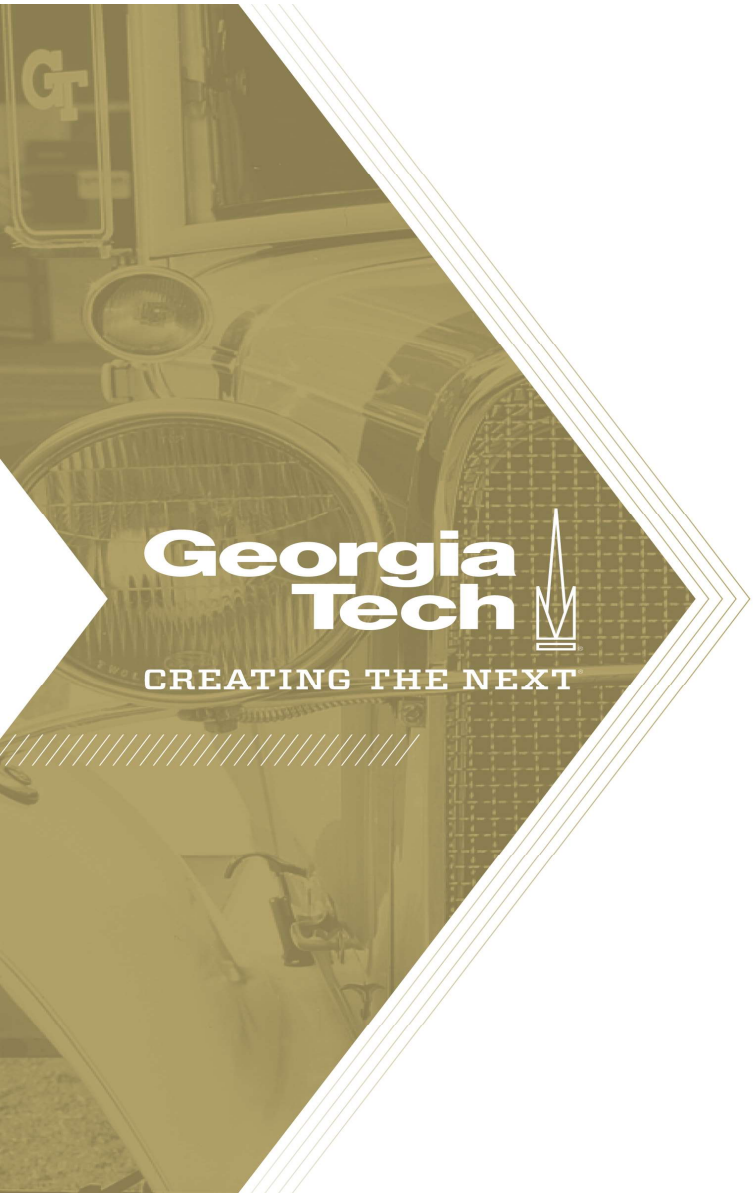


Common Subexpression Convergence (CSC)

Sana Damani and Vivek Sarkar

Habanero Extreme Scale Software Research Lab

Georgia Institute of Technology

Short paper at LCPC ’19, Atlanta, GA

Agenda

- Motivation
- Common Subexpression Convergence Transformations
- Approach
- Preliminary Results and Discussion

MOTIVATION

Divergence in SIMT processors

- SIMT (Single Instruction Multiple Threads)
  - All threads in a warp execute the same instruction in parallel

- Divergence
  - A conditional branch dependent on thread-local values
  - Threads in the warp execute different paths
  - Serialized execution of a warp

```
if (threadIdx.x < 4) {
    A;                  (1)
    B;                  (2)
} else {
    X;                  (3)
    Y;                  (4)
}
Z;                      (5)
```

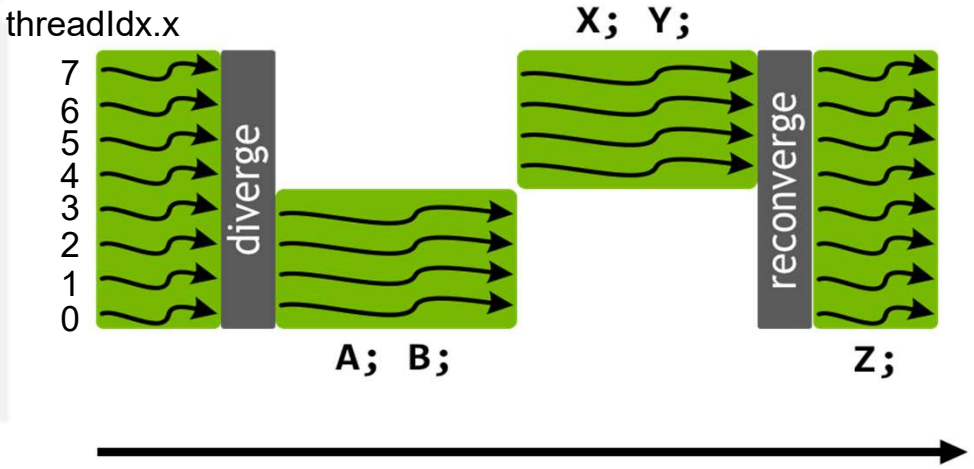


Image credits: https://devblogs.nvidia.com/inside-volta

Problem: Serialization of common code

- Divergent Code

```
b = ...;
c = ...;
if (threadId % 2)  {
  a = b * c;      // common code
  ...
} else {
  a = b * c;      // common code
  ...
}
use a;
```

- Warp Execution

COMMON SUBEXPRESSION CONVERGENCE

Hoist

```
b = ...;
c = ...;
if (threadId % 2)  {
  a = b * c;      // common code
} else {
  a = b * c;      // common code
}
use a;
```

```
b = ...;
c = ...;
a = b * c;
if (threadId % 2)  {
  use a;
} else {
  ...
}
```

- Move to convergent common ancestor

Sink

```
c = ...;
if (threadId % 2)  {
  b = 10;
  a = b * c;      // common code
} else {
  a = b * c;      // common code
}
use a;
```

```
c = ...;
if (threadId % 2)  {
  b = 10;
  ...
} else {
  ...
}
a = b * c;
use a;
```

- Move to convergent common successor

Split

```
b = ...;
c = ...;
if (threadId % 2)  {
  a = b * c;      // common code: cannot sink
  use a;
} else {
  b = 10;
  a = b * c;      // common code: cannot hoist
}
```

```
b = ...;
c = ...;
if (threadId % 2)  {
  ...
} else {
  b = 10;
}
a = b * c;
if (threadId % 2)  {
  use a;
} else {
  ...
}
```

- Move to new convergent join point
- Duplicate conditional branch
- Alternative solution: hoist defs/sink uses

Operand Renaming

```
if (threadId % 2)  {
  b = ...;
  c = ...;
  a = b * c;      // common code
} else {
  e = ...;
  f = ...;
  a = e * f;      // common code
}
use a;
```

```
if (threadId % 2)  {
  b = ...;
  c = ...;
  t1 = b;
  t2 = c;
} else {
  e = ...;
  f = ...;
  t1 = e;
  t2 = f;
}
a = t1 * t2;
use a;
```

- Insert copy instructions then sink/split

Branches

```
b = ...;
c = ...;
if (threadId % 2)  {
  a = b * c;      // common code
} else {
  if (condition) {
    a = b * c;     // conditional common code
  }
}
use a;
```

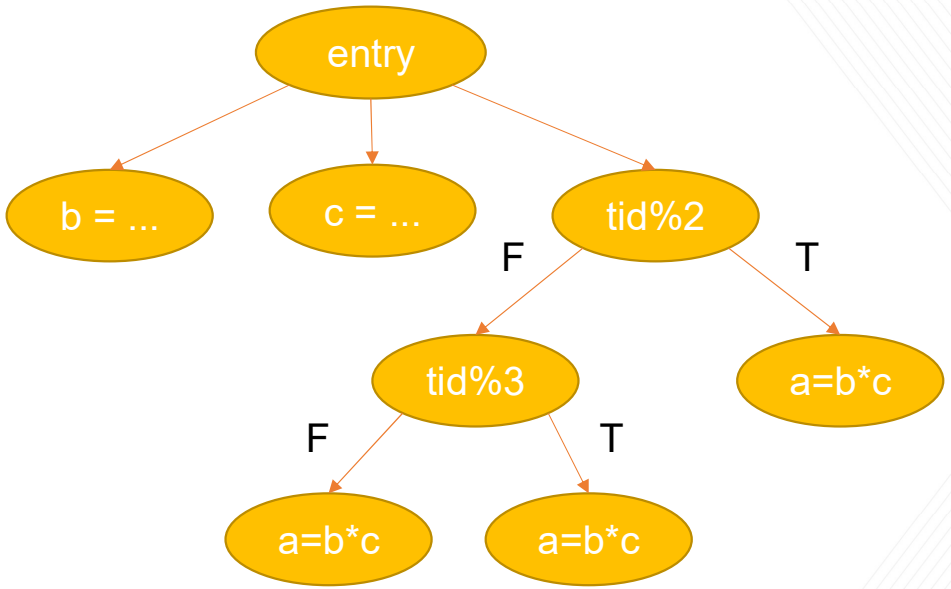
```
b = ...;
c = ...;
if (threadId % 2)  {
  p = true;
} else {
  p = condition;
}
(p) a = b * c;
use a;
```

- Flatten branch, then sink/split

Recursive CSC

```
b = ...;
c = ...;
if (threadId % 2)  {
   a = b * c;      // common code
} else {
   if (threadId % 3) {
      a = b * c;     // common code
   } else {
      a = b * c;     // common code
   }
}
use a;
```

```
b = ...;
c = ...;
a = b * c;     // common code
if (threadId % 2)  {
} else {
   if (threadId % 3) {
   } else {
   }
}
use a;
```



Bottom-Up Traversal Through CDG

Common Loops

```
b = ...;
c = ...;
if (threadId % 2)  {
  while (condition) {
    ...
    a = b * c;      // common code
  }
} else {
  while (condition) {
    ...
    a = b * c;      // common code
  }
}
use a;
```

- Loop distribution
- Index set splitting

APPROACH

Problem Statement

Given a GPU program, identify and move divergent common code to a convergent region using Hoist/Sink/Split such that dependences are preserved, and the benefit of code motion is maximized.

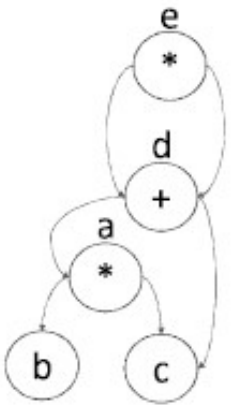
Algorithm

Algorithm 1 CSC

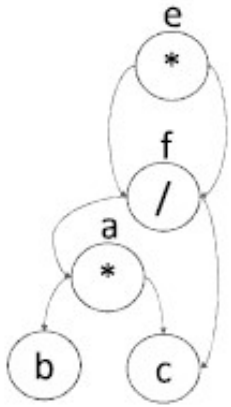
```
1: for (each region in a bottom-up traversal) do
2:     for each divergent branch do
3:         identify profitable common code;
4:         Hoist/Sink/Split;
5:     end for
6: end for
```

Identifying common code: Dynamic Programming

```
if (condition) { a = b * c;     d = a + c;     e = d * d; }
else           { a = b * c;     f = a / c;     e = f * f; }
```



Expression DAG for Taken Path



Expression DAG for Not Taken Path

	b	c	a=b*c	d=a+b	e=d*d
b	0	-Cost(copy)	-inf	-inf	-inf
c	-Cost(copy)	0	-inf	-inf	-inf
a=b*c	-inf	-inf	Benefit(*) + T(b,b) + T(c,c)	-inf	Benefit(*) + T(b,d) + T(c,d)
f=a/b	-inf	-inf	-inf	-inf	-inf
e=f*f	-inf	-inf	Benefit(*) + T(b,f) + T(c,f)	-inf	Benefit(*) + T(d,f) + T(d,f)

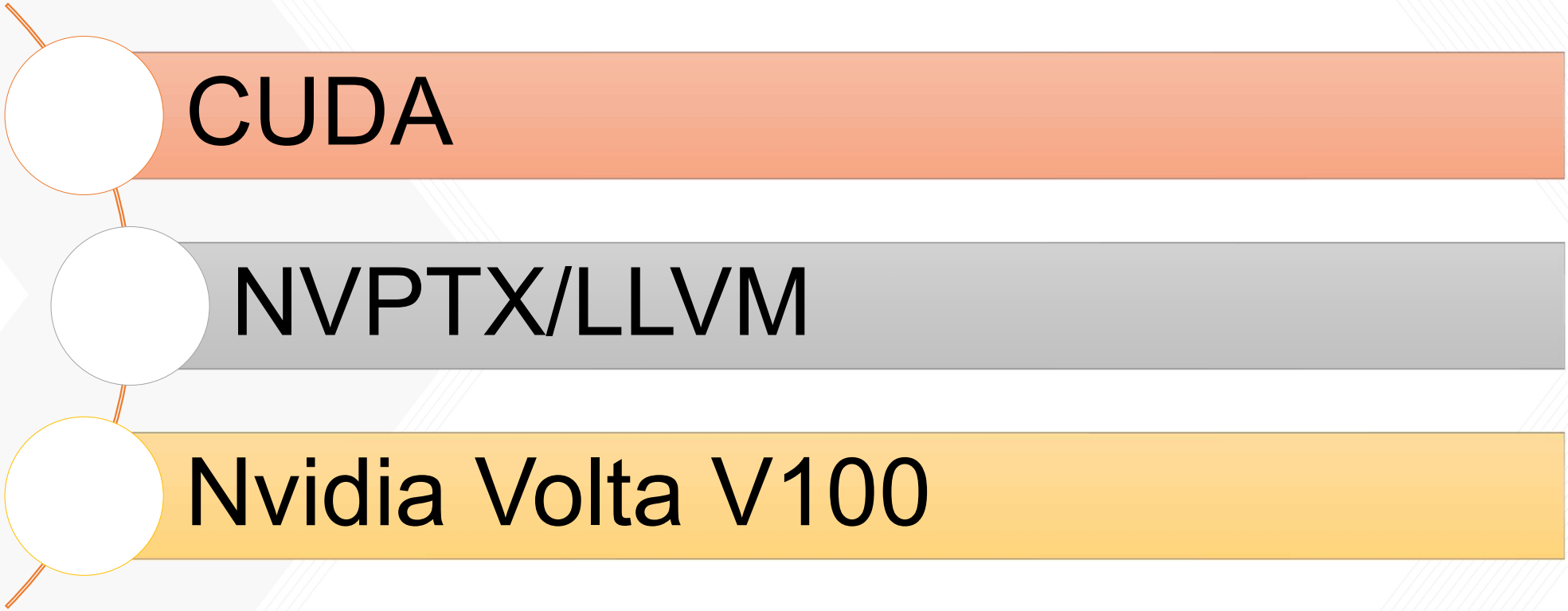
Profitability Heuristics

- **Benefit:**
  - Function Call > Memory Instructions > Math Instructions > Copy Instructions
  - Loop nest depth

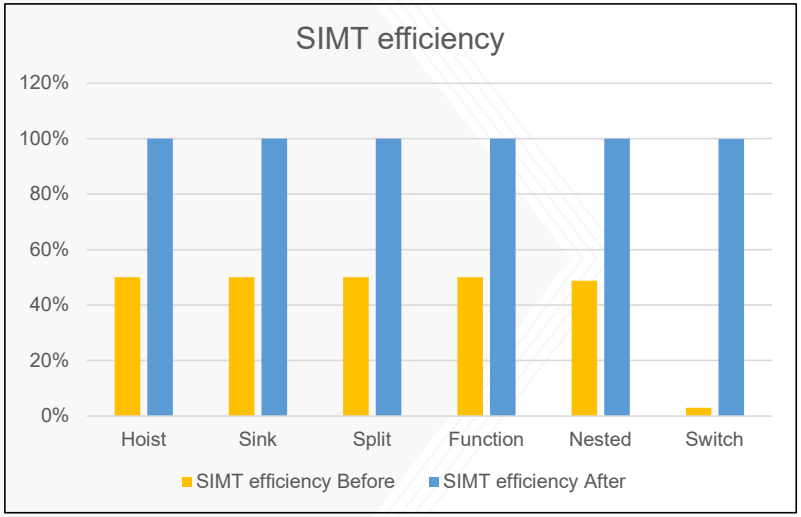
- **Cost:**
  - Copy Instructions for Operand Renaming
  - Increase in register live range and/or stalls with hoist/sink
  - Increase in branches, smaller blocks, more barriers with Split

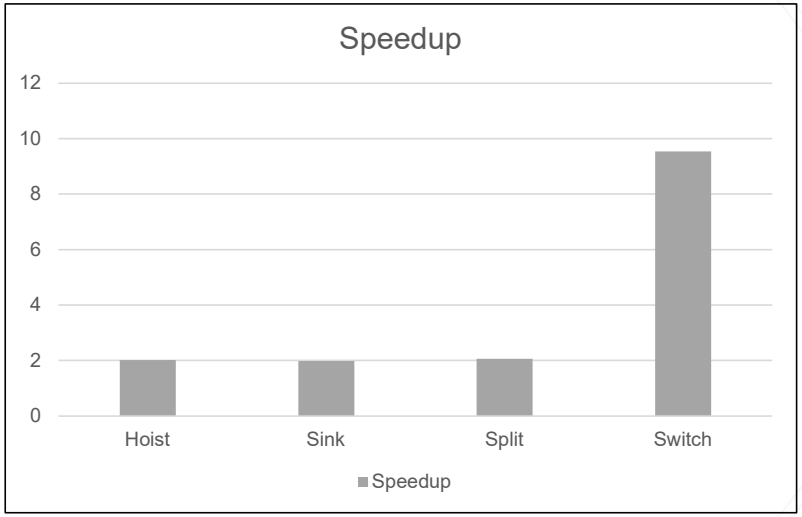
PRELIMINARY RESULTS & DISCUSSION

Experimental Setup



Preliminary Results: Microbenchmarks

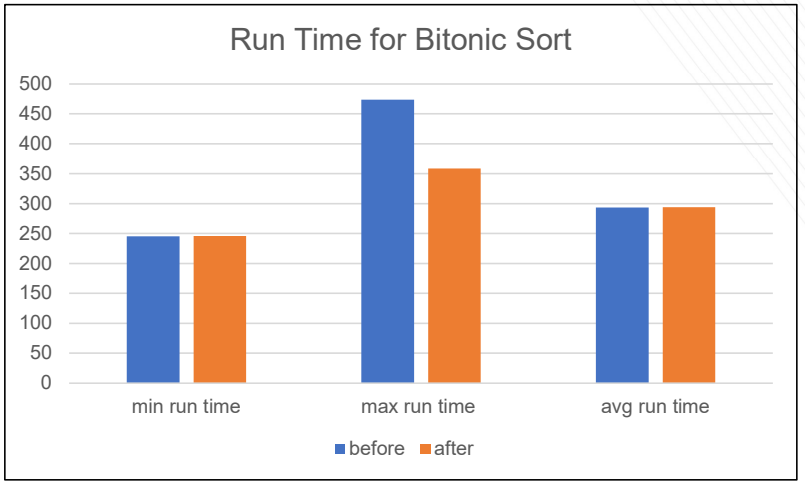


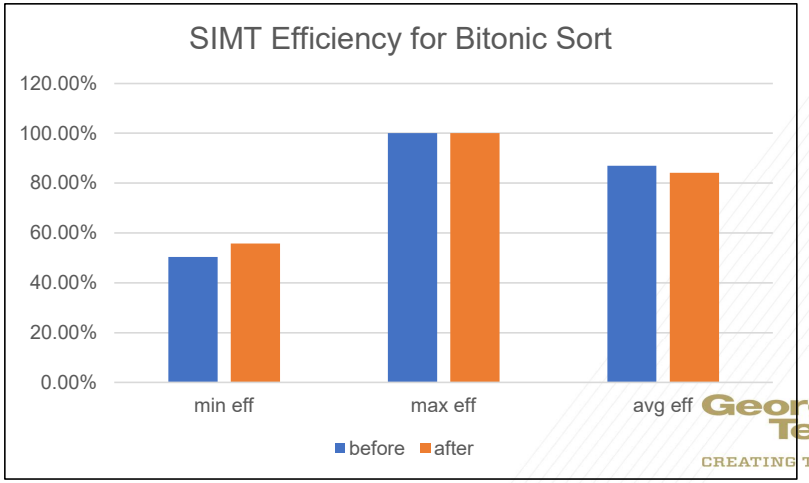


Note: nvprof shows major gains due to reduction in global reads of up to 27% with CSC (common address reads/coalesced accesses)

Preliminary Results: Bitonic Sort

```
if ((i&k)==0) {
  if (dev_values[i]>dev_values[ixj]) {
    float temp = dev_values[i];
    dev_values[i] = dev_values[ixj];
    dev_values[ixj] = temp;
  }
}
if ((i&k)!=0) {
  if (dev_values[i]<dev_values[ixj]) {
    float temp = dev_values[i];
    dev_values[i] = dev_values[ixj];
    dev_values[ixj] = temp;
  }
}
```





Discussion and Future Work

- Legality
- CSE and PRE
- Interprocedural analysis
- Opportunity in automatically parallelized programs
- Profile information for divergence, cost, bottlenecks