MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

GPUDrano: Detecting Uncoalesced Accesses in GPU Programs

Authors: Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija and Nimit Singhania

Presented by group 10: Hanlin Bi, Yuxiang Chen, Ziang Li, Qifa Wang, Zihao Ye

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

Dynamic Analysis

Evaluation

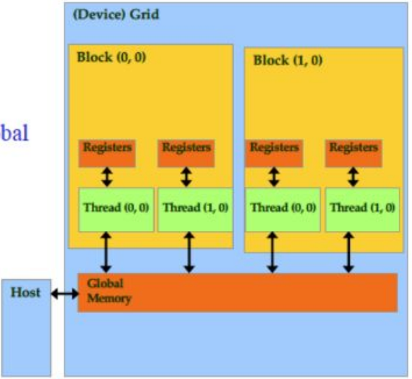
Comments

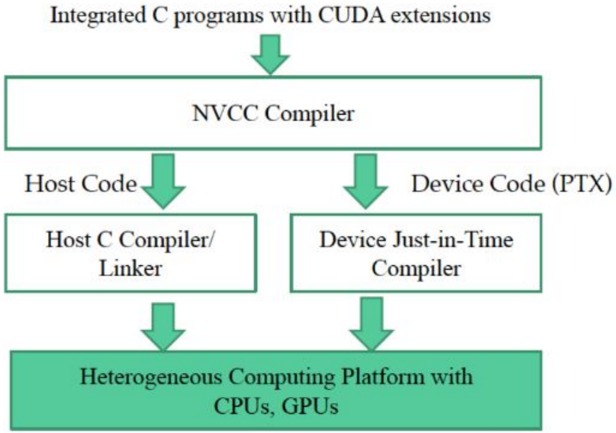
Q&A

Intro to GPU Architecture & CUDA Programming

- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory
- Host code can
  - Transfer data to/from per grid global memory

We will cover more later.





Integrated C programs with CUDA extensions

NVCC Compiler

Host Code

Device Code (PTX)

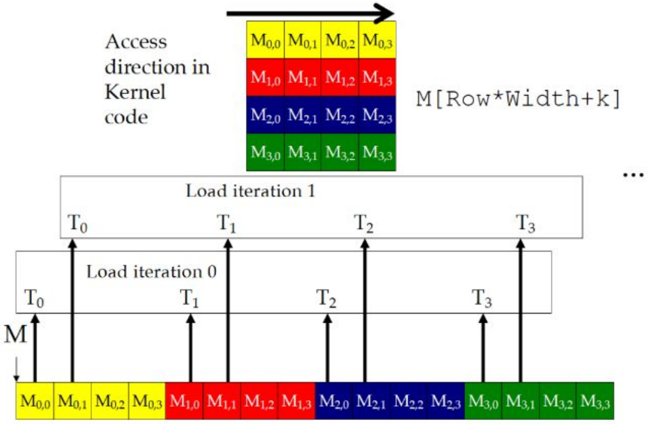
Host C Compiler/ Linker

Device Just-in-Time Compiler

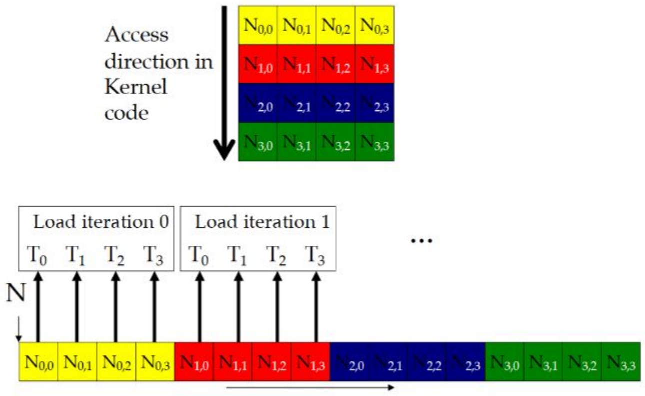
Heterogeneous Computing Platform with CPUs, GPUs

Memory Coalescing & Research Objectives

M accesses are not coalesced


Access direction in Kernel code
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M[Row*Width+k]
...
Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

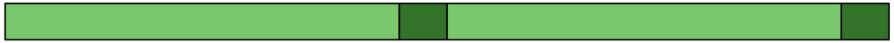
N accesses are coalesced


Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
...
N
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$ $N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$ $N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$ $N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

DRAM Burst

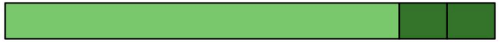
- Accessing data in different bursts (rows)
  - Need to access the array again

Timeline:



- Accessing data in the same burst (row)
  - No need to access the array again, just the multiplexer

Timeline:



- Accessing data in the same burst is faster than accessing data in different bursts

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

Dynamic Analysis

Evaluation

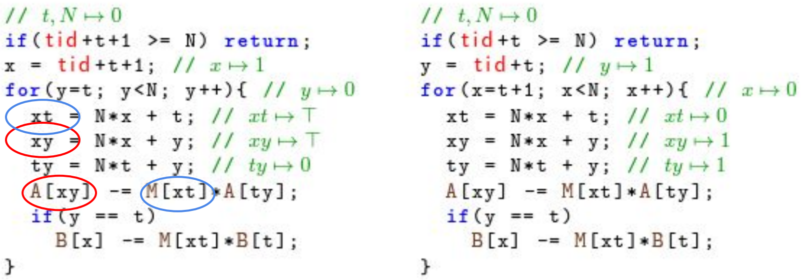
Comments

Q&A

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Formalization of Uncoalesced Access - Illustration

- Access to A[xy] on left is uncoalesced access
  - A[N(t+1)+y], A[N(t+2)+y], …, A[N(t+32)+y]
- On Right is modified, coalesced access
  - A[Nx+t], A[Nx+t+1], …, A[Nx+t+31]
- Access to M[xt] is similar
- 25% reduction in run-time when N = 1024



```
// t,N ↦ 0
if(tid+t+1 >= N) return;
x = tid+t+1; // x ↦ 1
for(y=t; y<N; y++){ // y ↦ 0
  xt = N*x + t; // xt ↦ ⊤
  xy = N*x + y; // xy ↦ ⊤
  ty = N*t + y; // ty ↦ 0
  A[xy] -= M[xt]*A[ty];
  if(y == t)
    B[x] -= M[xt]*B[t];
}
```

(a) Original Fan2 snippet

```
// t,N ↦ 0
if(tid+t >= N) return;
y = tid+t; // y ↦ 1
for(x=t+1; x<N; x++){ // x ↦ 0
  xt = N*x + t; // xt ↦ 0
  xy = N*x + y; // xy ↦ 1
  ty = N*t + y; // ty ↦ 1
  A[xy] -= M[xt]*A[ty];
  if(y == t)
    B[x] -= M[xt]*B[t];
}
```

(b) Fixed Fan2 snippet

Fig. 2: Kernel snippets from Gaussian Elimination program.

Formalization of Uncoalesced Access - Machine Model

- Representation of GPU program
  - <**T**: all threads, **V_L**: local var, **V_G**: global var, **K**: kernel(instructions)>
- A few assumptions
  - Execute in lock-steps (no longer true for all GPU)
  - Not distinguishing memory levels, considering memory as a whole (Not related)

Formalization of Uncoalesced Access - Semantics

σ: the state of a program

π: active set of threads

⊥: represent an undefined or error state (and vice versa for T)

S: A general statement in the program

AS: denotes an assignment statement within the program

l: local variable

g: global variable

[l’ := g(l)]: global array read

[g(l) := l’]: global array write

W: warp

η: Bandwidth of GPU

Formalization of Uncoalesced Access - Definition

$$\Gamma(\sigma, \pi, AS, W) = \bigcup_{t \in W \cap \pi} [\sigma(l,t).k, \sigma(l,t).k + k - 1]$$

N (σ, π, AS, W): *Number of transaction required for the access* = a/η**,** where a is the number of unique elements in Γ(σ, π, AS, W); **η** = 128 bytes, **W** = 32 threads, **Γ** = 1 as threshold for most GPU

Ranged-based uncoalesced access

Assume the index variable l is a linear function of tid, l = c*tid+c0

Range of access: $(31|kc| + k - 1)$ bytes

Plug in, N = $|kc|/4$, K >= 4 and c >= 1 will make N>=1 and thus coalesced

Alignment-based uncoalesced access

Suppose k = 4, c = 1, but c0 = 8. The addresses accessed by warp with tids [0,31] is [32, 159]. Same size, but doesn't fall into one cache block

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

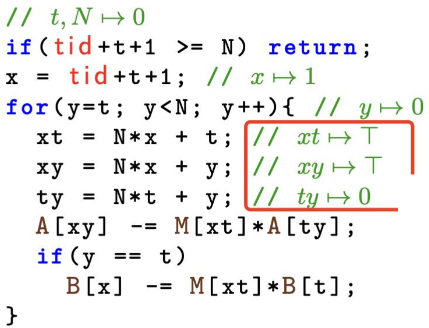
Dynamic Analysis

Evaluation

Comments

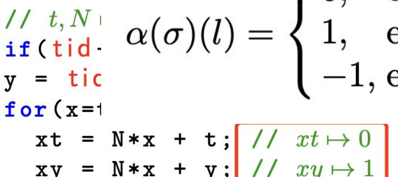
Q&A

Abstraction



```
// t,N ↦ 0
if(tid+t+1 >= N) return;
x = tid+t+1; // x ↦ 1
for(y=t; y<N; y++){ // y ↦ 0
  xt = N*x + t; // xt ↦ ⊤
  xy = N*x + y; // xy ↦ ⊤
  ty = N*t + y; // ty ↦ 0
  A[xy] -= M[xt]*A[ty];
  if(y == t)
    B[x] -= M[xt]*B[t];
}
```

(a) Original Fan2 snippet

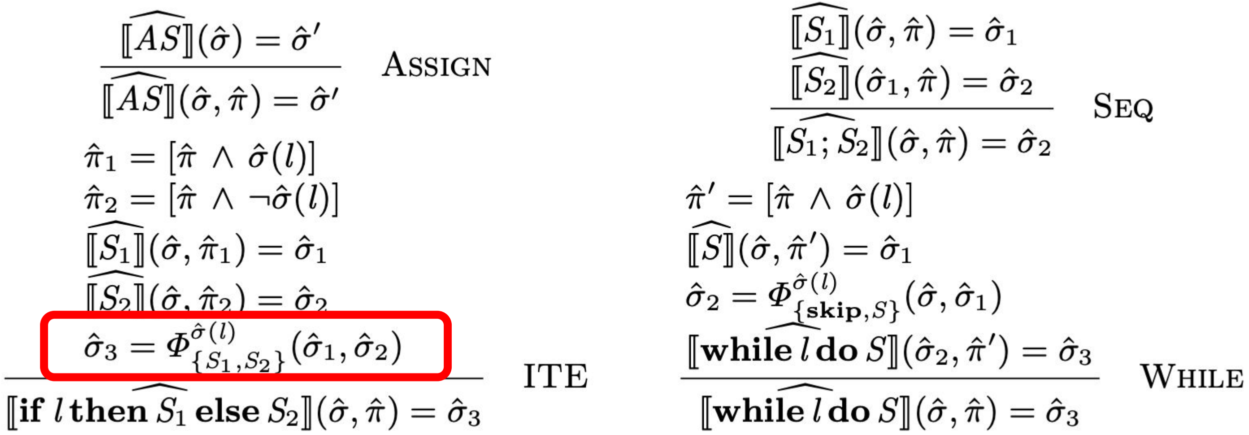


```
// t,N
if(tid
y = tic
for(x=t
  xt = N*x + t; // xt ↦ 0
  xy = N*x + y; // xy ↦ 1
```

$$\alpha(\sigma)(l) = \begin{cases} 0, & \text{exists } c_0 \text{ s.t. for all } t \in T,\ \sigma(l,t) = c_0 \\ 1, & \text{exists } c_0 \text{ s.t. for all } t \in T,\ \sigma(l,t) = \mathsf{tid}(t) + c_0 \\ -1, & \text{exists } c_0 \text{ s.t. for all } t \in T,\ \sigma(l,t) = -\mathsf{tid}(t) + c_0 \end{cases}$$

$$\alpha(\sigma)(l) \equiv \begin{cases} \mathsf{T}, & \text{for all } t \in T,\ \sigma(l,t) = \mathsf{true} \\ \mathsf{T}^-, & \text{exists } t \in T \text{ s.t. } \sigma(l,t) = \mathsf{false} \\ & \text{and for all } t' \in T \setminus t,\ \sigma(l,t') = \mathsf{true} \\ \mathsf{F}, & \text{for all } t \in T,\ \sigma(l,t) = \mathsf{false} \\ \mathsf{F}^-, & \text{exists } t \in T \text{ s.t. } \sigma(l,t) = \mathsf{true} \\ & \text{and for all } t' \in T \setminus t,\ \sigma(l,t') = \mathsf{false} \end{cases}$$

Abstract Semantic

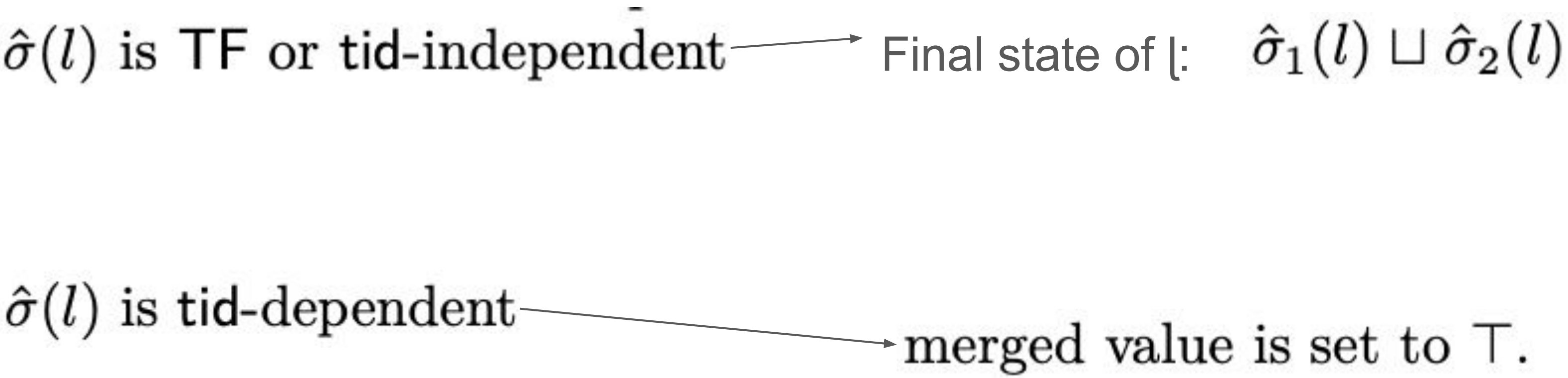


$$\frac{\widehat{[\![AS]\!]}(\hat{\sigma}) = \hat{\sigma}'}{\widehat{[\![AS]\!]}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}'}\ \textsc{Assign}
\qquad
\frac{\begin{array}{c}\widehat{[\![S_1]\!]}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_1 \\ \widehat{[\![S_2]\!]}(\hat{\sigma}_1, \hat{\pi}) = \hat{\sigma}_2\end{array}}{\widehat{[\![S_1; S_2]\!]}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_2}\ \textsc{Seq}$$

$$\frac{\begin{array}{l}\hat{\pi}_1 = [\hat{\pi} \wedge \hat{\sigma}(l)] \\ \hat{\pi}_2 = [\hat{\pi} \wedge \neg\hat{\sigma}(l)] \\ \widehat{[\![S_1]\!]}(\hat{\sigma}, \hat{\pi}_1) = \hat{\sigma}_1 \\ \widehat{[\![S_2]\!]}(\hat{\sigma}, \hat{\pi}_2) = \hat{\sigma}_2 \\ \boxed{\hat{\sigma}_3 = \Phi^{\hat{\sigma}(l)}_{\{S_1, S_2\}}(\hat{\sigma}_1, \hat{\sigma}_2)}\end{array}}{\widehat{[\![\textbf{if } l \textbf{ then } S_1 \textbf{ else } S_2]\!]}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_3}\ \text{ITE}
\qquad
\frac{\begin{array}{l}\hat{\pi}' = [\hat{\pi} \wedge \hat{\sigma}(l)] \\ \widehat{[\![S]\!]}(\hat{\sigma}, \hat{\pi}') = \hat{\sigma}_1 \\ \hat{\sigma}_2 = \Phi^{\hat{\sigma}(l)}_{\{\textbf{skip}, S\}}(\hat{\sigma}, \hat{\sigma}_1) \\ \widehat{[\![\textbf{while } l \textbf{ do } S]\!]}(\hat{\sigma}_2, \hat{\pi}') = \hat{\sigma}_3\end{array}}{\widehat{[\![\textbf{while } l \textbf{ do } S]\!]}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_3}\ \textsc{While}$$

Fig. 4: Abstract semantics for compound statements.

Merge



$\hat{\sigma}(l)$ is TF or tid-independent → Final state of l: $\hat{\sigma}_1(l) \sqcup \hat{\sigma}_2(l)$

$\hat{\sigma}(l)$ is tid-dependent → merged value is set to $\top$.

example:
Y = (tid < N) ? 10:20;
K = arr[Y]

Uncoalesced Access

$(\hat{\sigma}, \hat{\pi}, AS)$ is "uncoalesced"

AS: global array read or write
Array element size: k bytes

- $\hat{\pi} = \top$ Accessed by more than one thread

- $(\hat{\sigma}(l) = \top) \vee (\hat{\sigma}(l) \in \{1, -1\} \wedge k > 4)$

Non-linear function or element size greater than 4 bytes

Presumes default parameters

GPU parameters:

- Bandwidth: 128 B (cache line)
- Thread warp size: 32

Index function: $l \equiv c.\mathrm{tid} + c_0.$

Range address accessed: $(31|kc| + k - 1)$ bytes

Number of trx needed: $|kc|/4$

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

Dynamic Analysis

Evaluation

Comments

Q&A

Dynamic Analysis Approaches

- Utilized the lowest tid thread in each warp as a computing thread
- The computing thread collects all unique address accessed by the warp
- Operated address division into bins for corresponding cachelines
- Aggregates the information from each dynamic instance of an instruction by averaging
- If average cache line access > 1.0, the it’s not coalesced

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

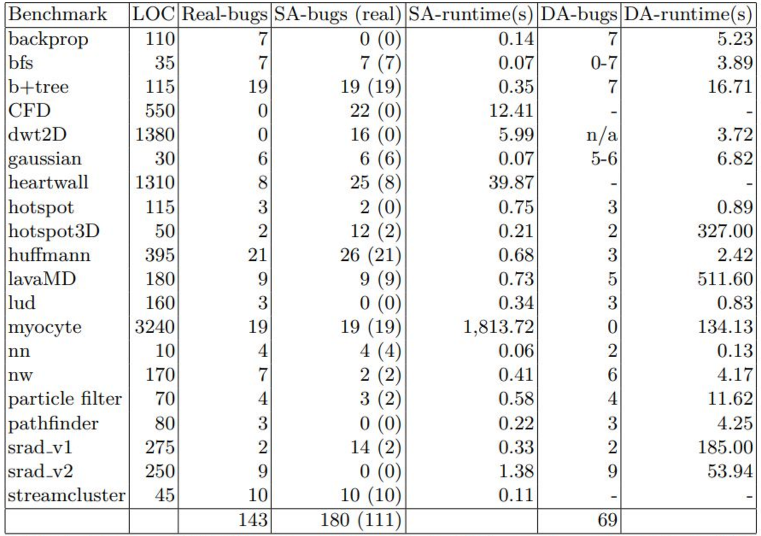
Dynamic Analysis

Evaluation

Comments

Q&A

Evaluations



Benchmark	LOC	Real-bugs	SA-bugs (real)	SA-runtime(s)	DA-bugs	DA-runtime(s)
backprop	110	7	0 (0)	0.14	7	5.23
bfs	35	7	7 (7)	0.07	0-7	3.89
b+tree	115	19	19 (19)	0.35	7	16.71
CFD	550	0	22 (0)	12.41	-	-
dwt2D	1380	0	16 (0)	5.99	n/a	3.72
gaussian	30	6	6 (6)	0.07	5-6	6.82
heartwall	1310	8	25 (8)	39.87	-	-
hotspot	115	3	2 (0)	0.75	3	0.89
hotspot3D	50	2	12 (2)	0.21	2	327.00
huffmann	395	21	26 (21)	0.68	3	2.42
lavaMD	180	9	9 (9)	0.73	5	511.60
lud	160	3	0 (0)	0.34	3	0.83
myocyte	3240	19	19 (19)	1,813.72	0	134.13
nn	10	4	4 (4)	0.06	2	0.13
nw	170	7	2 (2)	0.41	6	4.17
particle filter	70	4	3 (2)	0.58	4	11.62
pathfinder	80	3	0 (0)	0.22	3	4.25
srad_v1	275	2	14 (2)	0.33	2	185.00
srad_v2	250	9	0 (0)	1.38	9	53.94
streamcluster	45	10	10 (10)	0.11	-	-
		143	180 (111)		69	

Table 1: Results of GPUDrano's static analysis (SA) and dynamic analysis (DA) on Rodinia benchmark programs. "-" indicates the DA hit the 2-hour timeout.

Real Bugs: 143
Uncoalesced Accesses: 180 (some uncoalesced patterns are not avoidable)

Bug Detect:
Static Analysis: 111
Dynamic Analysis: 69

Scalability:
Static Analysis: fast
Dynamic Analysis: slow on long running program

Limitation

- Not a precise analysis for function calls and pointers
- Assumed the call context of the kernel is always empty

Other Related Analysis Tools

CuMAPz: runtime trace
CUDA-lite: programmer annotations (available only to affine access pattern)
Dymaxion: eschew static analysis for programmer input
CUPL: similar to GPUDrano, but no formalizations and experimental result
GKLEE: limited to underlying SMT solver and not scalable to larger kernel
PUG: same problem as GKLEE

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

Dynamic Analysis

Evaluation

Comments

Q&A

Group Comments - Positive

- Formal definitions of both the analysis and the bugs
  - Easy integration with existing performance models
- Scalable framework for tracing memory access patterns
  - Useful across architectures
    - Different DRAM burst size, SIMD processing unit, multi-bank memory, etc

Group Comments - Limitations and Improvements

- A more subdivided definition
  - Not all “bugs” should or even can be fixed
  - We focus on gather/scatter pattern and column-major access on row-major data
- Integration with roofline analysis
  - For a memory bounded program, extend the analysis:
    - Access pattern flaws => bandwidth improvements => performance gains
- Newer GPU architectures no longer guarantee warp-synchronicity
  - AKA independent thread scheduling inside a warp
  - Fails the dynamic analysis, can be fixed by adding sync when inserting bookkeeping
  - Static analysis may not match real-world hardware behavior
  - The paper was written in 2017, the exact year Volta was released

Road Map

Introduction to GPU and CUDA

Formalization

Static Analysis

Dynamic Analysis

Evaluation

Comments

Q&A

Thank you!