Measuring Memory Access Efficiency in Nsight Compute

Carl Pearson

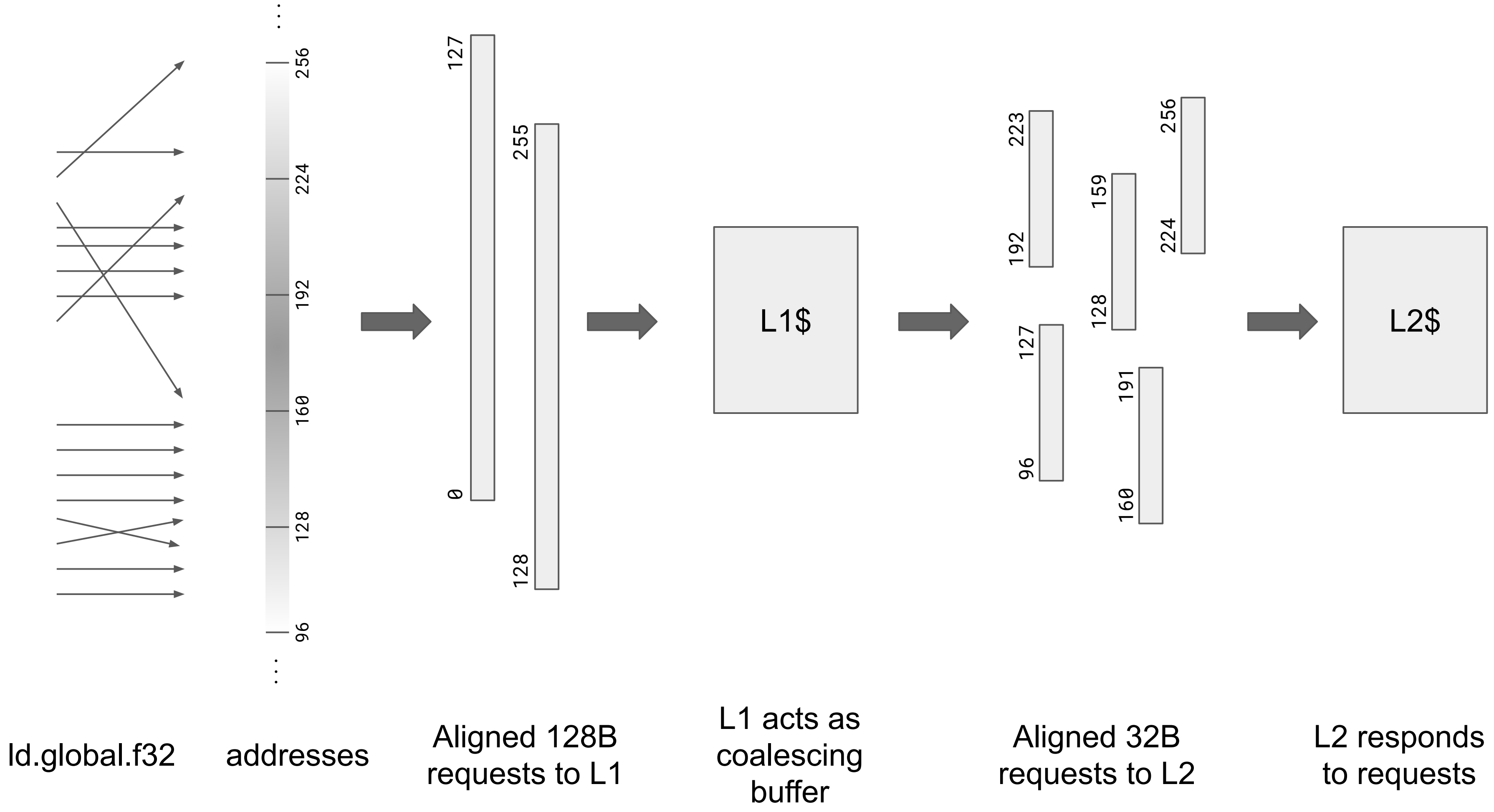
github.com/cwpearson/nvidia-performance-tools

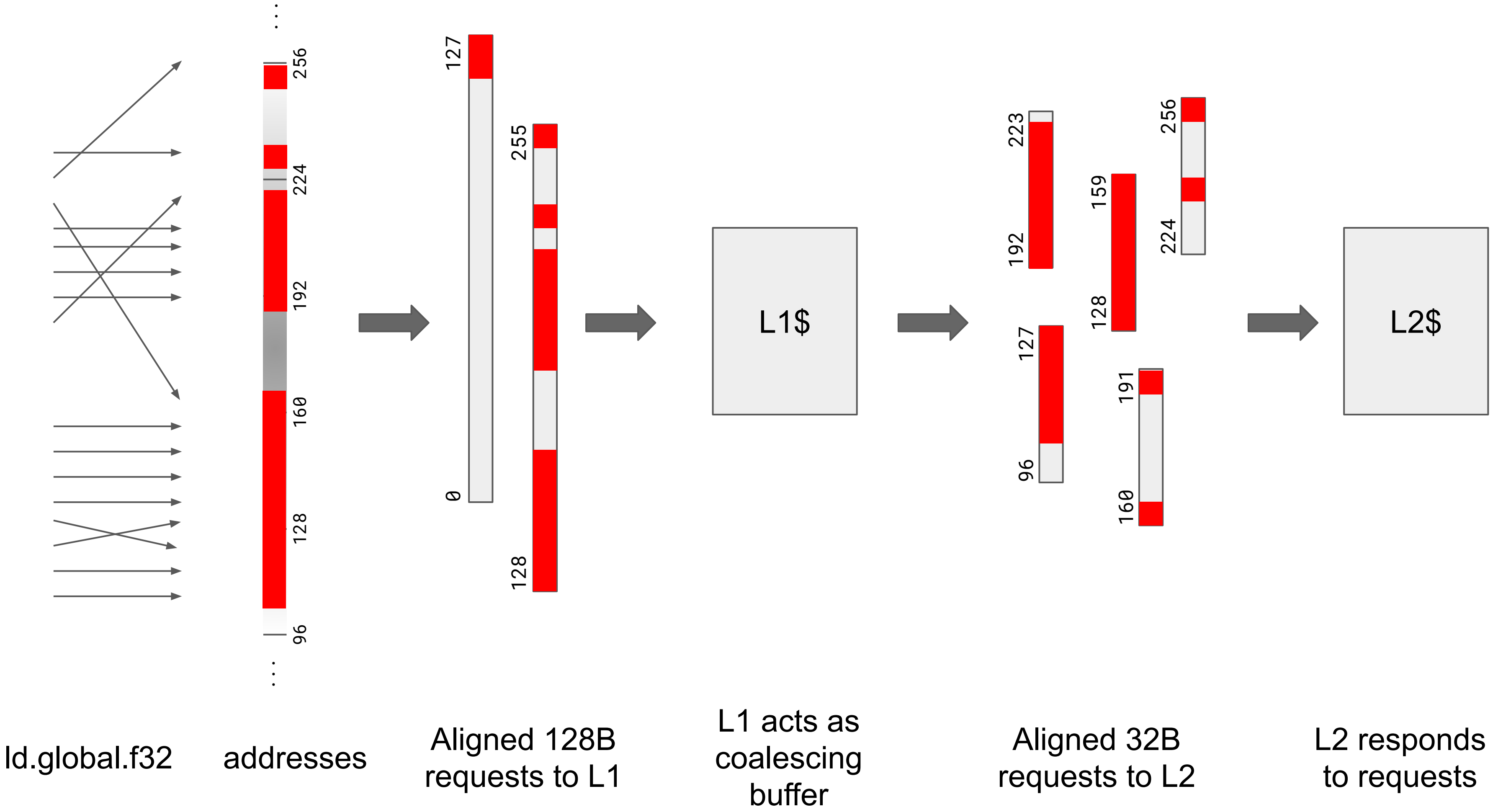
coalescing

Volta (GV100) Architecture

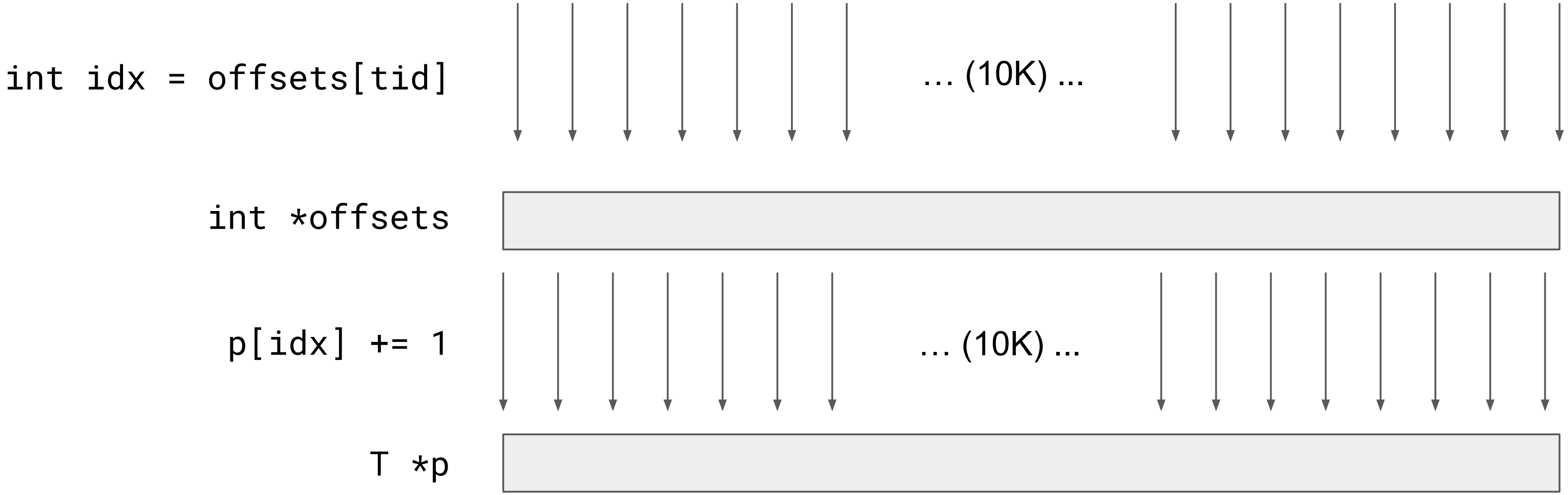
- Compute Capability 7.0
- Global memory is cached in L1[4] and L2[2]
- L1 is coalescing buffer[1]
- Cache line is aligned 128B[3]
- L1 accesses are 128B, L2 are 32B[3]
- Warp memory requests are split into independent 128B requests[3]

[1] Volta Tuning Guide, 1.4.3.2 Unified Shared Memory/L1/Texture Cache
[2] CUDA Programming Guide, H.4 Compute Capability 5.x
[3] CUDA Programming Guide, H.3 Compute Capability 3.x
[4] CUDA Programming Guide, H.6 Compute Capability 7.x


...
256
224
192
160
128
96
...
127
0
255
128
L1$
223
192
127
96
159
128
191
160
256
224
L2$
ld.global.f32
addresses
Aligned 128B requests to L1
L1 acts as coalescing buffer
Aligned 32B requests to L2
L2 responds to requests

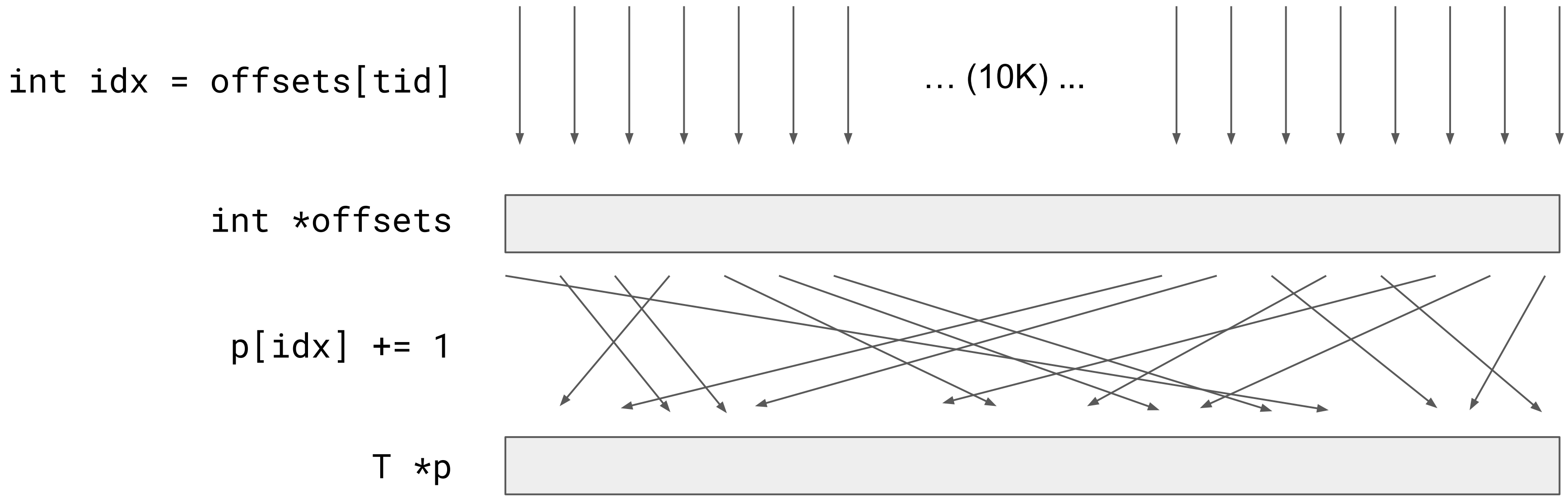

ld.global.f32
addresses
256
224
192
160
128
96
Aligned 128B requests to L1
127
0
255
128
L1$
L1 acts as coalescing buffer
Aligned 32B requests to L2
223
192
127
96
159
128
256
224
191
160
L2$
L2 responds to requests

Setup (Coalesced)



10,000 = 10K offsets and elements (p).

Setup (Coalesced)



10,000 = 10K offsets and elements (p).

Source & PTX

```
int idx = off[i];
float f = p[idx];
f += 1;
p[idx] = f;
```

```
ld.global.u32 %r9 [%rd10]
ld.global.f32 %f1 [%rd12]
add.f32 %f2, %f1, 0f3F800000
std.global.f32 [%rd12] %f2
```

```
int idx = off[i];
double f = p[idx];
f += 1;
p[idx] = f;
```

```
ld.global.u32 %r9 [%rd10]
ld.global.f64 %fd1 [%rd12]
add.f64 %fd2, %fd1, 0f3FF0000000000000
std.global.f64 [%rd12] %fd2
```

Coalesced: (Titan V, GV100, CC7.0)

Source	PTX	Instrs. Executed	Predicated-On Thread Instrs. Executed	Mem Access Size	Mem L2 Transactions Global	Mem L1 Transactions Global
`float f = p[idx];`	`ld.global.f32 %f1 [%rd12]`	313	10,000	32	1250	313
`double f = p[idx];`	`ld.global.f64 %fd1 [%rd12]`	313	10,000	64	2500	625

10K instructions / (32 threads / warp) = 312.5 warps = 313 load instructions (312 full, 1 partial)

Each float load: 32 consecutive, aligned 4B regions
= 1 128B L1 request -> 313 * 1 = 313 L1 transactions
= 4 32B L2 request -> (312 * 4) + (1 * 2) = 1250 L2 transactions

Each double load: 32 consecutive, aligned 8B regions
= 2 128B L1 request
= 8 32B L2 request
10K doubles = 80KB contiguous region
= 80KB / 128B = 625 128B L1 transactions = 2500 32B L2 transactions

Uncoalesced: (Titan V, GV100, CC7.0)

Source	PTX	Instrs. Executed	Predicated-On Thread Instrs. Executed	Mem Access Size	Mem L2 Transactions Global	Mem L1 Transactions Global
`float f = p[idx];`	`ld.global.f32 %f1 [%rd12]`	313	10,000	32	9895	9566
`double f = p[idx];`	`ld.global.f64 %fd1 [%rd12]`	313	10,000	64	9947	9740

Each float load: 32 random 4B regions
= 32 128B L1 request -> 313 * 32 = ~10K L1 transactions (some will randomly coalesce)
= 32 32B L2 request -> 313 * 31 = ~10K L2 transactions

Each double load: 32 random 8B regions
= 32 128B L1 request -> 313 * 32 = ~10K L1 transactions (some will randomly coalesce)
= 32 32B L2 request -> 313 * 31 = ~10K L2 transactions

Summary: Computing Memory Access Efficiency

Ideal L1 transactions per instruction

T_i = E * (S/8) / 128

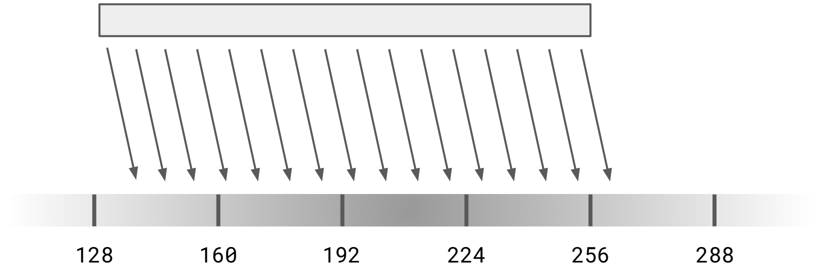
E: (Predicated-On Thread Instructions Executed)
S: Memory Access Size (bits)
128: L1 transaction size (bytes)

T_e = Actual L1 transactions: measured by nsight

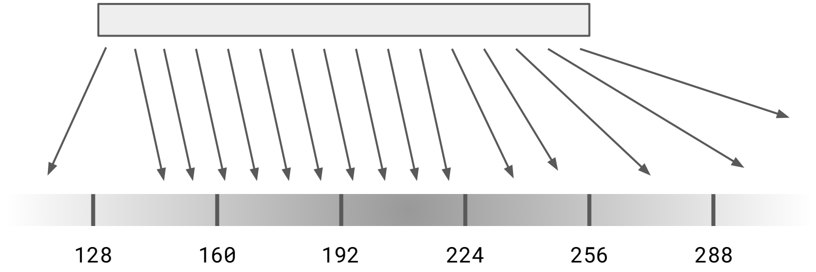
Efficiency: T_i / T_e

How to Reduce Efficiency

Unaligned:
2 requests:
128-255
256-383



Non-sequential:
3 requests:
0-127
128-255
256-383


128
160
192
224
256
288