memorization is performed to avoid redundant emulation according to the register environment

PTX Emulation (Memory Analysis)

```
!$acc  kernels loop independent gang(65535)
!$acc& present(w0(1:nx,1:ny), w1(1:nx,1:ny))
do j = 2, ny-1
  !$acc loop independent vector(512)
  do i = 2, nx-1
    w1(i, j) = c0 *  w0(i,   j  ) +                &
               c1 * (w0(i-1, j  ) + w0(i  , j-1)  + &
                     w0(i+1, j  ) + w0(i  , j+1)) + &
               c2 * (w0(i-1, j-1) + w0(i-1, j+1)  + &
                     w0(i+1, j-1) + w0(i+1, j+1))
  enddo
enddo
```

Extract Trace

```
LD: 0xc + (load(param2) + ((((0x1 + %ctaid.x) * load(param6) // w0(i-1, j+1)
                  + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 14)) + loop(0, 53)) << 0x2)

LD: 0xc + (load(param2) + (((load(param6) * (0x3 + %ctaid.x) // w0(i+1, j+1)
                  + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 13)) + loop(0, 52)) << 0x2)

LD: 0x4 + (load(param2) + ((((0x1 + %ctaid.x) * load(param6) // w0(i-1, j-1)
                  + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 14)) + loop(0, 53)) << 0x2)

LD: 0x4 + (load(param2) + (((load(param6) * (0x3 + %ctaid.x) // w0(i+1, j-1)
                  + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 13)) + loop(0, 52)) << 0x2)

LD: 0xc + (load(param2) + (((%tid.x + %ctaid.y << 0x9)        // w0(i  , j+1)
              + loop(0, 57)) + ((- load(param5)) + load(param6) * ((0x2 + %ctaid.x) + loop(0, 21)))) << 0x2)

/* LD: w0(i+1, j  ), w0(i  , j-1), w0(i-1, j  ), w0(i  , j  ) */

ST: 0x8 + (load(param3) + (((%tid.x + %ctaid.y << 0x9)        // w1(i  , j  )
              + loop(0, 57)) + ((- load(param5)) + load(param6) * ((0x2 + %ctaid.x) + loop(0, 21)))) << 0x2)
```

Shuffle Opportunity Detection

A(%tid)=

```
LD: 0xc + (load(param2) + ((((0x1 + %ctaid.x) * load(param6) // w0(i-1, j+1)
                          + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 14)) + loop(0, 53)) << 0x2)
```

B(%tid)=

```
LD: 0x4 + (load(param2) + ((((0x1 + %ctaid.x) * load(param6) // w0(i-1, j-1)
                          + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 14)) + loop(0, 53)) << 0x2)
```

- When $\mathrm{A}(\%\mathrm{tid} + N) = \mathrm{B}(\%\mathrm{tid})$ and $-31 \leq N \leq 31$ are satisfied, the load A can be utilized for B
  - In this case, the solver can find N=-2

Shuffle Synthesis

Original

```
ld.global.nc.f32 %f4, [%rd31+12];// w0(i-1, j+1)
/* ... */
ld.global.nc.f32 %f7, [%rd31+4]; // w0(i-1, j-1)
```

PTXASW

```
ld.global.nc.f32 %f4, [%rd31+12];
mov.f32 %source, %f4;
/* ... */
mov.u32 %warp_id, %tid.x; rem.u32 %warp_id, %warp_id, 32;
activemask.b32 %mask; setp.ne.s32 %incomplete, %mask, -1;
setp.lt.u32 %out_of_range, %warp_id, 2;
or.pred %pred, %incomplete, %out_of_range;
shfl.sync.up.b32 %f7, %source, 2, 0, %mask;
@%pred ld.global.nc.f32 %f7, [%rd31+4];
```

Shuffle Synthesis

Original

```
ld.global.nc.f32 %f4, [%rd31+12];// w0(i-1, j+1)
/* ... */
ld.global.nc.f32 %f7, [%rd31+4]; // w0(i-1, j-1)
```

PTXASW

```
ld.global.nc.f32 %f4, [%rd31+12];
mov.f32 %source, %f4;
/* ... */
mov.u32 %warp_id, %tid.x; rem.u32 %warp_id, %warp_id, 32;
activemask.b32 %mask; setp.ne.s32 %incomplete, %mask, -1;
setp.lt.u32 %out_of_range, %warp_id, 2;
or.pred %pred, %incomplete, %out_of_range;
shfl.sync.up.b32 %f7, %source, 2, 0, %mask;
@%pred ld.global.nc.f32 %f7, [%rd31+4];
```

← Check the completeness of the warp

Shuffle Synthesis

Original

```
ld.global.nc.f32 %f4, [%rd31+12];// w0(i-1, j+1)
/* ... */
ld.global.nc.f32 %f7, [%rd31+4]; // w0(i-1, j-1)
```

PTXASW

```
ld.global.nc.f32 %f4, [%rd31+12];
mov.f32 %source, %f4;
/* ... */
mov.u32 %warp_id, %tid.x; rem.u32 %warp_id, %warp_id, 32;
activemask.b32 %mask; setp.ne.s32 %incomplete, %mask, -1;
setp.lt.u32 %out_of_range, %warp_id, 2;
or.pred %pred, %incomplete, %out_of_range;
shfl.sync.up.b32 %f7, %source, 2, 0, %mask;
@%pred ld.global.nc.f32 %f7, [%rd31+4];
```

Check the case in which the source does not exist

Shuffle Synthesis

```
ld.global.nc.f32 %f4, [%rd31+12];// w0(i-1, j+1)  Original
/* ... */
ld.global.nc.f32 %f7, [%rd31+4]; // w0(i-1, j-1)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ld.global.nc.f32 %f4, [%rd31+12];                   PTXASW
mov.f32 %source, %f4;
/* ... */
mov.u32 %warp_id, %tid.x; rem.u32 %warp_id, %warp_id, 32;
activemask.b32 %mask; setp.ne.s32 %incomplete, %mask, -1;
setp.lt.u32 %out_of_range, %warp_id, 2;
or.pred %pred, %incomplete, %out_of_range;
shfl.sync.up.b32 %f7, %source, 2, 0, %mask;
@%pred ld.global.nc.f32 %f7, [%rd31+4];
```

Execute the shuffle anyway

Shuffle Synthesis

Original

```
ld.global.nc.f32 %f4, [%rd31+12];// w0(i-1, j+1)
/* ... */
ld.global.nc.f32 %f7, [%rd31+4]; // w0(i-1, j-1)
```

PTXASW

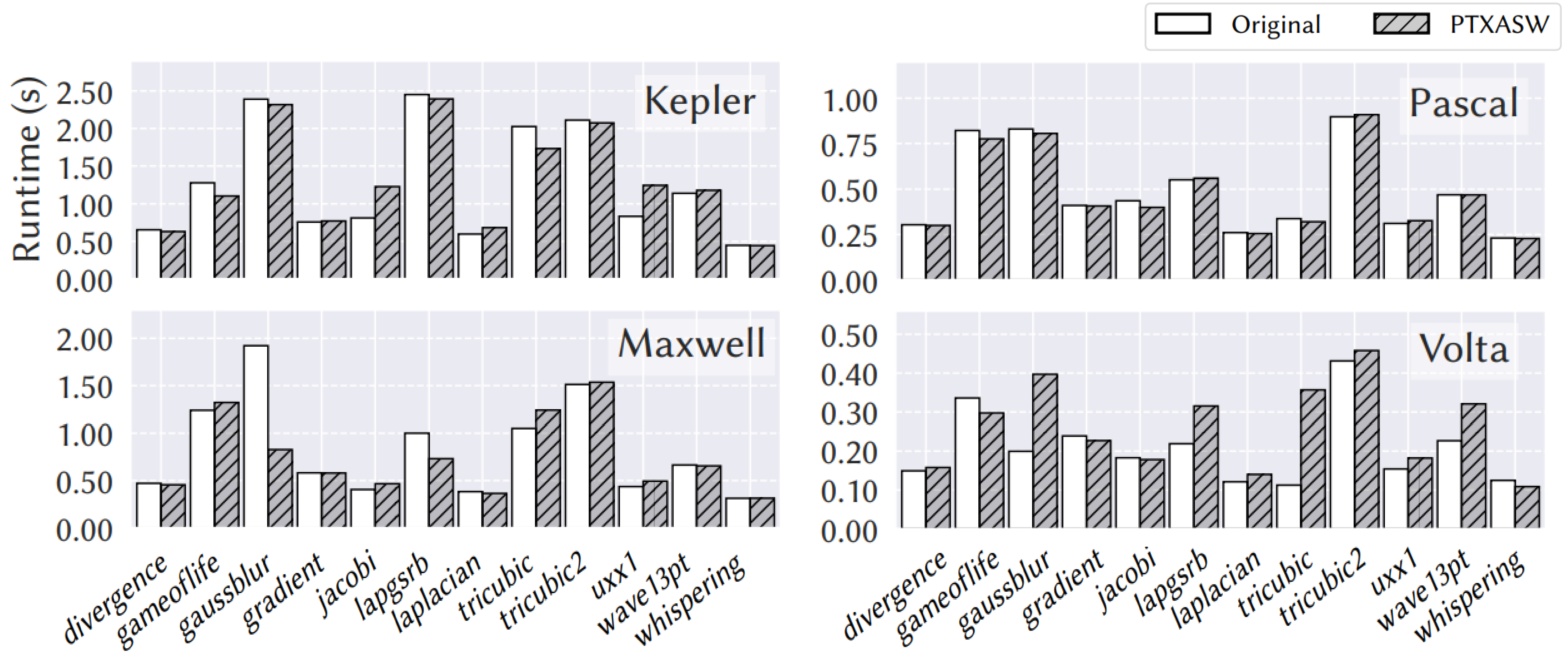
```
ld.global.nc.f32 %f4, [%rd31+12];
mov.f32 %source, %f4;
/* ... */
mov.u32 %warp_id, %tid.x; rem.u32 %warp_id, %warp_id, 32;
activemask.b32 %mask; setp.ne.s32 %incomplete, %mask, -1;
setp.lt.u32 %out_of_range, %warp_id, 2;
or.pred %pred, %incomplete, %out_of_range;
shfl.sync.up.b32 %f7, %source, 2, 0, %mask;
@%pred ld.global.nc.f32 %f7, [%rd31+4];
```

The corner cases perform original loads

Methodology

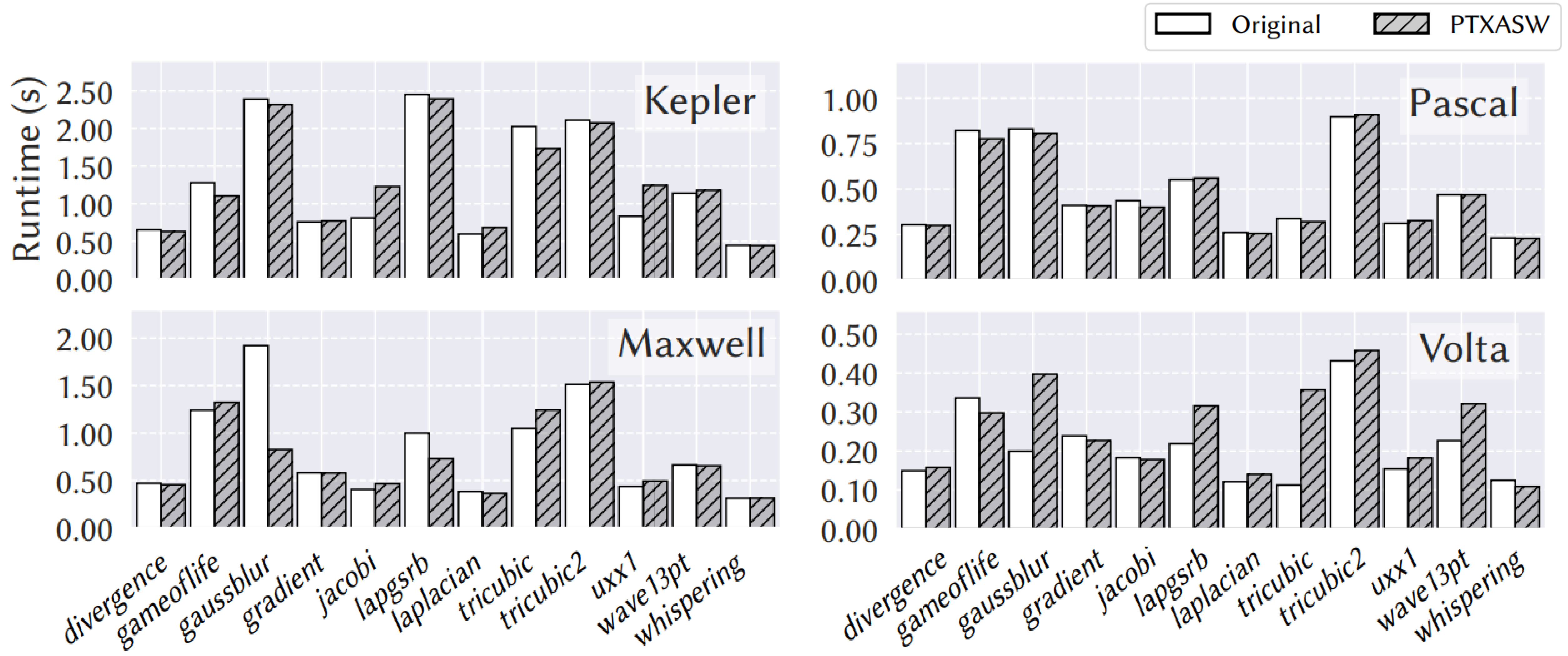
- We build PTXAS-Wrapper (**PTXASW**) using Rosette, a symbolic-evaluation system upon the Racket language
  - Equipped with a PTX parser
  - Runs the emulation of the parsed code while expressing runtime parameters as symbolic bitvectors provided by Rosette
- We evaluate our shuffle mechanism with NVHPC-compiler-generated PTX (NVHPC ver. 22.3; CUDA 11.6)
  - Fully automated and no user intervention
- We use the KernelGen benchmark suite for OpenACC
- On four generations of GPUs, the evaluation is performed: Kepler, Maxwell, Pascal, Volta

Evaluation



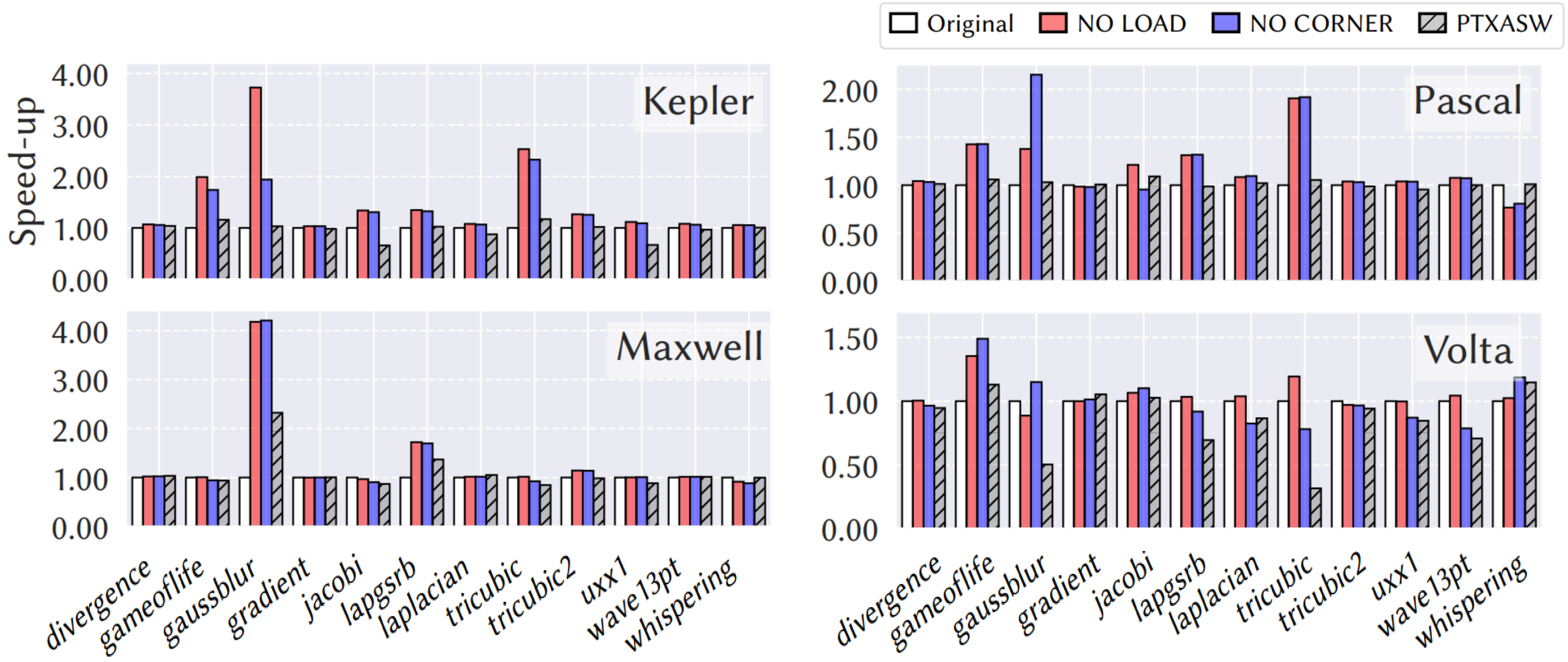
- The performance improvement is confirmed with 7/6/9/4 benchmarks showing up to 16.9%/132.3%/9.1%/14.7% on Kepler/Maxwell/Pascal/Volta

Evaluation



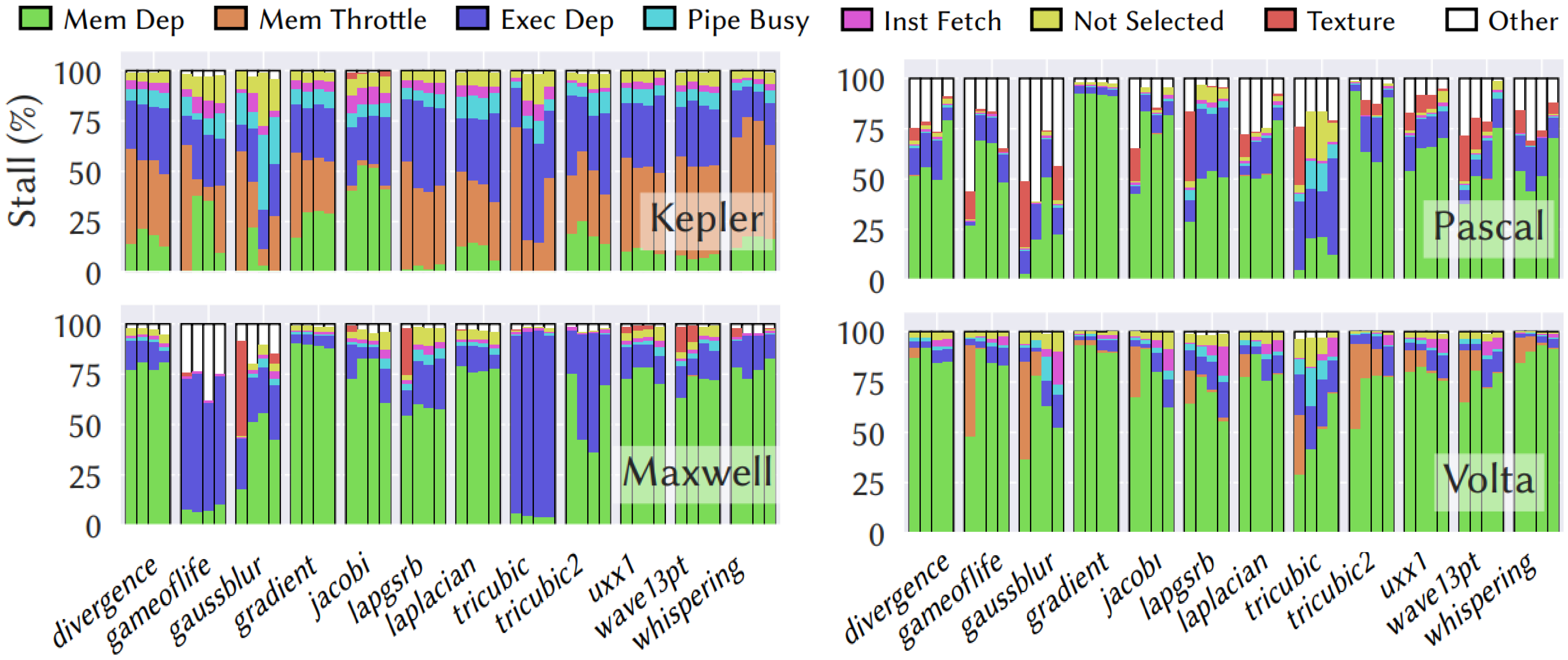
- The average improvement is -3.3%/10.9%/1.8%/-15.2% on Kepler/Maxwell/Pascal/Volta

Evaluation



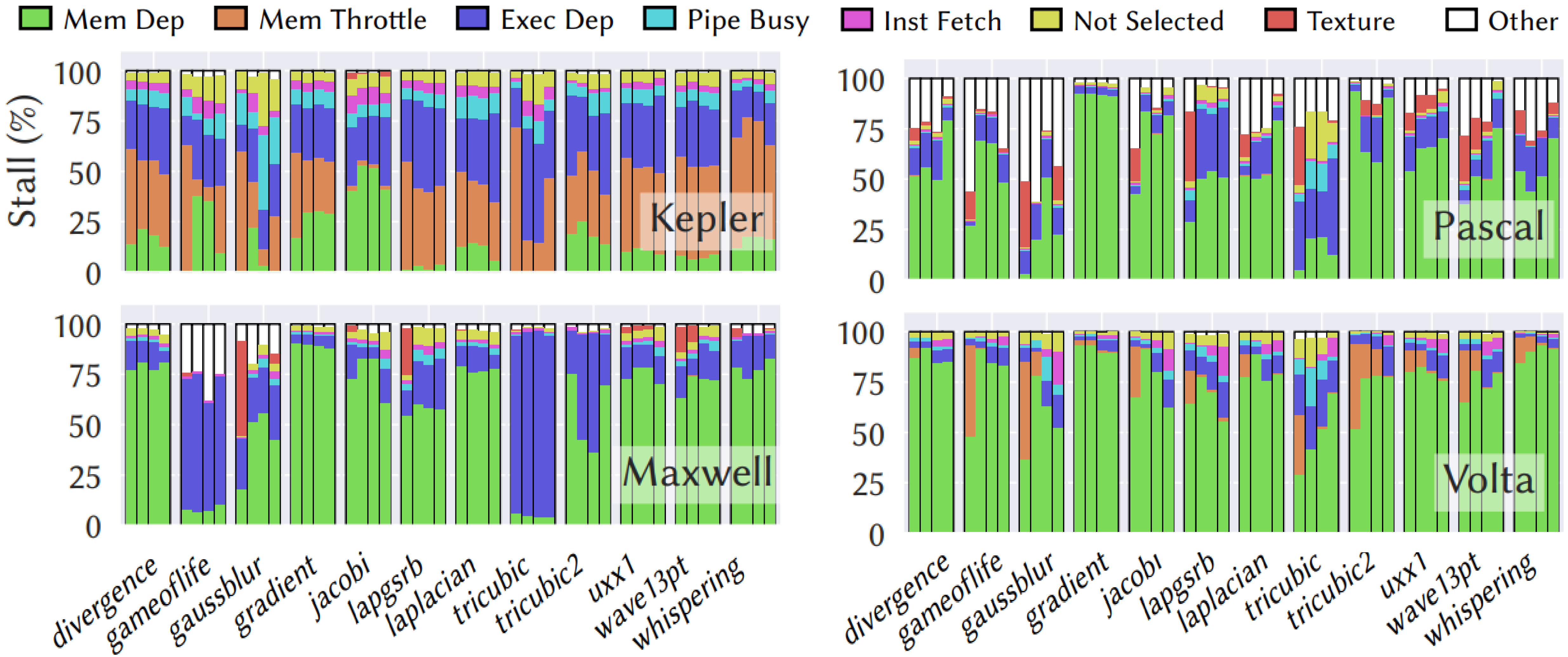
- Speedup graph with NO LOAD (eliminating loads which are covered by shuffle without having shuffles) and NO CORNER (which only executes shuffles without supporting corner cases)

Analysis



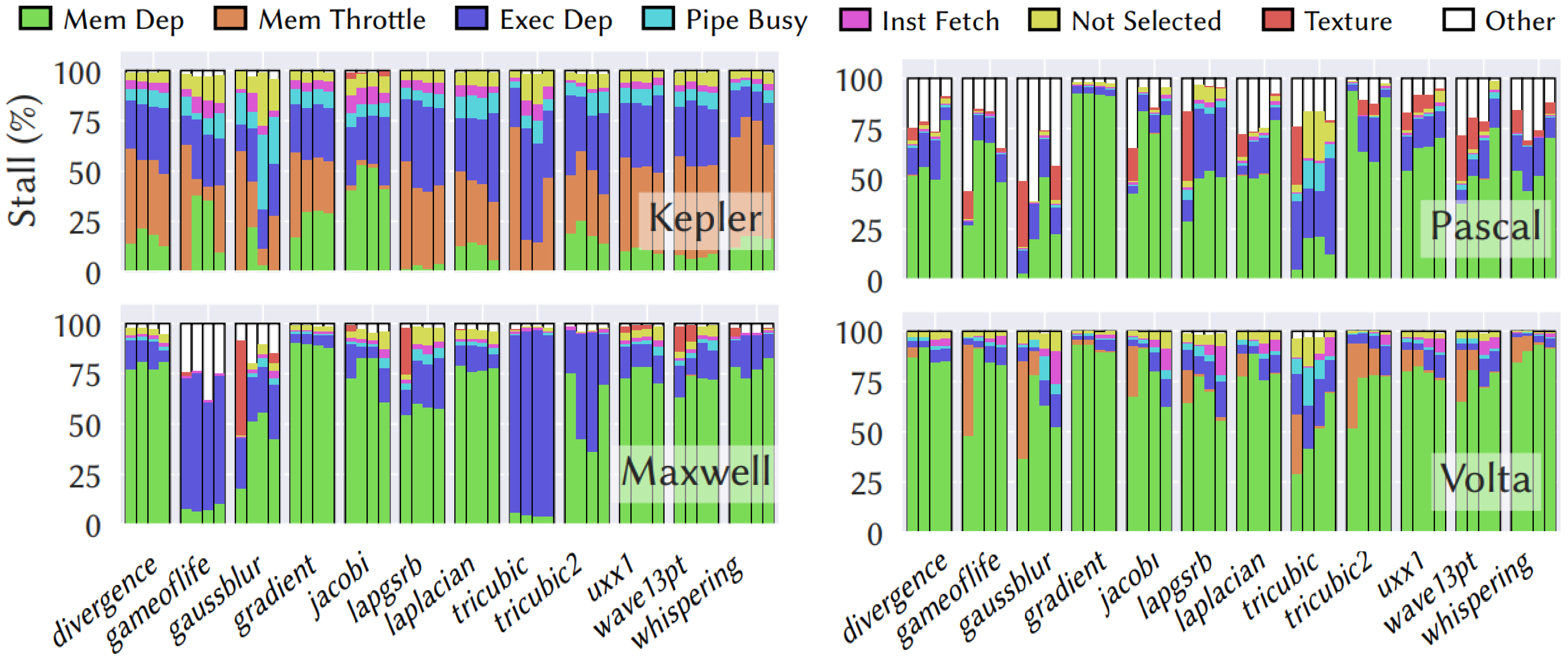
- Speedup graph with NO LOAD (eliminating loads which are covered by shuffle without having shuffles) and NO CORNER (which only executes shuffles without supporting corner cases)

Analysis



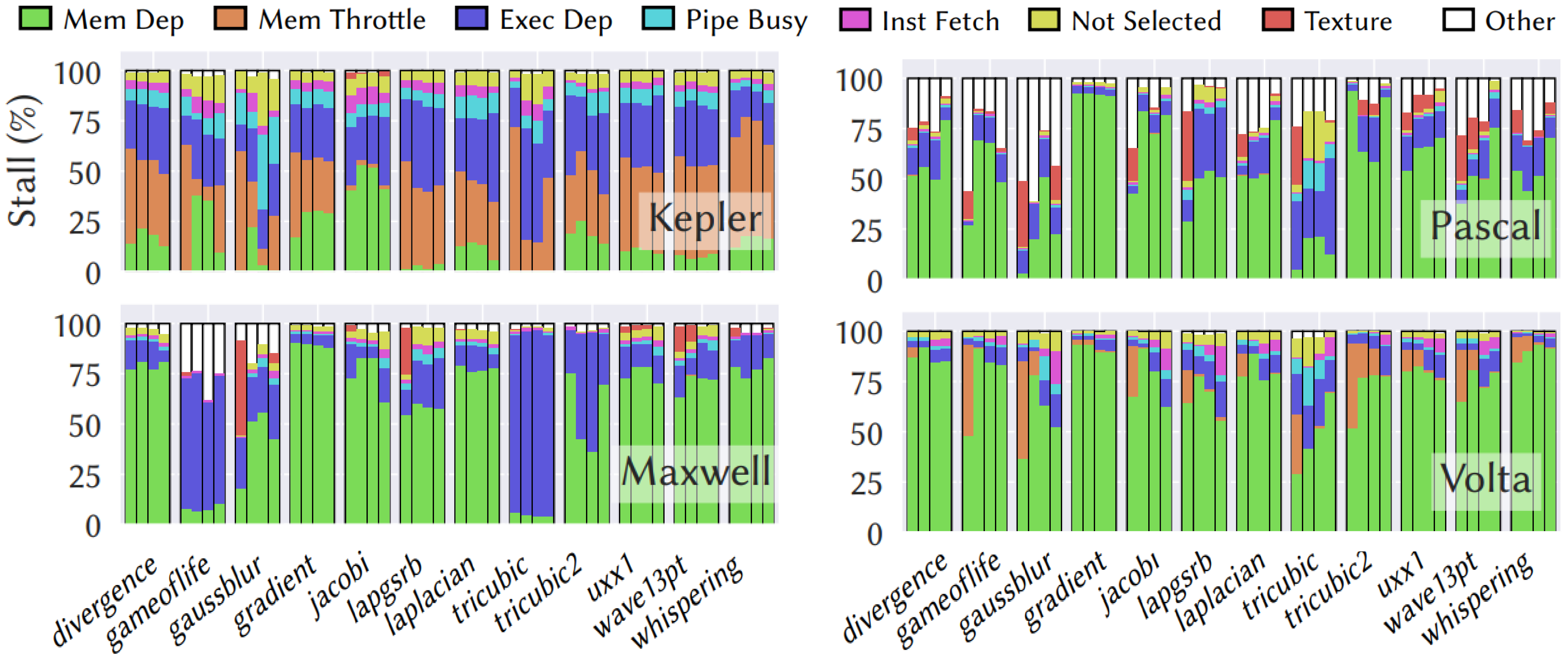
- Kepler suffers memory throttle and additional computation with corner cases
- On Maxwell/Pascal, PTXASW successfully removes the latency of texture memory

Analysis



- Memory dependency is the good indicator of memory utilization (No shuffle opportunity for general register cache)
- Execution dependency could provide opportunity for shuffle to speed up calculation

Analysis



- Volta has apparent latency with the penalty of non-aligned computation and needs modification of algorithm
- Detailed analysis is found in paper

Conclusion

- We presented **PTXASW**, an automated shuffle synthesizer for the NVIDIA PTX code
  - Equipped with a PTX paper, emulator and generator
  - Automatic detection of shuffle opportunity among global memory loads
- In evaluation, we tested our shuffle synthesis with OpenACC benchmarks
  - Achieved better throughputs on Maxwell and Pascal, which have high L1 cache latencies.
- We provided the latency breakdown and analyzed the bottleneck of shuffle execution
  - The guideline of the shuffle operation is given for each GPU

BACKUP SLIDES

EPEEC

PRODUCTIVITY AT EXASCALE

PTX

```
__global__ void add(float *c, float *a, float *b, int *f)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (f[i]) c[i] = a[i] + b[i];
}
```

Listing 1: Addition kernel in CUDA

- CUDA/OpenACC is *brought to execution on GPUs through **NVIDIA PTX***
- **PTX code is sequential**
  - But *duplicated to be run over SMs in parallel*
  - Thread-specific variables (`%tid, %ctaid`) are set accordingly to its execution

```
.visible .entry add(.param .u64 c, .param .u64 a,
                    .param .u64 b, .param .u64 f) {
/* Variable Declarations */
.reg .pred %p<2>; .reg .f32 %f<4>;
.reg .b32 %r<6>;  .reg .b64 %rd<15>;
/* PTX Statements */
ld.param.u64 %rd1, [c];     ld.param.u64 %rd2, [a];
ld.param.u64 %rd3, [b];     ld.param.u64 %rd4, [f];
cvta.to.global.u64 %rd5, %rd4;
mov.u32 %r2, %ntid.x;       mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;        mad.lo.s32 %r1, %r3, %r2, %r4;
mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
// if (!f[i]) goto $LABEL_EXIT;
ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
@%p1 bra $LABEL_EXIT;
// %f3 = a[i] + b[i]
cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
add.f32 %f3, %f2, %f1;
// c[i] = %f3
cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
st.global.f32 [%rd14], %f3;
$LABEL_EXIT: ret;
}
```

Listing 2: Addition kernel in PTX (simplified)

PTX

```
__global__ void add(float *c, float *a, float *b, int *f)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (f[i]) c[i] = a[i] + b[i];
}
```

Listing 1: Addition kernel in CUDA

- CUDA/OpenACC is *brought to execution on GPUs through **NVIDIA PTX***

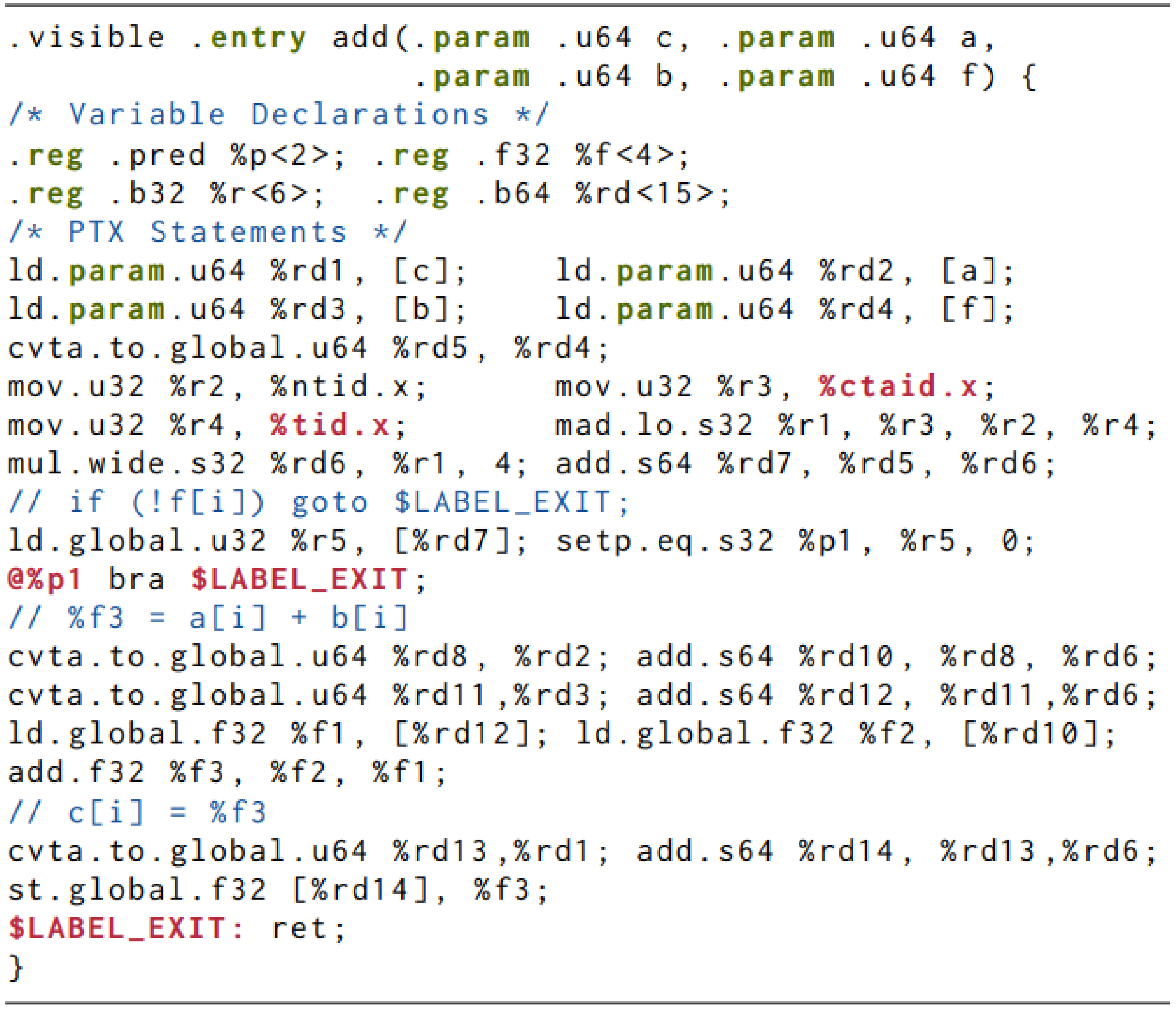
- **PTX code is sequential**
  - But *duplicated to be run over SMs in parallel*
  - Thread-specific variables (`%tid`, `%ctaid`) are set accordingly to its execution

```
.visible .entry add(.param .u64 c, .param .u64 a,
                    .param .u64 b, .param .u64 f) {
/* Variable Declarations */
.reg .pred %p<2>; .reg .f32 %f<4>;
.reg .b32 %r<6>;  .reg .b64 %rd<15>;
/* PTX Statements */
ld.param.u64 %rd1, [c];     ld.param.u64 %rd2, [a];
ld.param.u64 %rd3, [b];     ld.param.u64 %rd4, [f];
cvta.to.global.u64 %rd5, %rd4;
mov.u32 %r2, %ntid.x;       mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;        mad.lo.s32 %r1, %r3, %r2, %r4;
mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
// if (!f[i]) goto $LABEL_EXIT;
ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
@%p1 bra $LABEL_EXIT;
// %f3 = a[i] + b[i]
cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
add.f32 %f3, %f2, %f1;
// c[i] = %f3
cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
st.global.f32 [%rd14], %f3;
$LABEL_EXIT: ret;
}
```

Listing 2: Addition kernel in PTX (simplified)

PTX

- Variable declarations *correspond to the usage of on-chip resources*, especially registers
- The PTX instructions *take defined registers and save computed results* while some of them *perform the access to other resources* (`ld.global.u32`)



```
.visible .entry add(.param .u64 c, .param .u64 a,
                    .param .u64 b, .param .u64 f) {
/* Variable Declarations */
.reg .pred %p<2>; .reg .f32 %f<4>;
.reg .b32 %r<6>;  .reg .b64 %rd<15>;
/* PTX Statements */
ld.param.u64 %rd1, [c];    ld.param.u64 %rd2, [a];
ld.param.u64 %rd3, [b];    ld.param.u64 %rd4, [f];
cvta.to.global.u64 %rd5, %rd4;
mov.u32 %r2, %ntid.x;      mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;       mad.lo.s32 %r1, %r3, %r2, %r4;
mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
// if (!f[i]) goto $LABEL_EXIT;
ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
@%p1 bra $LABEL_EXIT;
// %f3 = a[i] + b[i]
cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
add.f32 %f3, %f2, %f1;
// c[i] = %f3
cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
st.global.f32 [%rd14], %f3;
$LABEL_EXIT: ret;
}
```

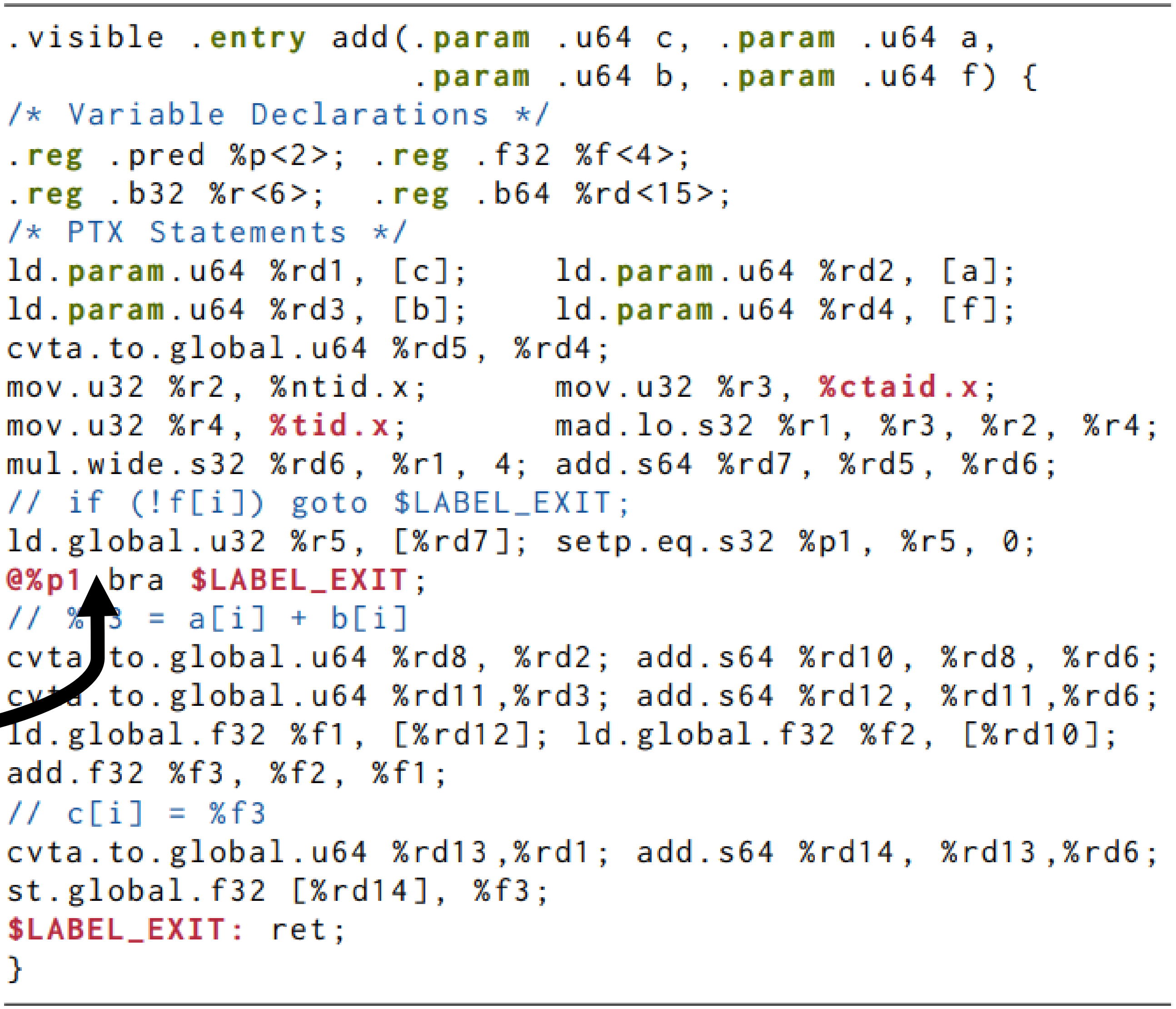
Listing 2: Addition kernel in PTX (simplified)

PTX

- <u>Variable declarations</u> *correspond to the usage of on-chip resources*, especially registers

- The PTX instructions *take defined registers and save computed results* while some of them *perform the access to other resources* (`ld.global.u32`)

global memory load



```
.visible .entry add(.param .u64 c, .param .u64 a,
                    .param .u64 b, .param .u64 f) {
/* Variable Declarations */
.reg .pred %p<2>; .reg .f32 %f<4>;
.reg .b32 %r<6>;  .reg .b64 %rd<15>;
/* PTX Statements */
ld.param.u64 %rd1, [c];    ld.param.u64 %rd2, [a];
ld.param.u64 %rd3, [b];    ld.param.u64 %rd4, [f];
cvta.to.global.u64 %rd5, %rd4;
mov.u32 %r2, %ntid.x;      mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;       mad.lo.s32 %r1, %r3, %r2, %r4;
mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
// if (!f[i]) goto $LABEL_EXIT;
ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
@%p1 bra $LABEL_EXIT;
// %f3 = a[i] + b[i]
cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
add.f32 %f3, %f2, %f1;
// c[i] = %f3
cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
st.global.f32 [%rd14], %f3;
$LABEL_EXIT: ret;
}
```

Listing 2: Addition kernel in PTX (simplified)

PTX

- The predicates (@%p1) limit the execution of the instructions stated under them, which can lead to branching based on the thread-specific values
- The labels ($LABEL_EXIT) are branch targets and allow backward jumps which can create loops

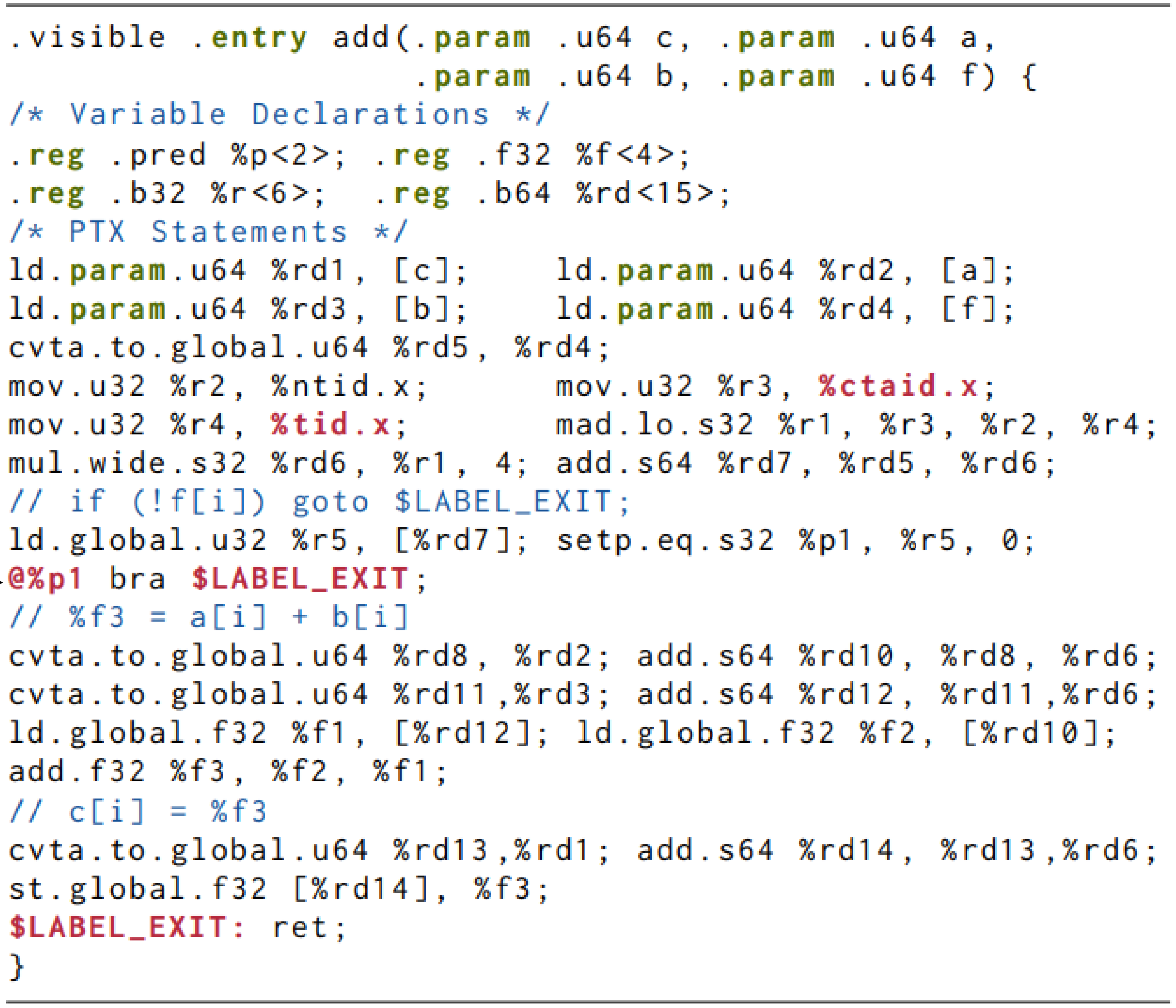
```
.visible .entry add(.param .u64 c, .param .u64 a,
                    .param .u64 b, .param .u64 f) {
/* Variable Declarations */
.reg .pred %p<2>; .reg .f32 %f<4>;
.reg .b32 %r<6>;  .reg .b64 %rd<15>;
/* PTX Statements */
ld.param.u64 %rd1, [c];     ld.param.u64 %rd2, [a];
ld.param.u64 %rd3, [b];     ld.param.u64 %rd4, [f];
cvta.to.global.u64 %rd5, %rd4;
mov.u32 %r2, %ntid.x;       mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;        mad.lo.s32 %r1, %r3, %r2, %r4;
mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
// if (!f[i]) goto $LABEL_EXIT;
ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
@%p1 bra $LABEL_EXIT;
// %f3 = a[i] + b[i]
cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
add.f32 %f3, %f2, %f1;
// c[i] = %f3
cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
st.global.f32 [%rd14], %f3;
$LABEL_EXIT: ret;
}
```

Listing 2: Addition kernel in PTX (simplified)

PTX

- The predicates (@%p1) limit the execution of the instructions stated under them, which can lead to branching based on the thread-specific values

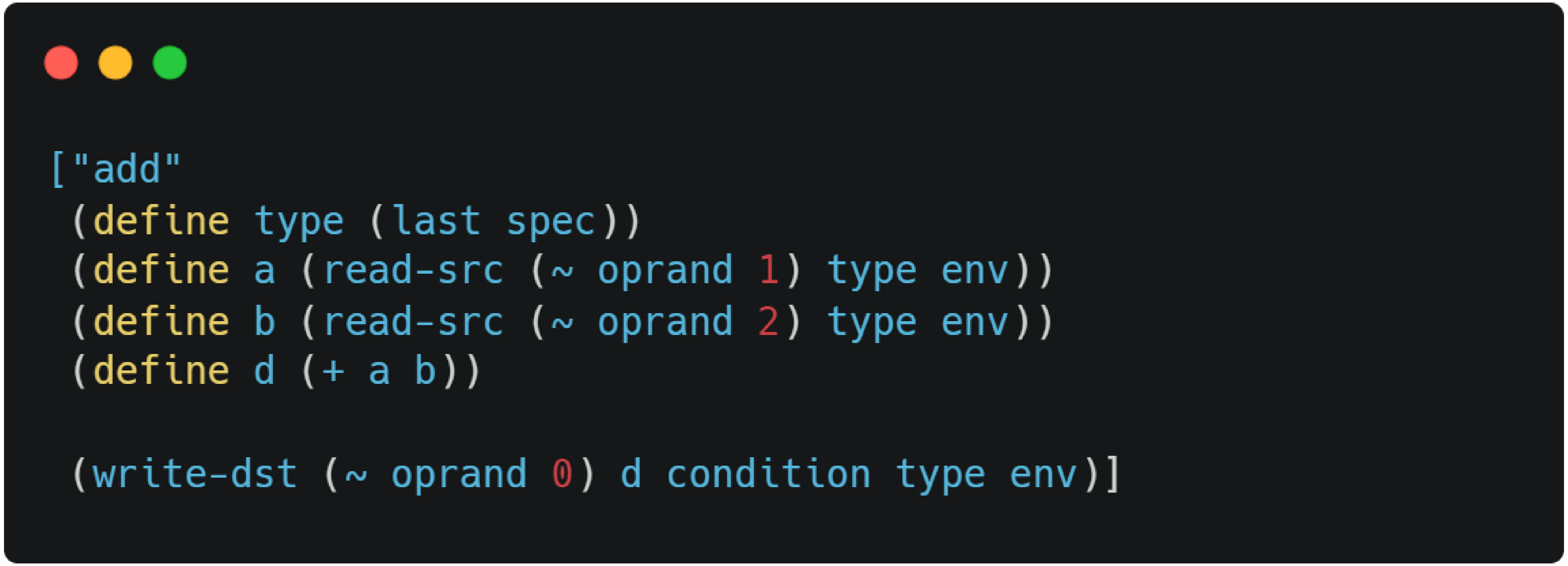
- The labels ($LABEL_EXIT) are branch targets and allow backward jumps which can create loops



```
.visible .entry add(.param .u64 c, .param .u64 a,
                    .param .u64 b, .param .u64 f) {
/* Variable Declarations */
.reg .pred %p<2>; .reg .f32 %f<4>;
.reg .b32 %r<6>;  .reg .b64 %rd<15>;
/* PTX Statements */
ld.param.u64 %rd1, [c];     ld.param.u64 %rd2, [a];
ld.param.u64 %rd3, [b];     ld.param.u64 %rd4, [f];
cvta.to.global.u64 %rd5, %rd4;
mov.u32 %r2, %ntid.x;       mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;        mad.lo.s32 %r1, %r3, %r2, %r4;
mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
// if (!f[i]) goto $LABEL_EXIT;
ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
@%p1 bra $LABEL_EXIT;
// %f3 = a[i] + b[i]
cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
add.f32 %f3, %f2, %f1;
// c[i] = %f3
cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
st.global.f32 [%rd14], %f3;
$LABEL_EXIT: ret;
}
```

Listing 2: Addition kernel in PTX (simplified)

PTX Emulation (Instruction Encoding)



```
["add"
 (define type (last spec))
 (define a (read-src (~ oprand 1) type env))
 (define b (read-src (~ oprand 2) type env))
 (define d (+ a b))

 (write-dst (~ oprand 0) d condition type env)]
```

Arithmetic calculation and bitwise operations are defined for the combinatory use of concrete and symbolic bitvectors

Methodology

name	Lang	Shuffle/Load	Delta	Analysis Time
divergence	C	1 / 6	2.00	4.281s
gameoflife	C	6 / 9	1.50	3.470s
gaussblur	C	20 / 25	2.50	7.938s
gradient	C	1 / 6	2.00	4.668s
jacobi	F	6 / 9	1.50	4.119s
lapgsrb	C	12 / 25	1.83	14.296s
laplacian	C	2 / 7	1.50	4.816s
matmul	F	0 / 8	-	13.971s
matvec	C	0 / 7	-	4.929s
sincos	F	0 / 2	-	1m41.424s
tricubic	C	48 / 67	2.00	1m39.476s
tricubic2	C	48 / 67	2.00	1m41.855s
uxx1	C	3 / 17	2.00	7.466s
vecadd	C	0 / 2	-	3.281s
wave13pt	C	4 / 14	2.50	6.967s
whispering	C	6 / 19	0.83	6.288s

Methodology

No neighboring accesses

No multiple loads sharing the same array

name	Lang	Shuffle/Load	Delta	Analysis Time
divergence	C	1 / 6	2.00	4.281s
gameoflife	C	6 / 9	1.50	3.470s
gaussblur	C	20 / 25	2.50	7.938s
gradient	C	1 / 6	2.00	4.668s
jacobi	F	6 / 9	1.50	4.119s
lapgsrb	C	12 / 25	1.83	14.296s
laplacian	C	2 / 7	1.50	4.816s
matmul	F	0 / 8	-	13.971s
matvec	C	0 / 7	-	4.929s
sincos	F	0 / 2	-	1m41.424s
tricubic	C	48 / 67	2.00	1m39.476s
tricubic2	C	48 / 67	2.00	1m41.855s
uxx1	C	3 / 17	2.00	7.466s
vecadd	C	0 / 2	-	3.281s
wave13pt	C	4 / 14	2.50	6.967s
whispering	C	6 / 19	0.83	6.288s