Software

OPTIMIZING APPLICATIONS WITH INTEL® SOFTWARE TOOLS

Shiran Afergan (shiran.Afergan@intel.com)
Compute Ecosystem Engineering
April 2019

“The number of transistors on a chip will **double** approximately **every two years**.”
[Gordon Moore]

INTEL® TOOLS - ANALYZE

- Intel® VTune™ Amplifier
- Intel® Advisor
- Intel® Inspector

INTEL® TOOLS - BUILD

- Intel® C++ Compiler
- Optimized Libraries
  - Intel® Math Kernel Library
  - Intel® Data Analytics Acceleration Library
  - Intel® Integrated Performance Primitives

PERFORMANCE KILLERS

- Inefficient memory accesses
- Ineffective vectorization
- False cache sharing & Alignment
- Threading overhead
- Branch misprediction
- Unnecessary computations

```
/* a: 2d vector (ROWS, COLS) */
/* b: 1d vector (ROWS) */
//For each row calculate sum of cols.
#pragma omp parallel for
for (int i = 0; i < ROWS; i++) {
    vector<char> a_row = a[i];
    for (int j = 0; j < COLS; j++) {
        b[i] += a_row[j]
    }
}
```

DEMO

```
//For each row calculate sum of cols.
#pragma omp parallel for
for (int i = 0; i < ROWS; i++) {
    vector<char>& a_row = a[i];
    for (int j = 0; j < COLS; j++) {
        b[i] += a_row[j]
    }
}
```

DEMO

```
double full_sum = 0.0;
#pragma omp parallel for schedule(static)
for (int y = 0; y < N1; y++) {
  for (int x = 0; x < N2; x++) {
    full_sum += (y * 0.2 + x);
  }
}
```

```
double full_sum = 0.0;
#pragma omp parallel for schedule(static)\
 reduction (+:full_sum)
for (int y = 0; y < N1; y++) {
  for (int x = 0; x < N2; x++) {
    full_sum += (y * 0.2 + x);
  }
}
```

N_threads = 2 → double Sum[2];

4 elements

Thread 0

Sum[0] = a + b

Thread 1

Sum[1] = c + d

full_sum =
Sum[0] + Sum[1]

```
#pragma intel optimization_level 0
double process0(int n_threads){
   double sum[n_threads];
   double full_sum = 0.0;
   #pragma omp parallel for schedule(static)num_threads(n_threads)
   for (int y = 0; y < N1; y++) {
      int id = omp_get_thread_num();
      for (int x = 0; x < N2; x++) {
         sum[id] += (y * 0.2 + x);
      }
   }
   for (int i = 0; i < n_threads; i++)
      full_sum += sum[i];
}
```

```
Wrapper(){
  process0(1);
  process0(10);
  process0(25);
  process0(36);
}
```

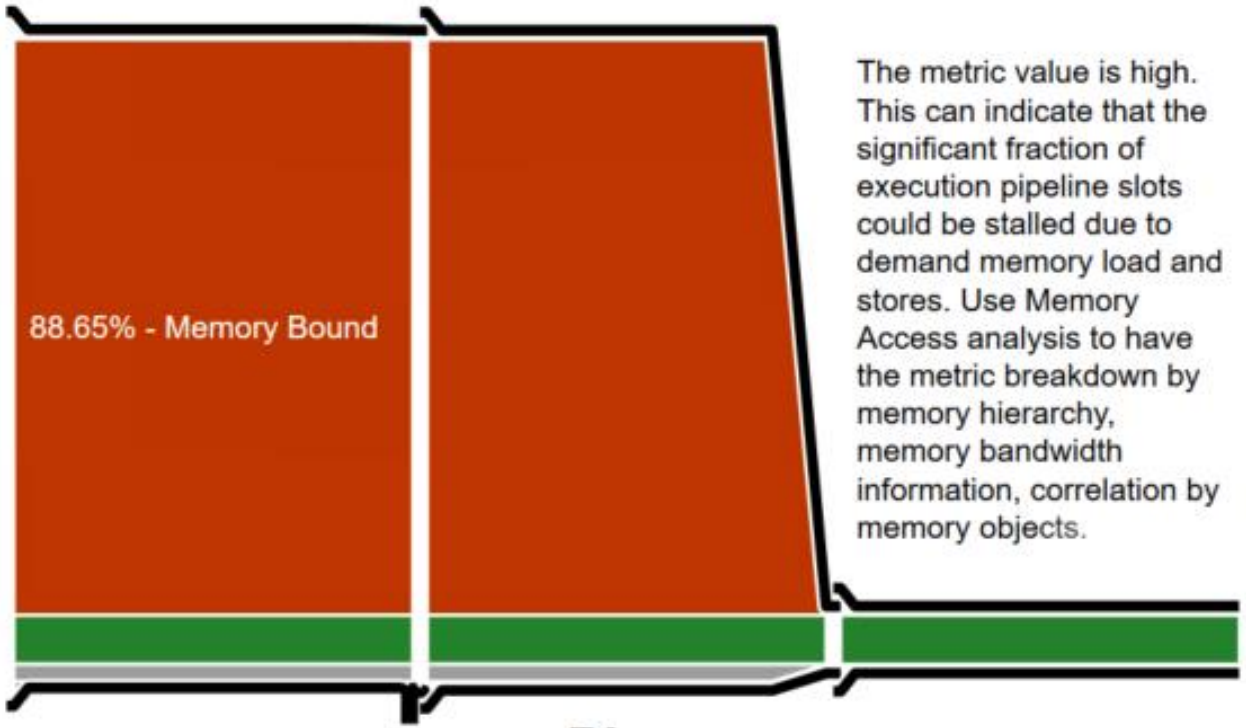
```
 num_threads = 1
Sum is -2147483648 in 40.64 seconds 1 thrds
         1.00x over single

 num_threads = 10
Sum is -2147483648 in 21.11 seconds 10 thrds
         1.93x over single

 num_threads = 25
Sum is -2147483648 in 17.51 seconds 25 thrds
         2.32x over single

 num_threads = 36
Sum is -2147483648 in 12.49 seconds 36 thrds
         3.25x over single
```

Microarchitecture Exploration Microarchitecture Exploration

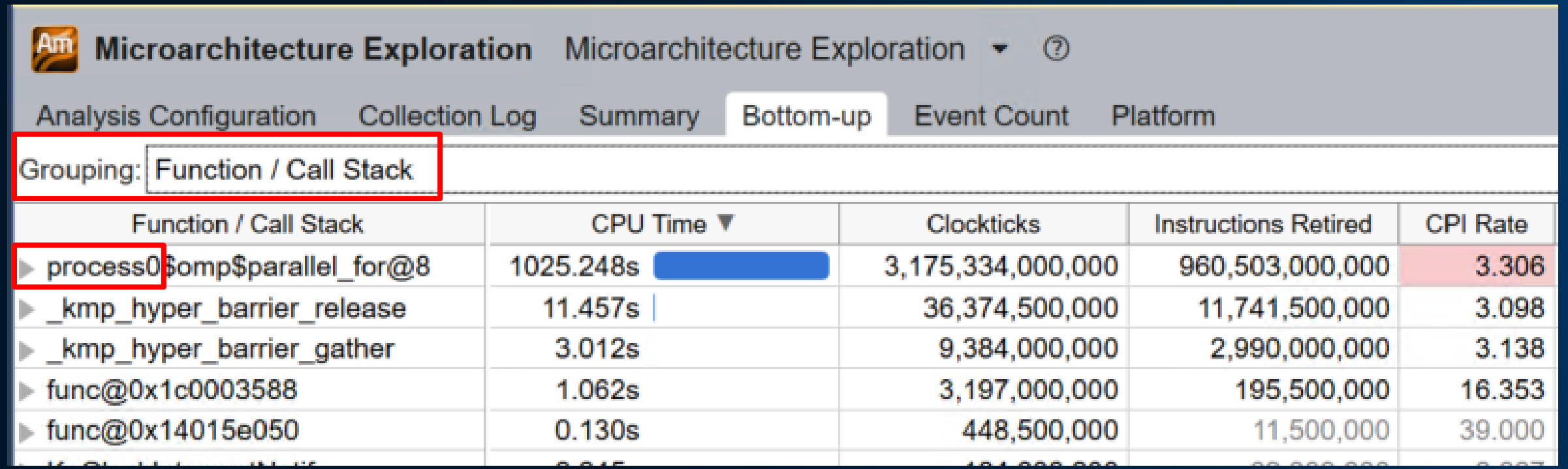
Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Elapsed Time: 101.861s

Clockticks:	3,225,531,500,000	
Instructions Retired:	975,798,000,000	
CPI Rate:	3.306	
MUX Reliability:	0.985	
Retiring:	7.6%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	1.0%	of Pipeline Slots
Back-End Bound:	91.3%	of Pipeline Slots
Memory Bound:	88.7%	of Pipeline Slots
L1 Bound:	3.1%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.6%	of Clockticks
DRAM Bound:	0.1%	of Clockticks
Store Bound:	80.7%	of Clockticks
Store Latency:	90.2%	of Clockticks
Store L2:	0.0%	of Clockticks
Store L3:	20.8%	of Clockticks
Store Memory:	69.4%	of Clockticks
False Sharing:	4.5%	of Clockticks
Split Stores:	0.0%	of Clockticks
DTLB Store Overhead:	0.0%	of Clockticks
Core Bound:	2.6%	of Pipeline Slots
Total Thread Count:	36	
Paused Time:	0.051s	
Frame Count:	4	



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.


Microarchitecture Exploration
Microarchitecture Exploration
Analysis Configuration
Collection Log
Summary
Bottom-up
Event Count
Platform
Grouping: Function / Call Stack
Function / Call Stack
CPU Time
Clockticks
Instructions Retired
CPI Rate
process0ompparallel_for@8
1025.248s
3,175,334,000,000
960,503,000,000
3.306
_kmp_hyper_barrier_release
11.457s
36,374,500,000
11,741,500,000
3.098
_kmp_hyper_barrier_gather
3.012s
9,384,000,000
2,990,000,000
3.138
func@0x1c0003588
1.062s
3,197,000,000
195,500,000
16.353
func@0x14015e050
0.130s
448,500,000
11,500,000
39.000

```
__itt_domain* proc1_d = __itt_domain_create(L"NumThreads = 1");
__itt_domain* proc2_d = __itt_domain_create(L"NumThreads = 10");
/* Same for 25 & 36 threads */


Wrapper(){
   __itt_frame_begin_v3(proc1_d, NULL);
   process0(1);
   __itt_frame_end_v3(proc1_d, NULL);
   __itt_frame_begin_v3(proc2_d, NULL);
   process0(10);
   __itt_frame_end_v3(proc2_d, NULL);
   /* Same for 25 & 36 threads */
}
```

Microarchitecture Exploration Microarchitecture Exploration

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Grouping: Frame Domain / Frame / Function / Call Stack

Frame Domain / Frame / Function / Call Stack	Frame Time ▼	Frame Count	Back-End Bound					
			Memory Bound					Core Bound
			L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
NumThreads = 1	40.744s	1	3.6%	0.0%	0.0%	0.0%	0.0%	52.4%
NumThreads = 10	32.642s	1	3.4%	0.0%	0.6%	0.1%	83.7%	2.3%
NumThreads = 25	16.816s	1	2.8%	0.0%	0.6%	0.1%	84.7%	1.8%
NumThreads = 36	11.514s	1	2.9%	0.0%	0.6%	0.1%	83.5%	2.2%

10 threads: 1.2x

25 threads: 2.4x

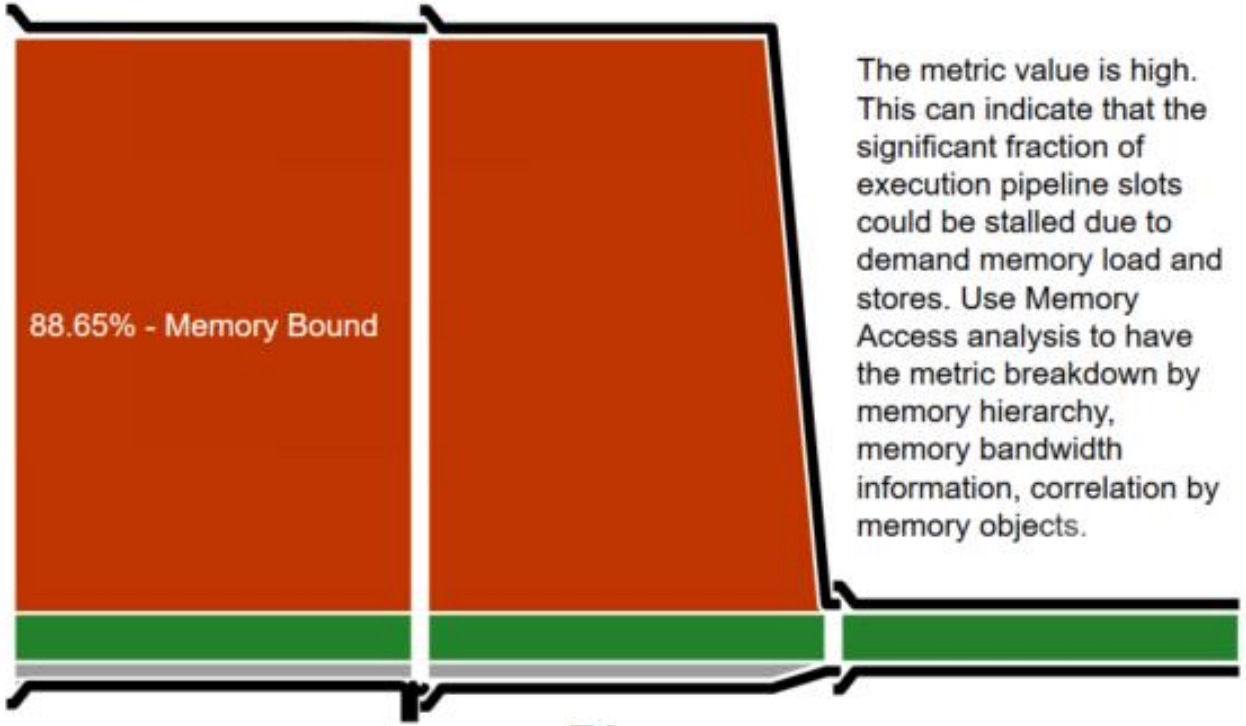
36 threads: 3.5x

Microarchitecture Exploration | Microarchitecture Exploration

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Elapsed Time: 101.861s

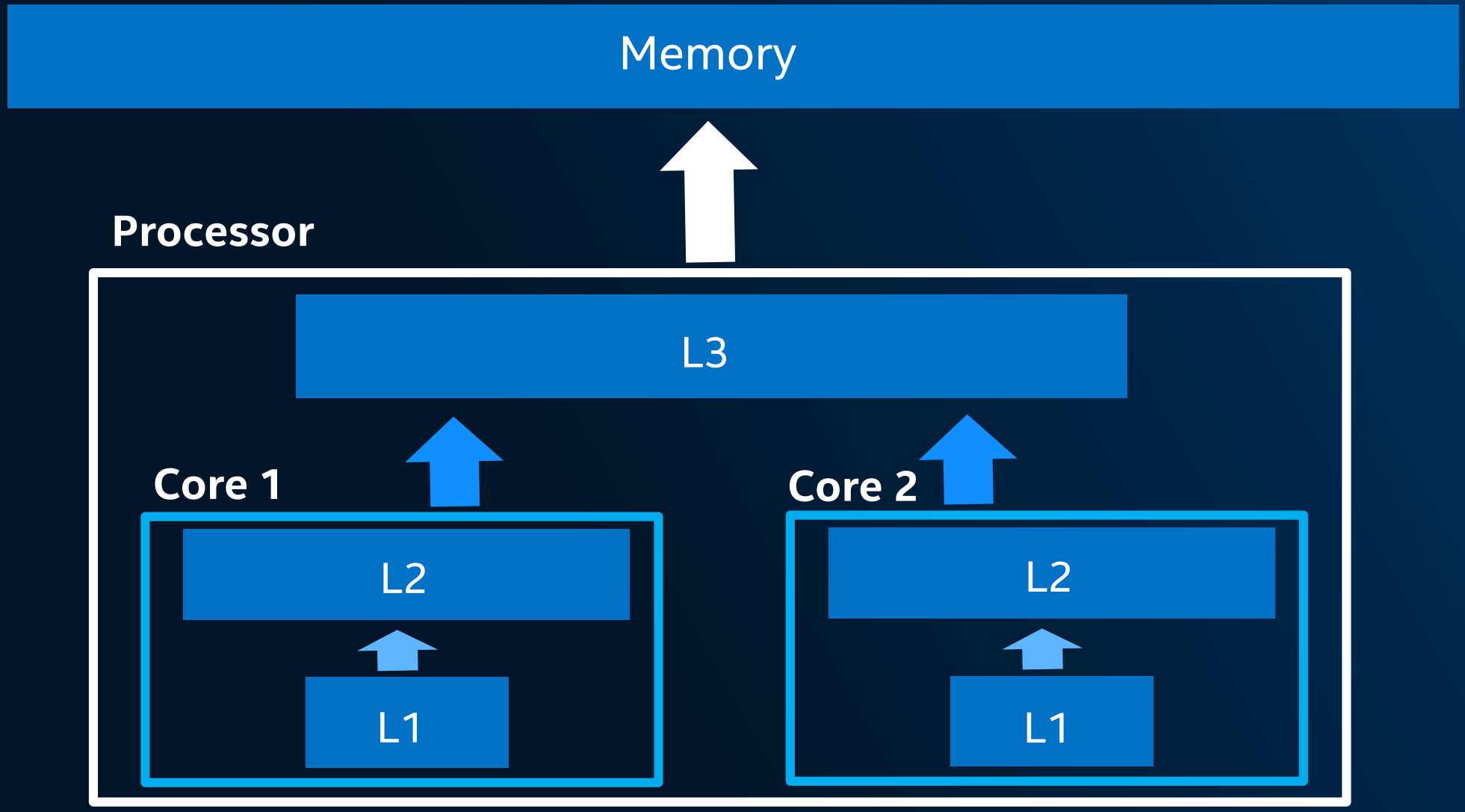
Clockticks:	3,225,531,500,000		
Instructions Retired:	975,798,000,000		
CPI Rate:	3.306		
MUX Reliability:	0.985		
Retiring:	7.6%	of Pipeline Slots	
Front-End Bound:	0.1%	of Pipeline Slots	
Bad Speculation:	1.0%	of Pipeline Slots	
Back-End Bound:	91.3%	of Pipeline Slots	
Memory Bound:	88.7%	of Pipeline Slots	
L1 Bound:	3.1%	of Clockticks	
L2 Bound:	0.0%	of Clockticks	
L3 Bound:	0.6%	of Clockticks	
DRAM Bound:	0.1%	of Clockticks	
Store Bound:	80.7%	of Clockticks	
Store Latency:	90.2%	of Clockticks	
Store L2:	0.0%	of Clockticks	
Store L3:	20.8%	of Clockticks	
Store Memory:	69.4%	of Clockticks	
False Sharing:	4.5%	of Clockticks	
Split Stores:	0.0%	of Clockticks	
DTLB Store Overhead:	0.0%	of Clockticks	
Core Bound:	2.6%	of Pipeline Slots	
Total Thread Count:	36		
Paused Time:	0.051s		
Frame Count:	4		



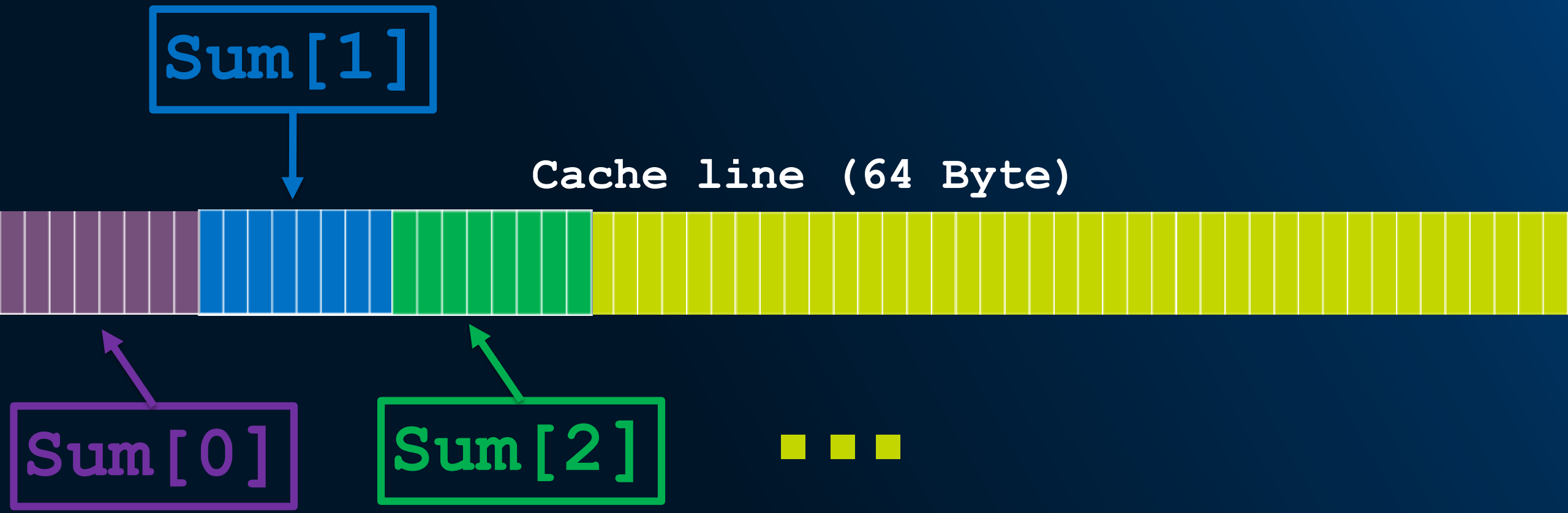
The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

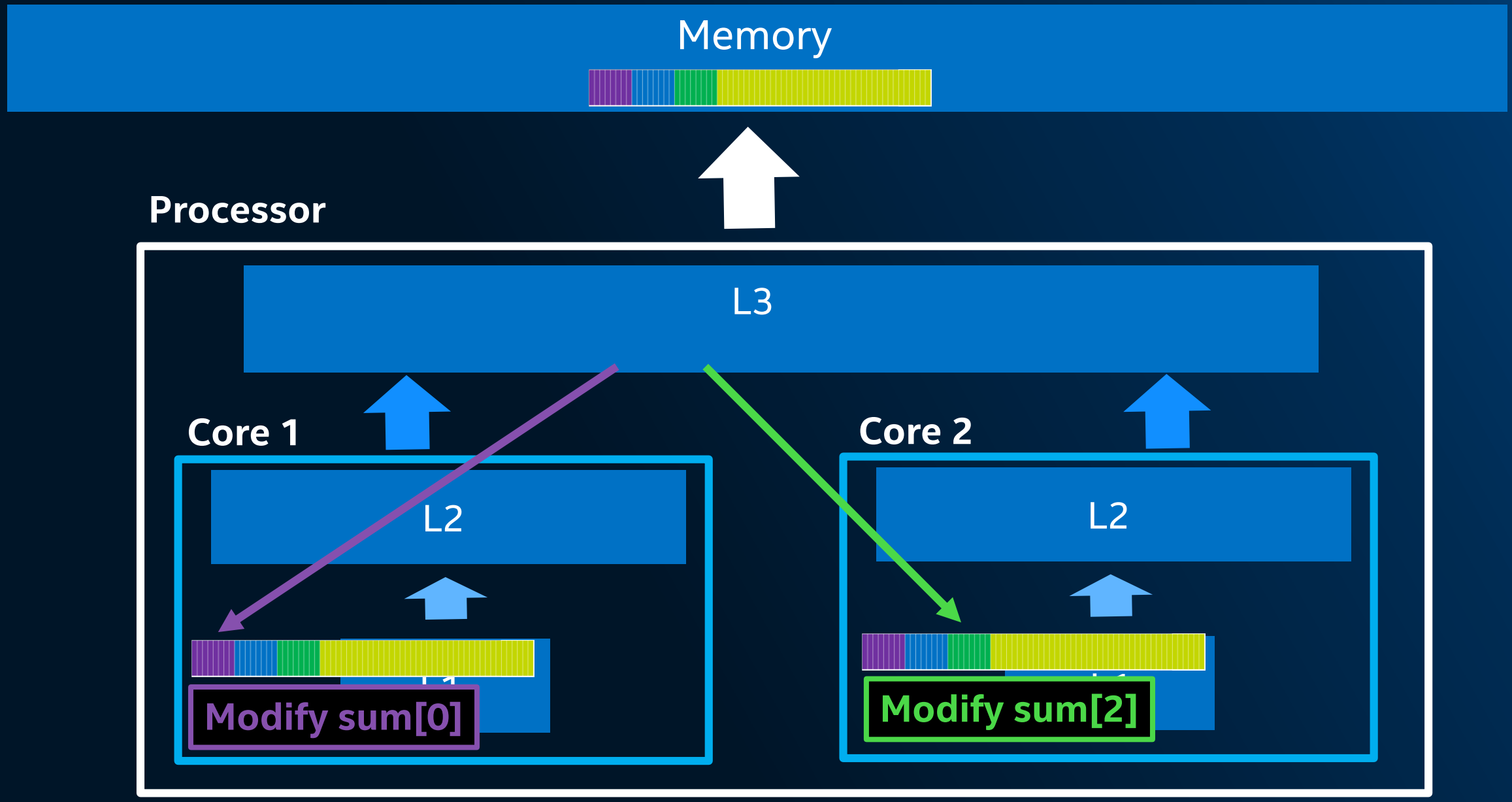
µPipe

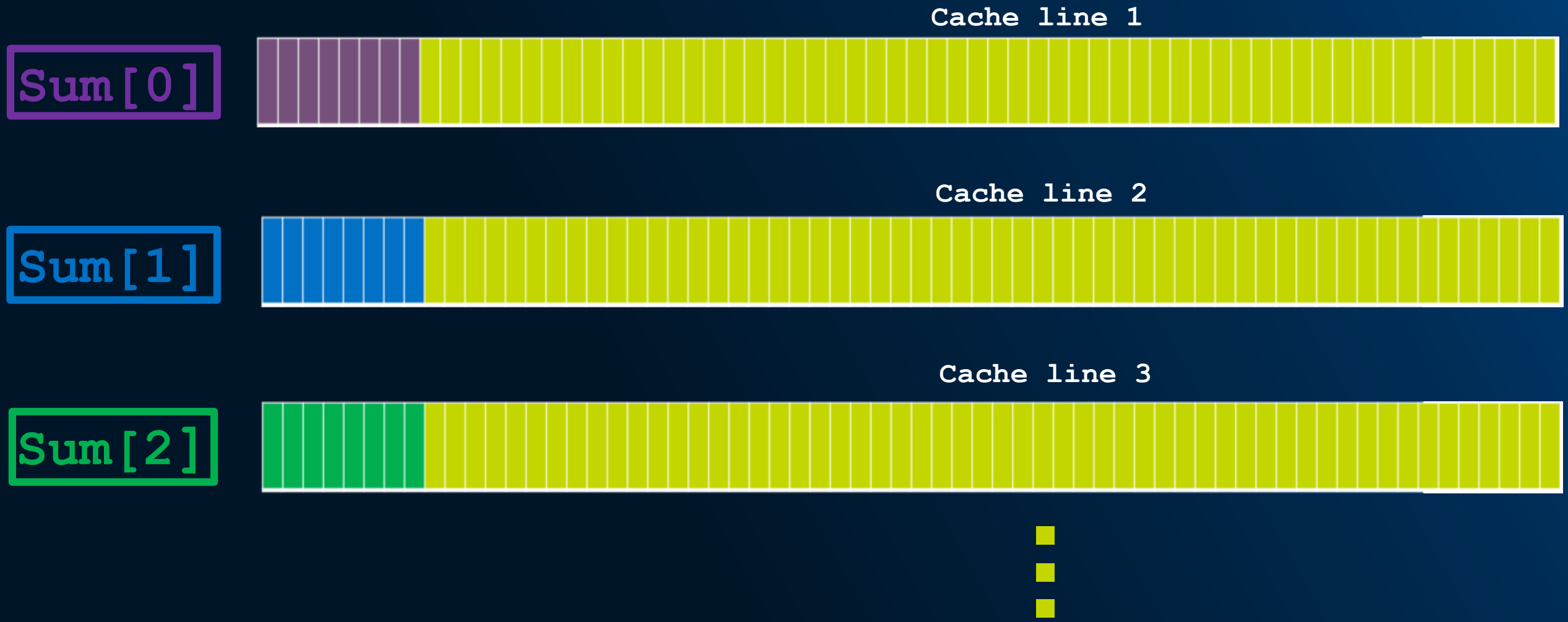
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.


Memory
Processor
L3
Core 1
L2
L1
Core 2
L2
L1

double sum[n_threads];



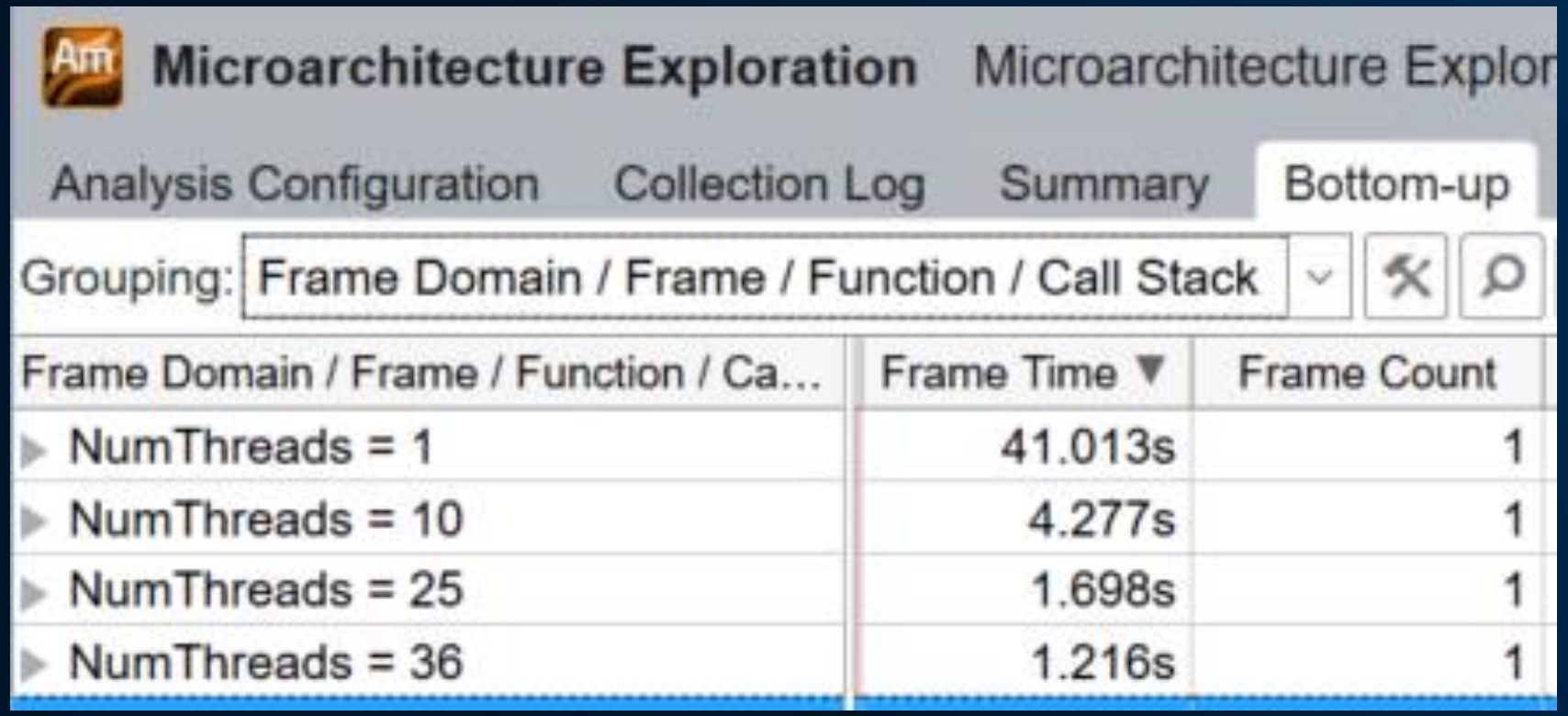

Memory
Processor
L3
Core 1
L2
Modify sum[0]
Core 2
L2
Modify sum[2]


Cache line 1
Sum[0]
Cache line 2
Sum[1]
Cache line 3
Sum[2]

```
#pragma intel optimization_level 0
double process1(int n_threads){
   double sum[n_threads * PAD];
   double full_sum = 0.0;
   #pragma omp parallel for schedule(static)num_threads(n_threads)
   for (int y = 0; y < N1; y++) {
      int id = omp_get_thread_num();
      for (int x = 0; x < N2; x++) {
         sum[id * PAD] += (y * 0.2 + x);
      }
   }
   for (int i = 0; i < n_threads; i++)
      full_sum += sum[i * PAD];
}
```

#define PAD 8

```
 num_threads = 1
Sum is -2147483648 in 40.88 seconds 1 thrds
        1.00x over single

 num_threads = 10
Sum is -2147483648 in 4.10 seconds 10 thrds
        9.97x over single

 num_threads = 25
Sum is -2147483648 in 1.64 seconds 25 thrds
        24.94x over single

 num_threads = 36
Sum is -2147483648 in 1.35 seconds 36 thrds
        30.25x over single

Press any key to continue . . .
```



Frame Domain / Frame / Function / Ca...	Frame Time ▼	Frame Count
NumThreads = 1	41.013s	1
NumThreads = 10	4.277s	1
NumThreads = 25	1.698s	1
NumThreads = 36	1.216s	1

10 threads: 10x

25 threads: 24x

36 threads: 34x

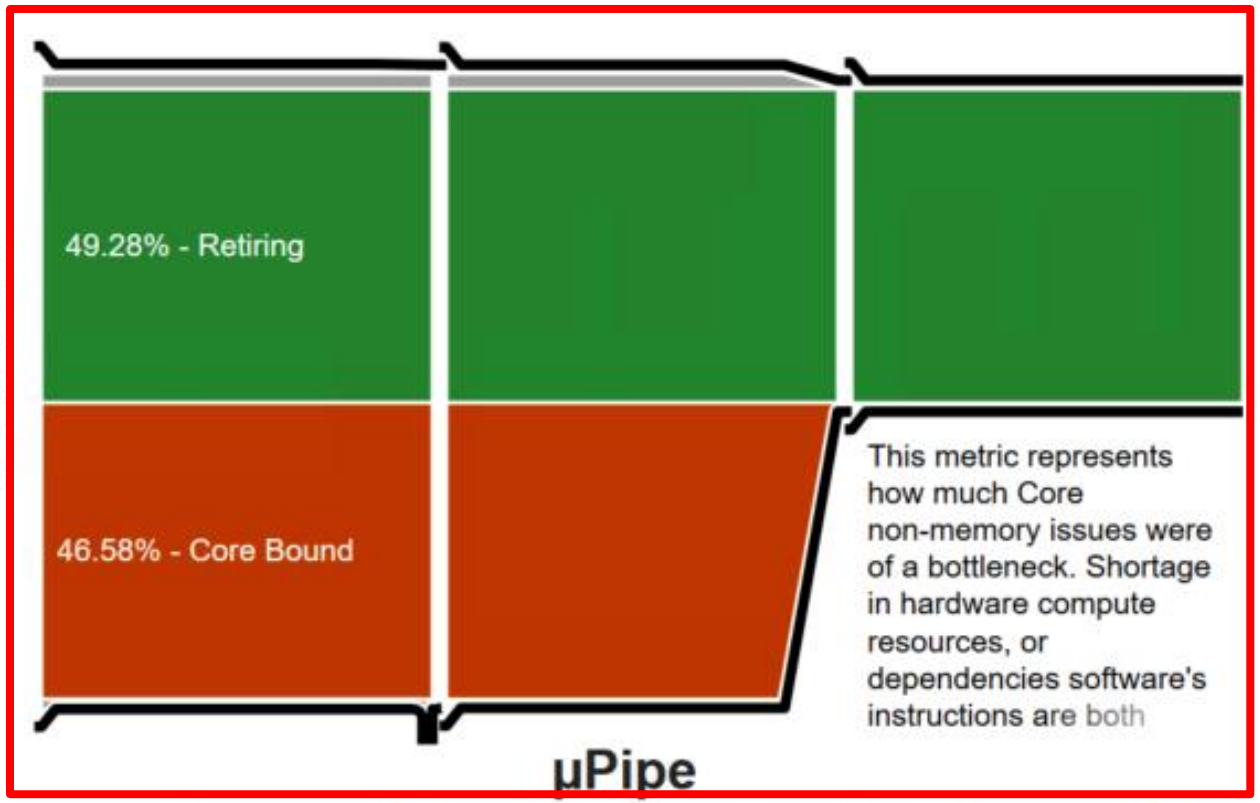
Microarchitecture Exploration Microarchitecture Exploration

Analysis Configuratic ary Bottom-up Event Count Platform Process1.h

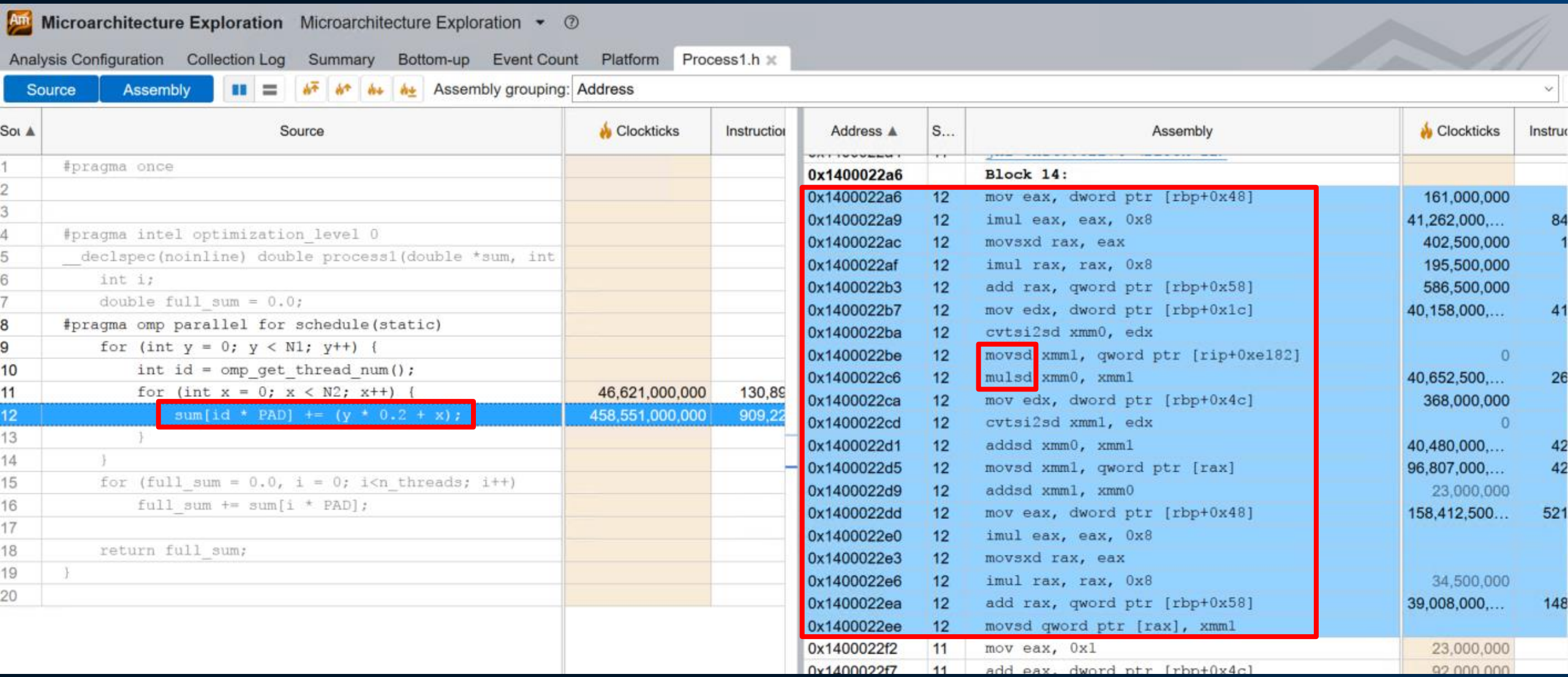
2x Boost

Elapsed Time: 48.124s

Clockticks:	512,888,500,000	
Instructions Retired:	1,042,475,000,000	
CPI Rate:	0.492	
MUX Reliability:	0.966	
Retiring:	49.3%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	1.5%	of Pipeline Slots
Back-End Bound:	49.0%	of Pipeline Slots
Memory Bound:	2.4%	of Pipeline Slots
Core Bound:	46.6%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	48.3%	of Clockticks
Cycles of 0 Ports Utilized:	0.0%	of Clockticks
Cycles of 1 Port Utilized:	17.7%	of Clockticks
Cycles of 2 Ports Utilized:	30.7%	of Clockticks
Cycles of 3+ Ports Utilized:	52.2%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Total Thread Count:	36	
Paused Time:	0.050s	
Frame Count:	4	



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.


Microarchitecture Exploration
Microarchitecture Exploration
Analysis Configuration
Collection Log
Summary
Bottom-up
Event Count
Platform
Process1.h
Source
Assembly
Assembly grouping: Address
Source
Clockticks
Address
Assembly
Clockticks
1 #pragma once
2
3
4 #pragma intel optimization_level 0
5 __declspec(noinline) double process1(double *sum, int
6 int i;
7 double full_sum = 0.0;
8 #pragma omp parallel for schedule(static)
9 for (int y = 0; y < N1; y++) {
10 int id = omp_get_thread_num();
11 for (int x = 0; x < N2; x++) { 46,621,000,000
12 sum[id * PAD] += (y * 0.2 + x); 458,551,000,000
13 }
14 }
15 for (full_sum = 0.0, i = 0; i<n_threads; i++)
16 full_sum += sum[i * PAD];
17
18 return full_sum;
19 }
20
0x1400022a6 Block 14:
0x1400022a6 12 mov eax, dword ptr [rbp+0x48] 161,000,000
0x1400022a9 12 imul eax, eax, 0x8 41,262,000,...
0x1400022ac 12 movsxd rax, eax 402,500,000
0x1400022af 12 imul rax, rax, 0x8 195,500,000
0x1400022b3 12 add rax, qword ptr [rbp+0x58] 586,500,000
0x1400022b7 12 mov edx, dword ptr [rbp+0x1c] 40,158,000,...
0x1400022ba 12 cvtsi2sd xmm0, edx
0x1400022be 12 movsd xmm1, qword ptr [rip+0xe182] 0
0x1400022c6 12 mulsd xmm0, xmm1 40,652,500,...
0x1400022ca 12 mov edx, dword ptr [rbp+0x4c] 368,000,000
0x1400022cd 12 cvtsi2sd xmm1, edx 0
0x1400022d1 12 addsd xmm0, xmm1 40,480,000,...
0x1400022d5 12 movsd xmm1, qword ptr [rax] 96,807,000,...
0x1400022d9 12 addsd xmm1, xmm0 23,000,000
0x1400022dd 12 mov eax, dword ptr [rbp+0x48] 158,412,500...
0x1400022e0 12 imul eax, eax, 0x8
0x1400022e3 12 movsxd rax, eax
0x1400022e6 12 imul rax, rax, 0x8 34,500,000
0x1400022ea 12 add rax, qword ptr [rbp+0x58] 39,008,000,...
0x1400022ee 12 movsd qword ptr [rax], xmm1
0x1400022f2 11 mov eax, 0x1 23,000,000

```
#pragma intel optimization_level 0
double process1(int n_threads){
   double sum[n_threads * PAD];
   double full_sum = 0.0;
   #pragma omp parallel for schedule(static)num_threads(n_threads)
   for (int y = 0; y < N1; y++) {
      int id = omp_get_thread_num();
      for (int x = 0; x < N2; x++) {
         sum[id * PAD] += (y * 0.2 + x);
      }
   }
   for (int i = 0; i < n_threads; i++)
      full_sum += sum[i * PAD];
}
```

```
#define PAD 8
```

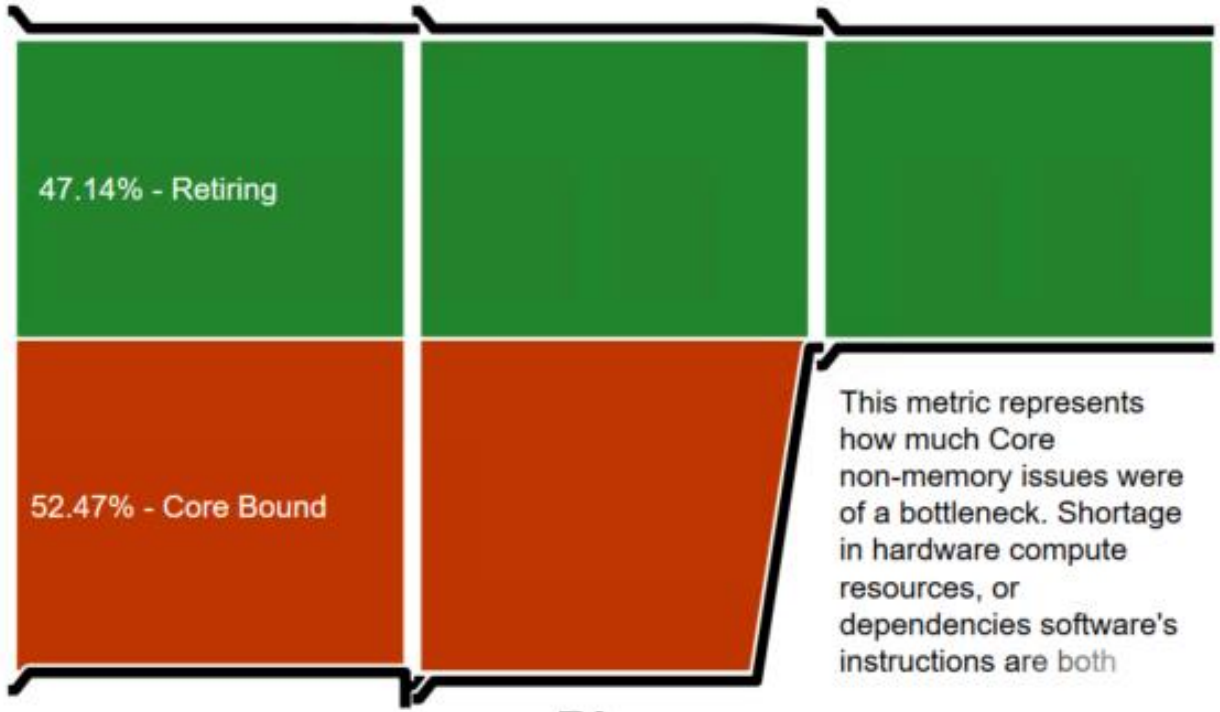
3x Boost (x6 overall)

Microar… itecture Exploration

Analysis Cont… | Bottom-up | Event Count | Platform | Process2.h

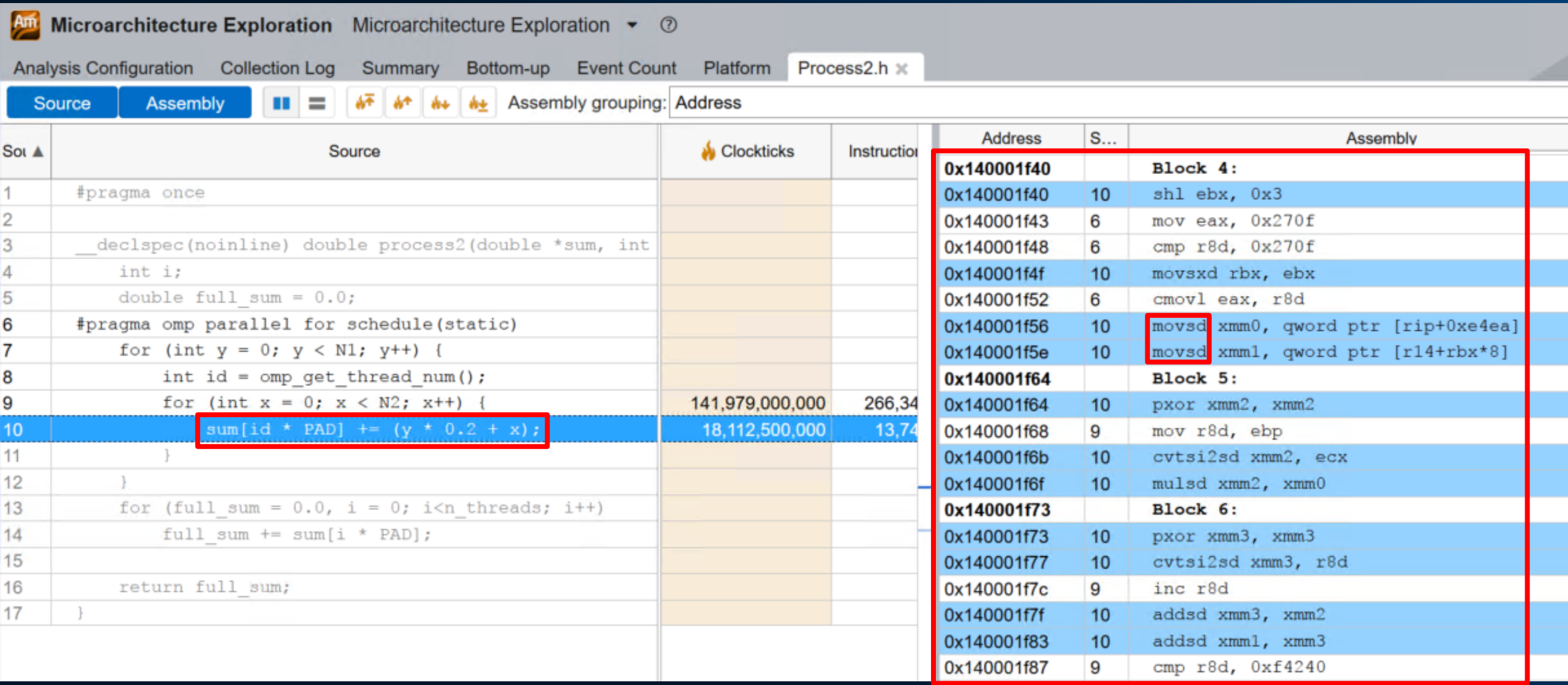
Elapsed Time: 15.490s

Clockticks:	163,058,500,000	
Instructions Retired:	280,887,500,000	
CPI Rate:	0.581	
MUX Reliability:	0.824	
Retiring:	47.1%	of Pipeline Slots
Front-End Bound:	0.0%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	52.8%	of Pipeline Slots
Memory Bound:	0.3%	of Pipeline Slots
Core Bound:	52.5%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	43.7%	of Clockticks
Cycles of 0 Ports Utilized:	13.6%	of Clockticks
Cycles of 1 Port Utilized:	30.1%	of Clockticks
Cycles of 2 Ports Utilized:	23.4%	of Clockticks
Cycles of 3+ Ports Utilized:	16.9%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Total Thread Count:	36	
Paused Time:	0.047s	
Frame Count:	4	



µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.


Microarchitecture Exploration
Microarchitecture Exploration
Analysis Configuration
Collection Log
Summary
Bottom-up
Event Count
Platform
Process2.h
Source
Assembly
Assembly grouping: Address
Source
Clockticks
Instructio
1 #pragma once
2
3 __declspec(noinline) double process2(double *sum, int
4 int i;
5 double full_sum = 0.0;
6 #pragma omp parallel for schedule(static)
7 for (int y = 0; y < N1; y++) {
8 int id = omp_get_thread_num();
9 for (int x = 0; x < N2; x++) { 141,979,000,000 266,34
10 sum[id * PAD] += (y * 0.2 + x); 18,112,500,000 13,74
11 }
12 }
13 for (full_sum = 0.0, i = 0; i<n_threads; i++)
14 full_sum += sum[i * PAD];
15
16 return full_sum;
17 }
Address
S...
Assembly
0x140001f40 Block 4:
0x140001f40 10 shl ebx, 0x3
0x140001f43 6 mov eax, 0x270f
0x140001f48 6 cmp r8d, 0x270f
0x140001f4f 10 movsxd rbx, ebx
0x140001f52 6 cmovl eax, r8d
0x140001f56 10 movsd xmm0, qword ptr [rip+0xe4ea]
0x140001f5e 10 movsd xmm1, qword ptr [r14+rbx*8]
0x140001f64 Block 5:
0x140001f64 10 pxor xmm2, xmm2
0x140001f68 9 mov r8d, ebp
0x140001f6b 10 cvtsi2sd xmm2, ecx
0x140001f6f 10 mulsd xmm2, xmm0
0x140001f73 Block 6:
0x140001f73 10 pxor xmm3, xmm3
0x140001f77 10 cvtsi2sd xmm3, r8d
0x140001f7c 9 inc r8d
0x140001f7f 10 addsd xmm3, xmm2
0x140001f83 10 addsd xmm1, xmm3
0x140001f87 9 cmp r8d, 0xf4240

```
__declspec(noinline) double process2(double *sum, int n_threads) {
    int i;
    double full_sum = 0.0;
#pragma omp parallel for schedule(static)
```

1 optimization note

Main loop

15541: outer loop was not auto-vectorized: consider using SIMD directive (line: 7, column: 2)

```
    for (int y = 0; y < N1; y++) {
        int id = omp_get_thread_num();
```

1 optimization note

Main loop

15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization. Consider using fast FP model (line: 10, column: 4)

```
        for (int x = 0; x < N2; x++) {
            sum[id * PAD] += (y * 0.2 + x);
        }
    }
```

1 optimization note

Main loop

15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization. Consider using fast FP model (line: 14, column: 3)

```
    for (full_sum = 0.0, i = 0; i<n_threads; i++)
        full_sum += sum[i * PAD];

    return full_sum;
}
```

```
#pragma float_control(fast, on, push)
#pragma intel optimization_parameter target_arch=COMMON-AVX512
__declspec(noinline) double process3(double *sum, int n_threads) {
    int i;
    double full_sum = 0.0;
#pragma omp parallel for schedule(static)
```

1 optimization note

▸ *Main loop*

```
    for (int y = 0; y < N1; y++) {
        int id = omp_get_thread_num();
```

1 optimization note

▾ *Main loop*

- *15300: LOOP WAS VECTORIZED (line: 11, column: 3)*
- *15475: --- begin vector cost summary --- (line: 11, column: 3)*
- *15476: scalar cost: 13 (line: 11, column: 3)*
- *15477: vector cost: 1.000 (line: 11, column: 3)*
- *15478: estimated potential speedup: 12.990 (line: 11, column: 3)*
- *15487: type converts: 1 (line: 11, column: 3)*
- *15488: --- end vector cost summary --- (line: 11, column: 3)*

```
        for (int x = 0; x < N2; x++) {
            sum[id * PAD] += (y * 0.2 + x);
        }
    }
```

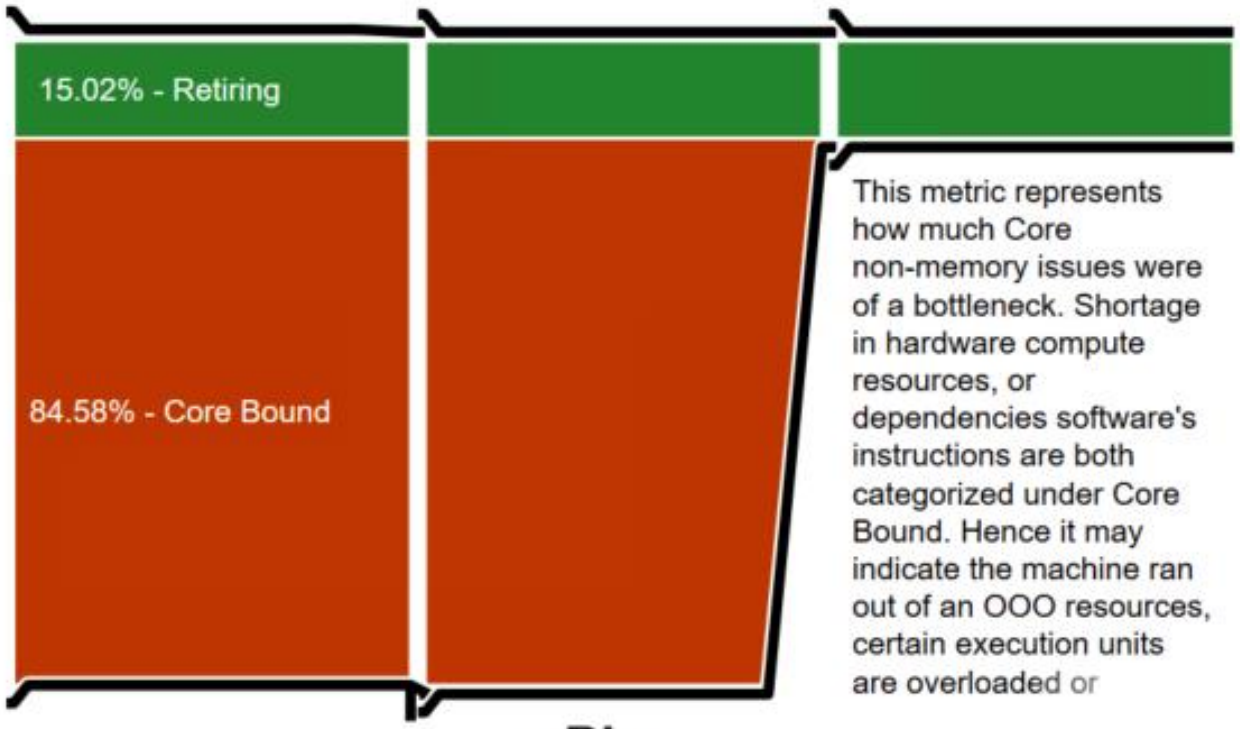
6x Boost (x43 overall)

Microarc… ure Exploration

Analysis Confi… ottom-up | Event Count | Platform | Process3.h

Elapsed Time: 2.334s

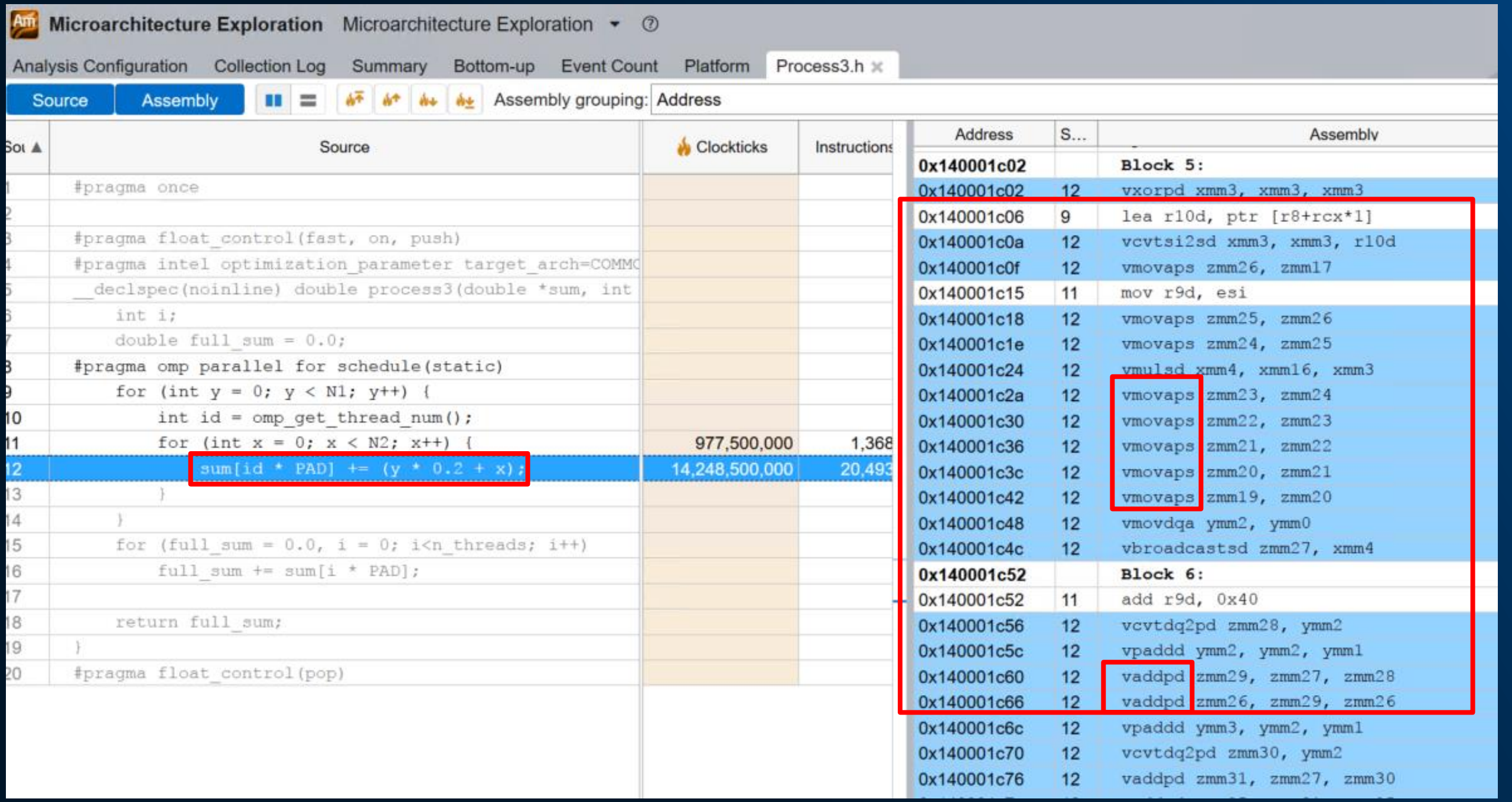
Clockticks:	17,710,000,000	
Instructions Retired:	22,678,000,000	
CPI Rate:	0.781	
MUX Reliability:	0.904	
Retiring:	15.0%	of Pipeline Slots
Front-End Bound:	0.4%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	84.6%	of Pipeline Slots
Memory Bound:	0.0%	of Pipeline Slots
Core Bound:	84.6%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	15.8%	of Clockticks
Cycles of 0 Ports Utilized:	0.0%	of Clockticks
Cycles of 1 Port Utilized:	15.8%	of Clockticks
Cycles of 2 Ports Utilized:	30.8%	of Clockticks
Cycles of 3+ Ports Utilized:	4.7%	of Clockticks
Vector Capacity Usage (FPU):	100.0%	
Total Thread Count:	36	
Paused Time:	0.049s	
Frame Count:	4	



This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.


Microarchitecture Exploration
Microarchitecture Exploration
Analysis Configuration
Collection Log
Summary
Bottom-up
Event Count
Platform
Process3.h
Source
Assembly
Assembly grouping: Address
Source
Clockticks
Instructions
#pragma once
#pragma float_control(fast, on, push)
#pragma intel optimization_parameter target_arch=COMMC
__declspec(noinline) double process3(double *sum, int
int i;
double full_sum = 0.0;
#pragma omp parallel for schedule(static)
for (int y = 0; y < N1; y++) {
int id = omp_get_thread_num();
for (int x = 0; x < N2; x++) {
977,500,000
1,368
sum[id * PAD] += (y * 0.2 + x);
14,248,500,000
20,493
}
}
for (full_sum = 0.0, i = 0; i<n_threads; i++)
full_sum += sum[i * PAD];
return full_sum;
}
#pragma float_control(pop)
Address
S...
Assembly
0x140001c02
Block 5:
0x140001c02 12 vxorpd xmm3, xmm3, xmm3
0x140001c06 9 lea r10d, ptr [r8+rcx*1]
0x140001c0a 12 vcvtsi2sd xmm3, xmm3, r10d
0x140001c0f 12 vmovaps zmm26, zmm17
0x140001c15 11 mov r9d, esi
0x140001c18 12 vmovaps zmm25, zmm26
0x140001c1e 12 vmovaps zmm24, zmm25
0x140001c24 12 vmulsd xmm4, xmm16, xmm3
0x140001c2a 12 vmovaps zmm23, zmm24
0x140001c30 12 vmovaps zmm22, zmm23
0x140001c36 12 vmovaps zmm21, zmm22
0x140001c3c 12 vmovaps zmm20, zmm21
0x140001c42 12 vmovaps zmm19, zmm20
0x140001c48 12 vmovdqa ymm2, ymm0
0x140001c4c 12 vbroadcastsd zmm27, xmm4
0x140001c52
Block 6:
0x140001c52 11 add r9d, 0x40
0x140001c56 12 vcvtdq2pd zmm28, ymm2
0x140001c5c 12 vpaddd ymm2, ymm2, ymm1
0x140001c60 12 vaddpd zmm29, zmm27, zmm28
0x140001c66 12 vaddpd zmm26, zmm29, zmm26
0x140001c6c 12 vpaddd ymm3, ymm2, ymm1
0x140001c70 12 vcvtdq2pd zmm30, ymm2
0x140001c76 12 vaddpd zmm31, zmm27, zmm30

```
class Fruit {
public:
    virtual float get_dim() const noexcept { return 3; }
};
class Apple : public Fruit {
public:
    float get_dim() const noexcept { return 1.2; }
};
class Banana : public Fruit {
public:
    float get_dim() const noexcept { return 2.5; }
};
```

```
double sumOfFruits(std::vector<Fruit*>*vec) {
    double sum = 0;
    for (auto *fruit : (*vec)) {
        sum += fruit->get_dim();
    }
    return sum;
}
```

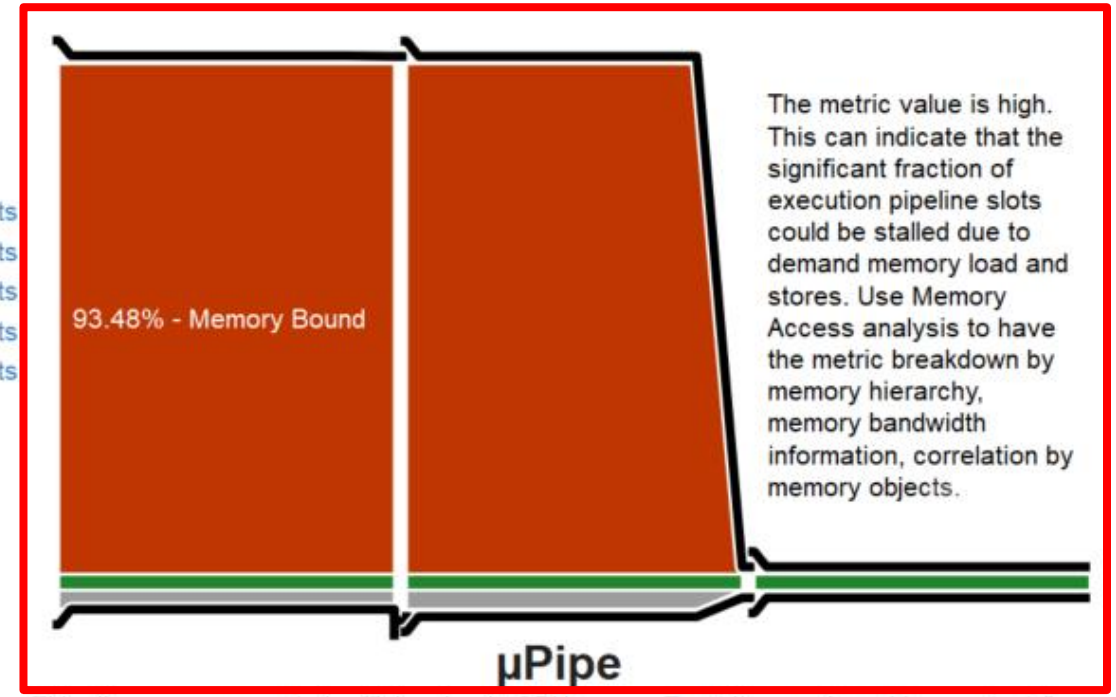
DEMO

```
//init
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++){
      Frame[i * COLUMNS + j] = 2 * i + 3 * j;
   }
}
//Process
#pragma omp parallel for schedule(static)
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
       Frame[i * COLUMNS + j] = Frame[i * COLUMNS + j] * Factor + Factor;
   }
}
```

MICROARCHITECTURE EXPLORATION

Elapsed Time: 30.296s

Clockticks:	1,342,362,500,000	
Instructions Retired:	156,562,500,000	
CPI Rate:	8.574	
MUX Reliability:	0.997	
Retiring:	2.9%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Back-End Bound:	96.9%	of Pipeline Slots
Memory Bound:	93.5%	of Pipeline Slots
L1 Bound:	0.3%	of Clockticks
DTLB Overhead:	27.9%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.1%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.4%	of Clockticks
L3 Bound:	1.3%	of Clockticks
DRAM Bound:	81.7%	of Clockticks
Memory Bandwidth:	68.2%	of Clockticks
Memory Latency:	18.7%	of Clockticks
Local DRAM:	44.4%	of Clockticks
Remote DRAM:	67.8%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	3.5%	of Pipeline Slots
Total Thread Count:	56	
Paused Time:	22.227s	
Frame Count:	5	



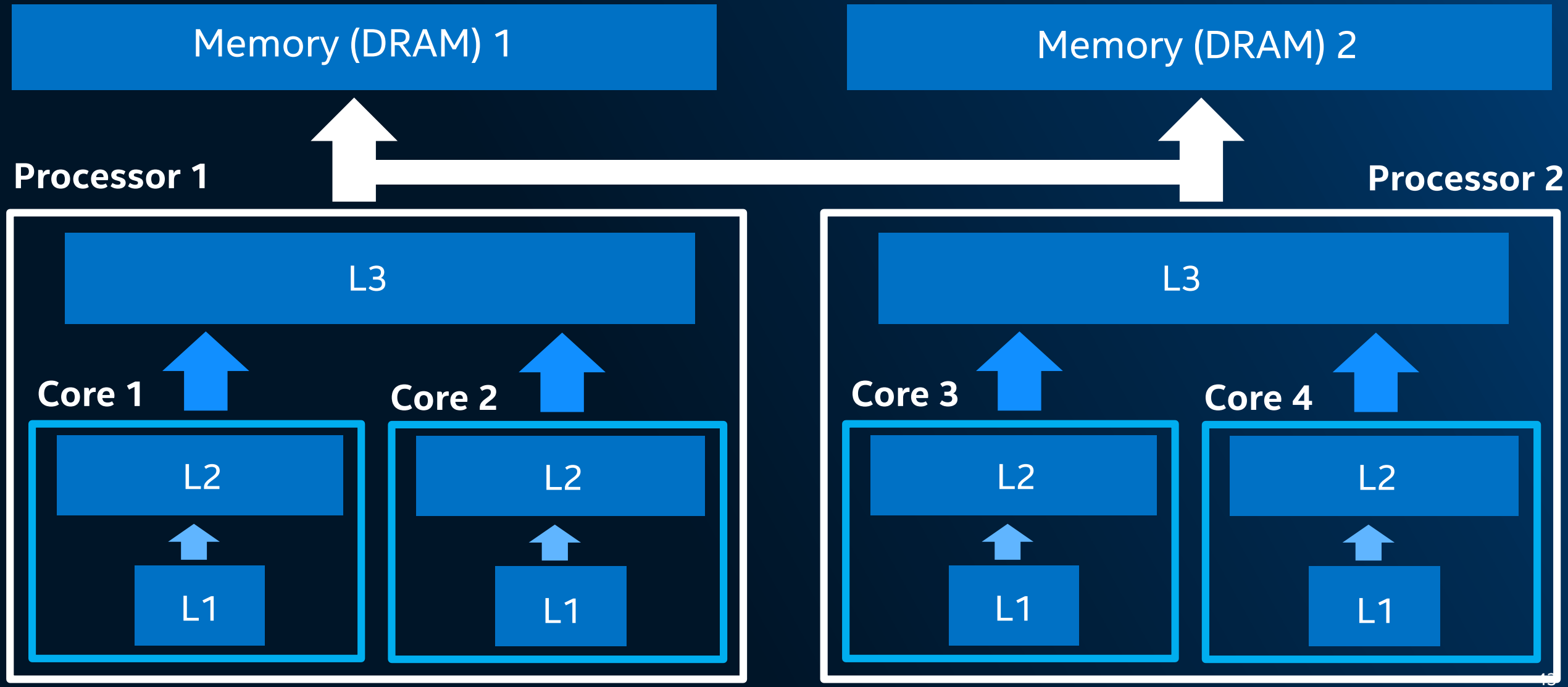
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

MICROARCHITECTURE EXPLORATION - FRAMES

```
__itt_domain* pD = __itt_domain_create(L"Enhance Frame");
__itt_frame_begin_v3(pD, NULL);
#pragma omp parallel for schedule(static)
for (int j = 0; j < COLUMNS; j++){
    for (int i = 0; i < LINES; i++) {
        Frame[i * COLUMNS + j] = Frame[i * COLUMNS + j] * Factor + Factor;
    }
}
__itt_frame_end_v3(pD, NULL);
```

Grouping: Frame Domain / Frame / Function / Call Stack

Frame Domain / Frame / Function / Call Stack	Back-End Bound									
	Memory Bound								Core Bound	Frame Time	Frame Count
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound				Store Bound			
				Memory Bandwidth	Memory Latency						
					Local DRAM	Remote DRAM	Remote Cache				
Enhance Frame	0.3%	0.4%	1.3%	68.2%	44.4%	67.8%	0.0%	0.0%	3.5%	8.059s	5
[No frame domain - Outside any frame]	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		


Memory (DRAM) 1
Memory (DRAM) 2
Processor 1
Processor 2
L3
L3
Core 1
Core 2
Core 3
Core 4
L2
L2
L2
L2
L1
L1
L1
L1

```
//init
float *Frame = (float*)malloc(LINES * COLUMNS * sizeof(float));
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++){
      Frame[i * COLUMNS + j] = 2 * i + 3 * j;
   }
}
//Process
#pragma omp parallel for schedule(static)
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      Frame[i * COLUMNS + j] = Fram
   }
}
```

DRAM Bound			
Memory Bandwidth	Memory Latency		
	Local DRAM	Remote DRAM	Remote Cache
68.2%	44.4%	67.8%	0.0%
0.0%	0.0%	0.0%	0.0%

```
float *Frame = \
                                                    at), \
/
#
f
}
```

DATA IS SPREAD ACROSS DRAMS!

HOW DO WE MAKE SURE THE DATA IS ON THE CLOSEST DRAM?

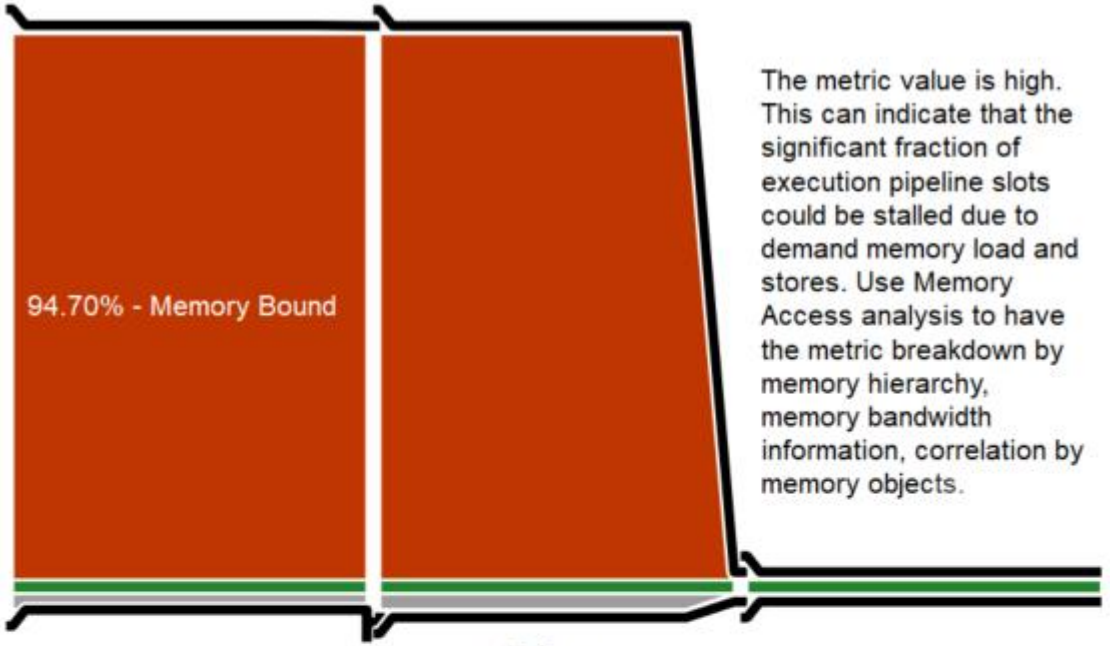
HOW DO WE MAKE SURE THE DATA IS ON THE CLOSEST DRAM?

- Pin thread affinity
  - OMP_PROC_BIND=close
  - OMP_PLACES=cores
- First touch access pattern should match the one used for processing

```
//init
#pragma omp parallel for schedule(static)
for (int j = COLUMNS - 1; j >= 0; j--){
    for (int i = 0; i < LINES; i++) {
        Frame[i * COLUMNS + j] = 2 * i + 3 * j;
    }
}
//Process
#pragma omp parallel for schedule(static)
for (int j = 0; j < COLUMNS; j++){
    for (int i = 0; i < LINES; i++) {
        Frame[i * COLUMNS + j] = Frame[i * COLUMNS + j] * Factor + Factor;
    }
}
```

MICROARCHITECTURE EXPLORATION

Clockticks:	1,698,137,500,000	
Instructions Retired:	165,400,000,000	
CPI Rate:	10.267	
MUX Reliability:	0.992	
Retiring:	2.4%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Back-End Bound:	97.4%	of Pipeline Slots
Memory Bound:	94.7%	of Pipeline Slots
L1 Bound:	0.3%	of Clockticks
DTLB Overhead:	23.0%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	1.0%	of Clockticks
4K Aliasing:	0.0%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.5%	of Clockticks
L3 Bound:	1.2%	of Clockticks
DRAM Bound:	82.9%	of Clockticks
Memory Bandwidth:	61.6%	of Clockticks
Memory Latency:	26.3%	of Clockticks
Local DRAM:	3.2%	of Clockticks
Remote DRAM:	100.0%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	2.7%	of Pipeline Slots
Total Thread Count:	56	
Paused Time:	15.534s	
Frame Count:	5	

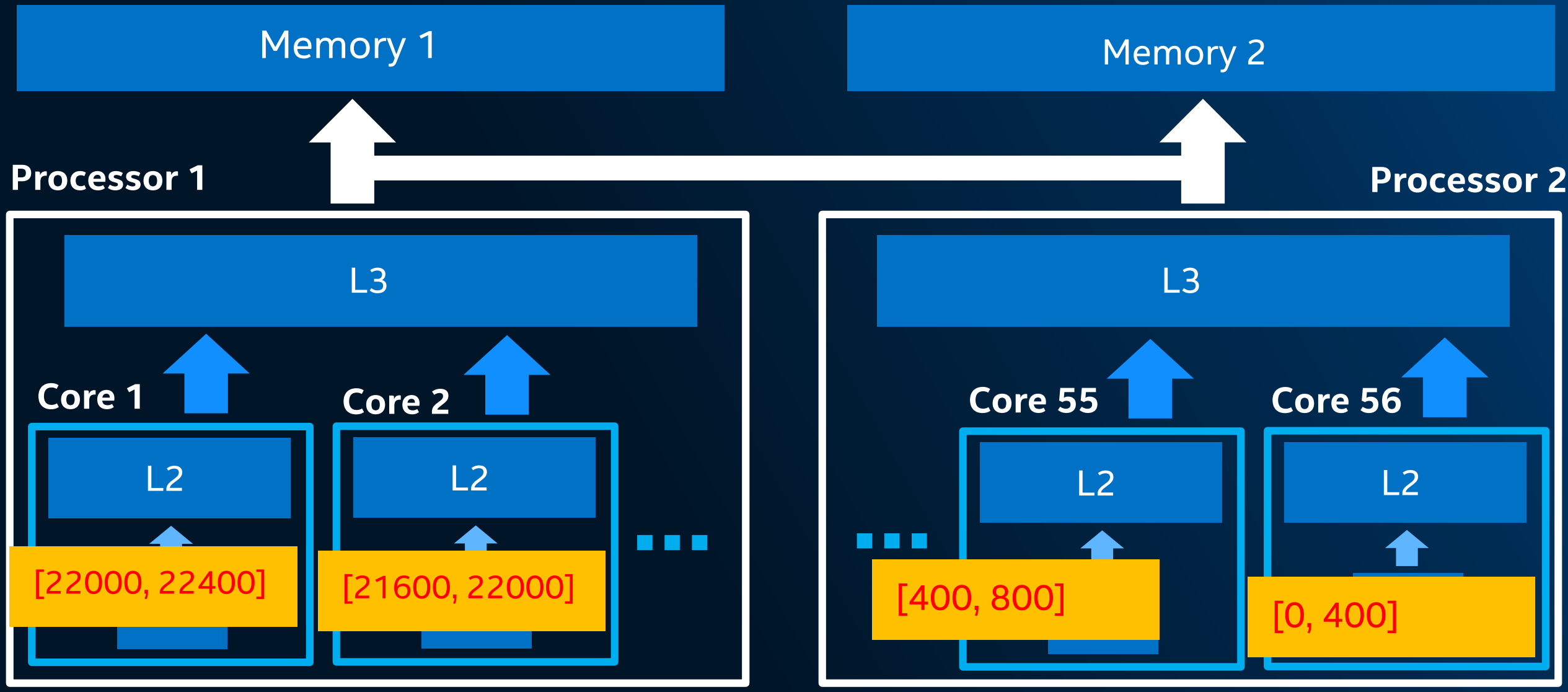


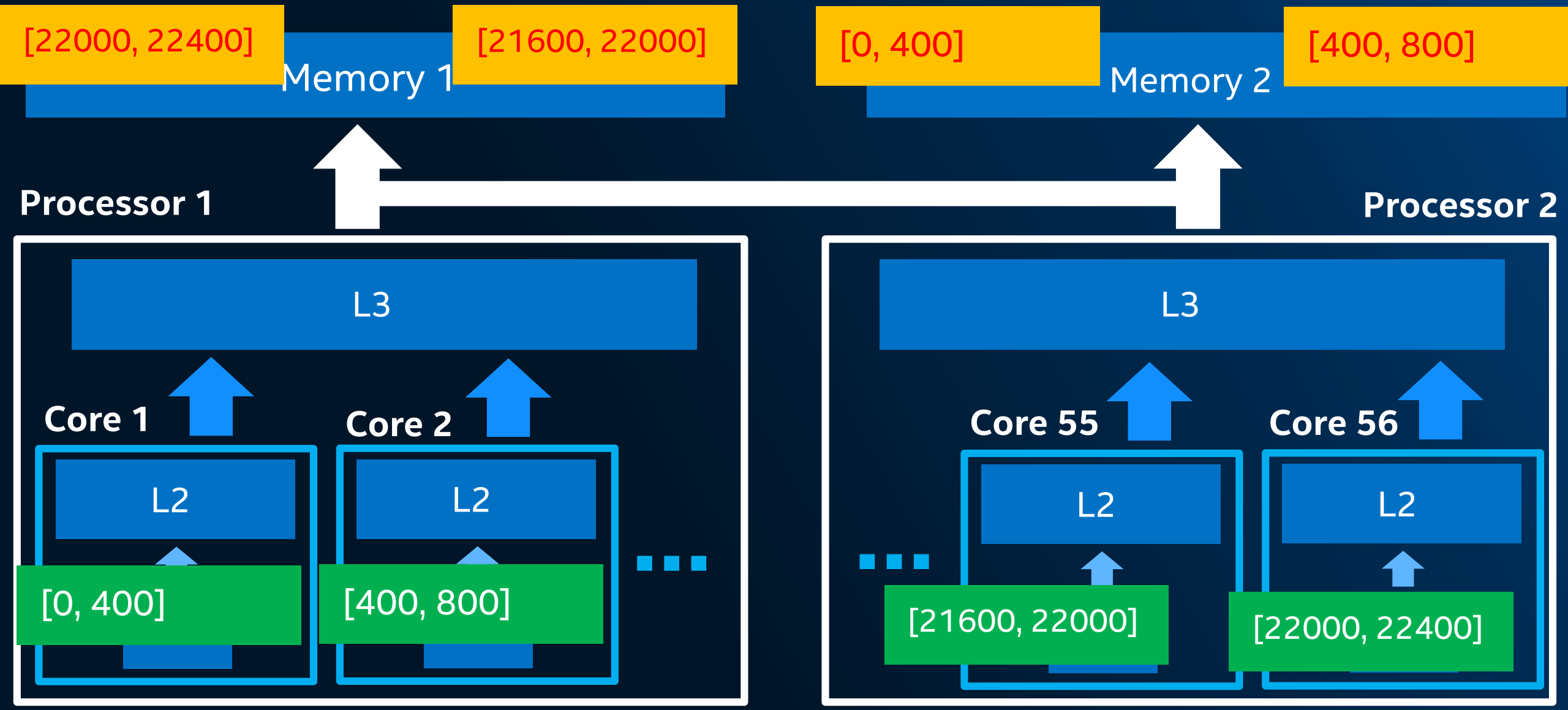
The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

COLUMNS = 22400, N_THREADS = 56 → 400 iterations per core

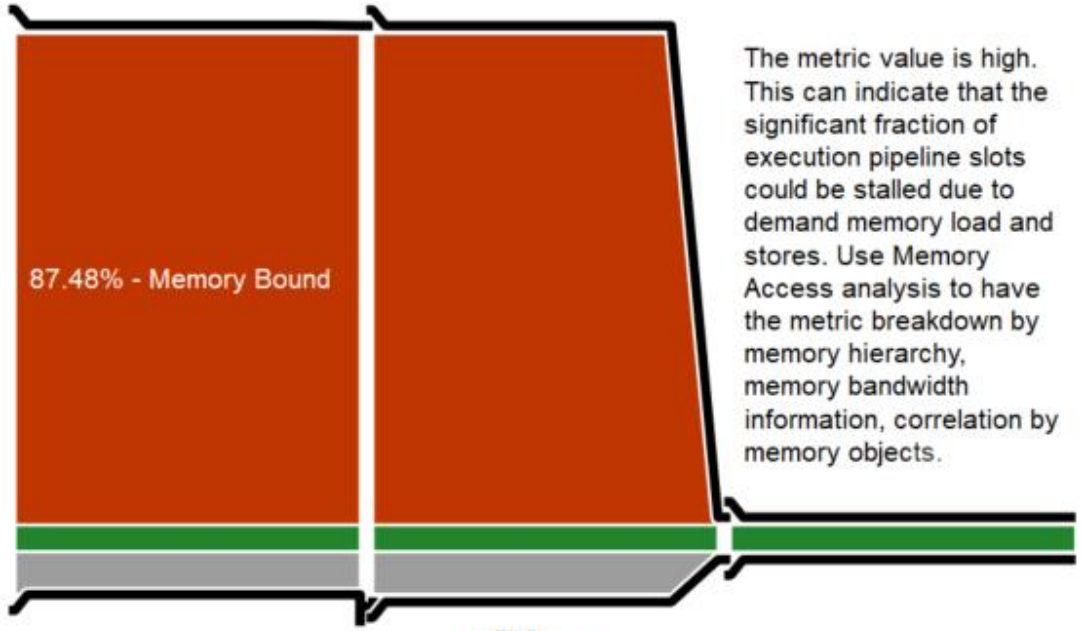



[22000, 22400]
[21600, 22000]
Memory 1
[0, 400]
[400, 800]
Memory 2
Processor 1
Processor 2
L3
L3
Core 1
Core 2
Core 55
Core 56
L2
L2
L2
L2
[0, 400]
[400, 800]
[21600, 22000]
[22000, 22400]

MICROARCHITECTURE EXPLORATION

Elapsed Time: 20.670s

Clockticks:	730,500,000,000	
Instructions Retired:	147,737,500,000	
CPI Rate:	4.945	
MUX Reliability:	0.997	
Retiring:	4.8%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Back-End Bound:	95.0%	of Pipeline Slots
Memory Bound:	87.5%	of Pipeline Slots
L1 Bound:	0.4%	of Clockticks
DTLB Overhead:	48.9%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.1%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.6%	of Clockticks
L3 Bound:	2.1%	of Clockticks
DRAM Bound:	70.4%	of Clockticks
Memory Bandwidth:	64.3%	of Clockticks
Memory Latency:	17.5%	of Clockticks
Local DRAM:	100.0%	of Clockticks
Remote DRAM:	6.1%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	7.5%	of Pipeline Slots
Total Thread Count:	56	
Paused Time:	16.368s	
Frame Count:	5	



The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

```
//Process
#pragma omp parallel for schedule(static)
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
       Frame[i * COLUMNS + j] = \
        Frame[i * COLUMNS + j] * Factor + Factor;
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Frame

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1																
2																

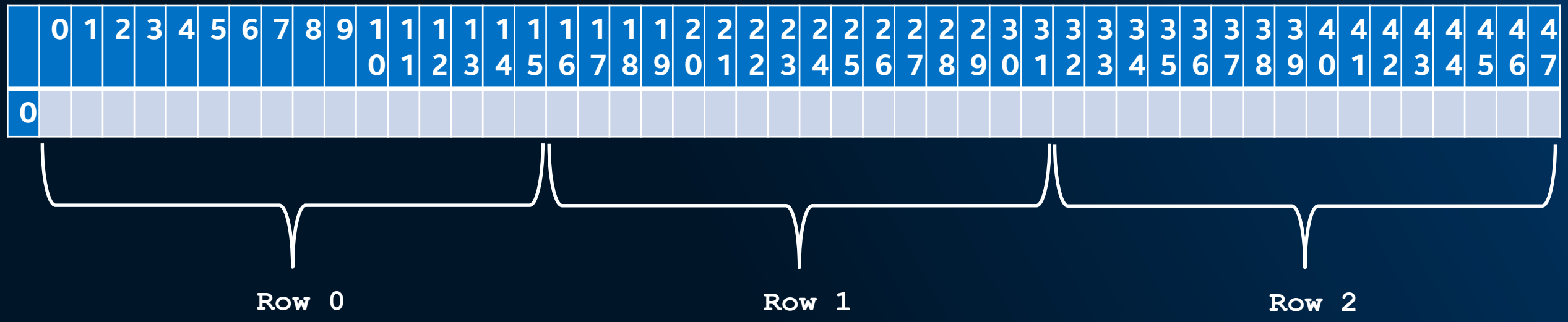
Frame in memory

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0																																																		

```
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      …Frame[i * COLUMNS + j]…
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Frame in memory

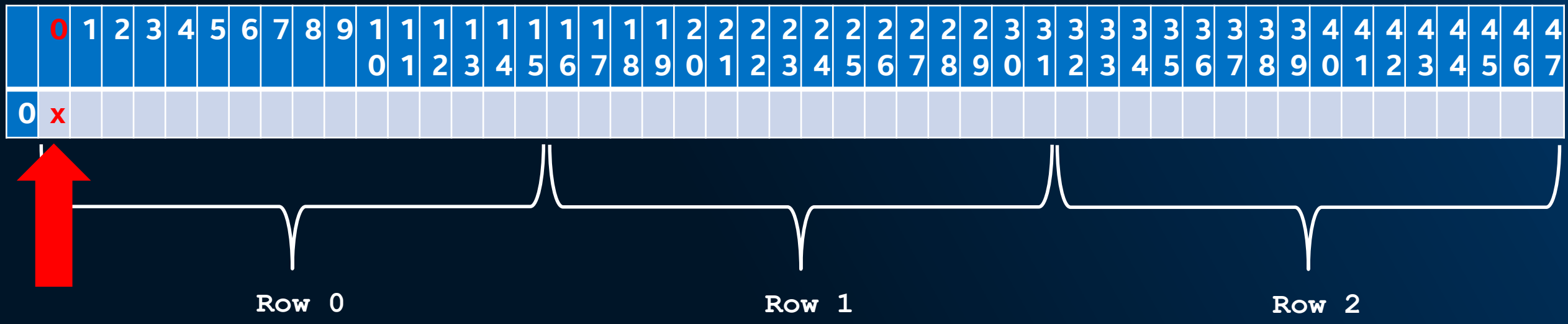


```
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      …Frame[i * COLUMNS + j]…
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Iteration 0 → Access element 0

Frame in memory

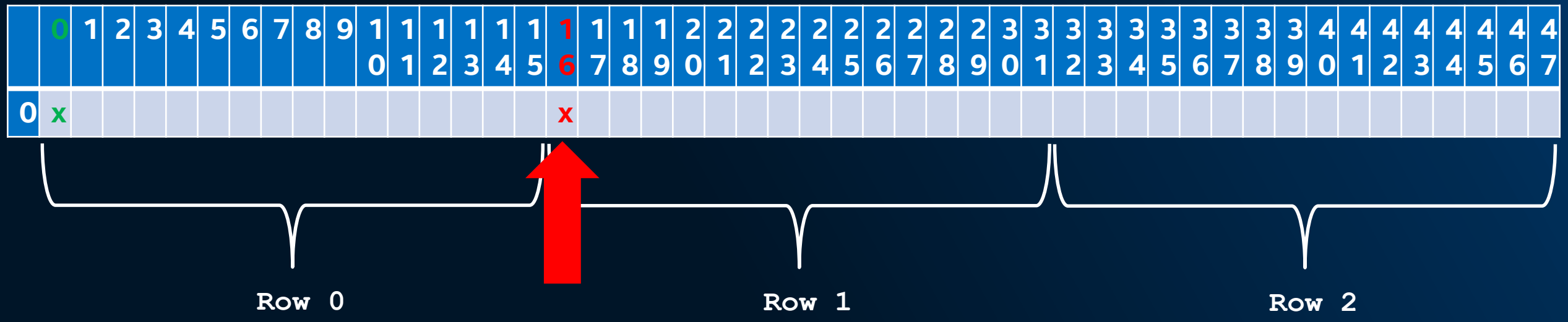


```
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      …Frame[i * COLUMNS + j]…
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Iteration 1 → Access element 16

Frame in memory



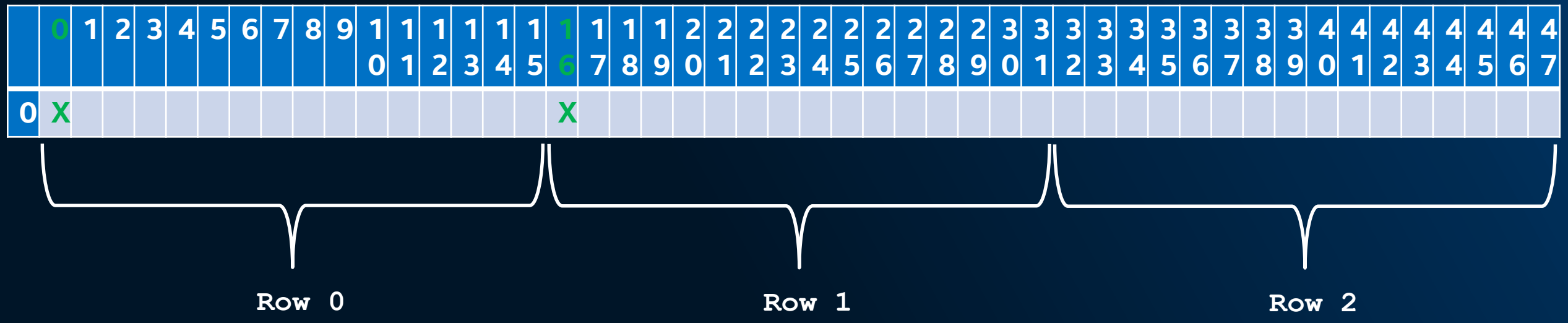
```
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      …Frame[i * COLUMNS + j]…
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Float is 4 bytes, cache line is 64 bytes

16 floats in 1 cache line

Frame in memory

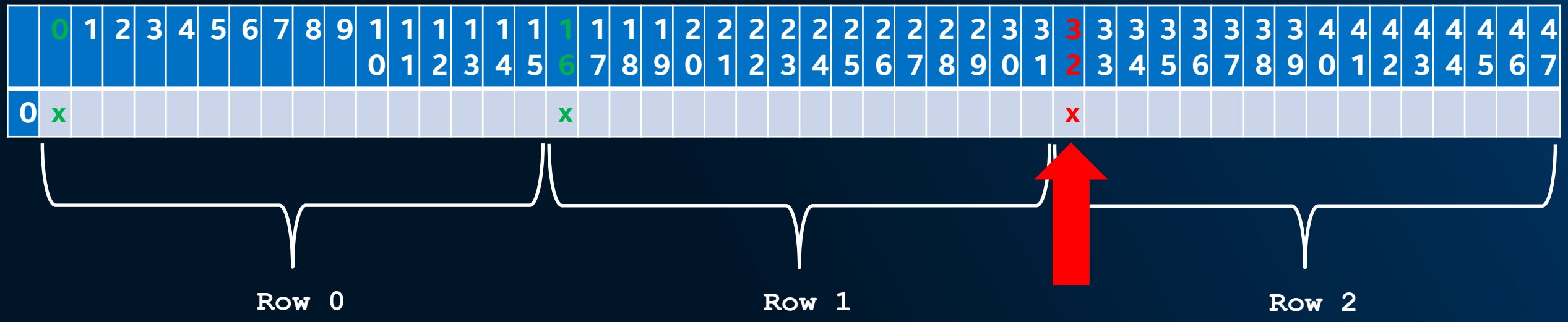


```
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      …Frame[i * COLUMNS + j]…
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Iteration 2 → Access element 32

Frame in memory

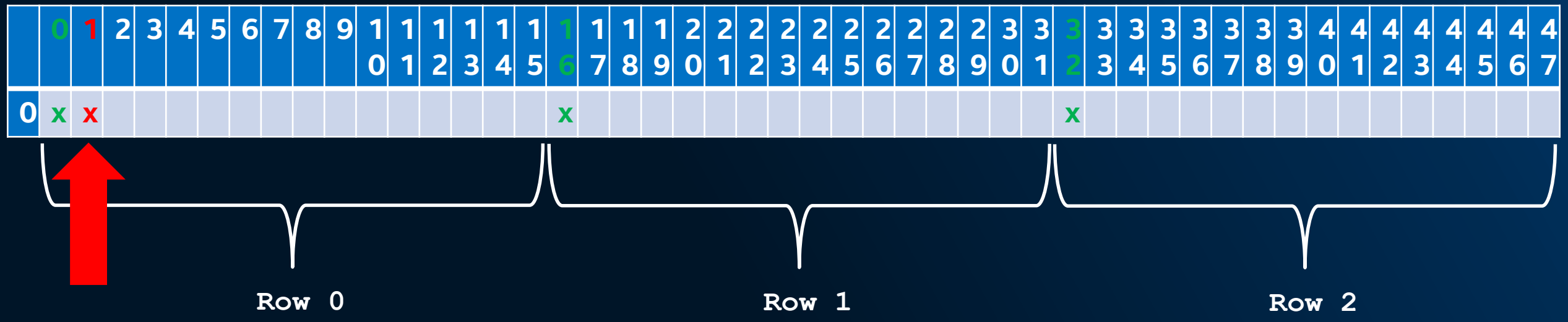


```
for (int j = 0; j < COLUMNS; j++){
   for (int i = 0; i < LINES; i++) {
      …Frame[i * COLUMNS + j]…
   }
}
```

```
#define COLUMNS 16
#define LINES 3
```

Iteration 3 → Access element 1

Frame in memory

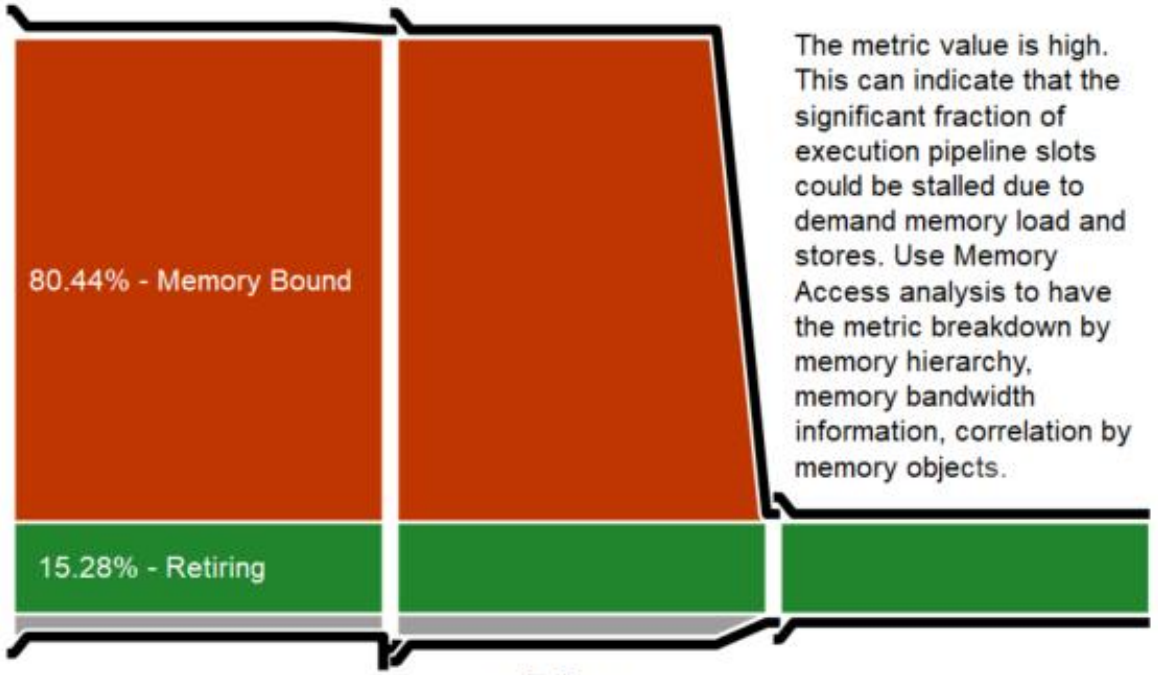


```
//init
for (int j = 0; j < COLUMNS; j++){
    for (int i = 0; i < LINES; i++){
        Frame[i * COLUMNS + j] = 2 * i + 3 * j;
    }
}
//Process
#pragma omp parallel for schedule(static)
for (int j = 0; j < COLUMNS; j++){
    for (int i = 0; i < LINES; i++){
            Frame[i * COLUMNS + j] = Frame[i * COLUMNS + j] * Factor + Factor;

    }
}
```

MICROARCHITECTURE EXPLORATION

Clockticks:	2,443,975,000,000	
Instructions Retired:	1,600,700,000,000	
CPI Rate:	1.527	
MUX Reliability:	0.995	
Retiring:	15.3%	of Pipeline Slots
Front-End Bound:	0.5%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Back-End Bound:	84.1%	of Pipeline Slots
Memory Bound:	80.4%	of Pipeline Slots
		of Clockticks
		of Clockticks
L3 Bound:	2.8%	of Clockticks
DRAM Bound:	30.7%	of Clockticks
Memory Bandwidth:	70.9%	of Clockticks
Memory Latency:	5.7%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	3.7%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	2.7%	of Clockticks
Cycles of 0 Ports Utilized:	0.0%	of Clockticks
Cycles of 1 Port Utilized:	2.7%	of Clockticks
Cycles of 2 Ports Utilized:	4.0%	of Clockticks
Cycles of 3+ Ports Utilized:	17.8%	of Clockticks
Vector Capacity Usage (FPU):	6.3%	
Total Thread Count:	56	
Paused Time:	2.889s	
Frame Count:	300	

Down from 70-80%



The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

INTEL COMPILER REPORT

```
#pragma omp parallel for  schedule(static)
```

1 optimization note

Main loop

15529: loop was not vectorized: volatile assignment was not vectorized. Try using non-volatile assignment. (line: 34, column: 3)

```
    for (int i = 0; i < LINES; i++) {
        __itt_task_begin(omp__domain, __itt_null, __itt_null, omp__task);
```

1 optimization note

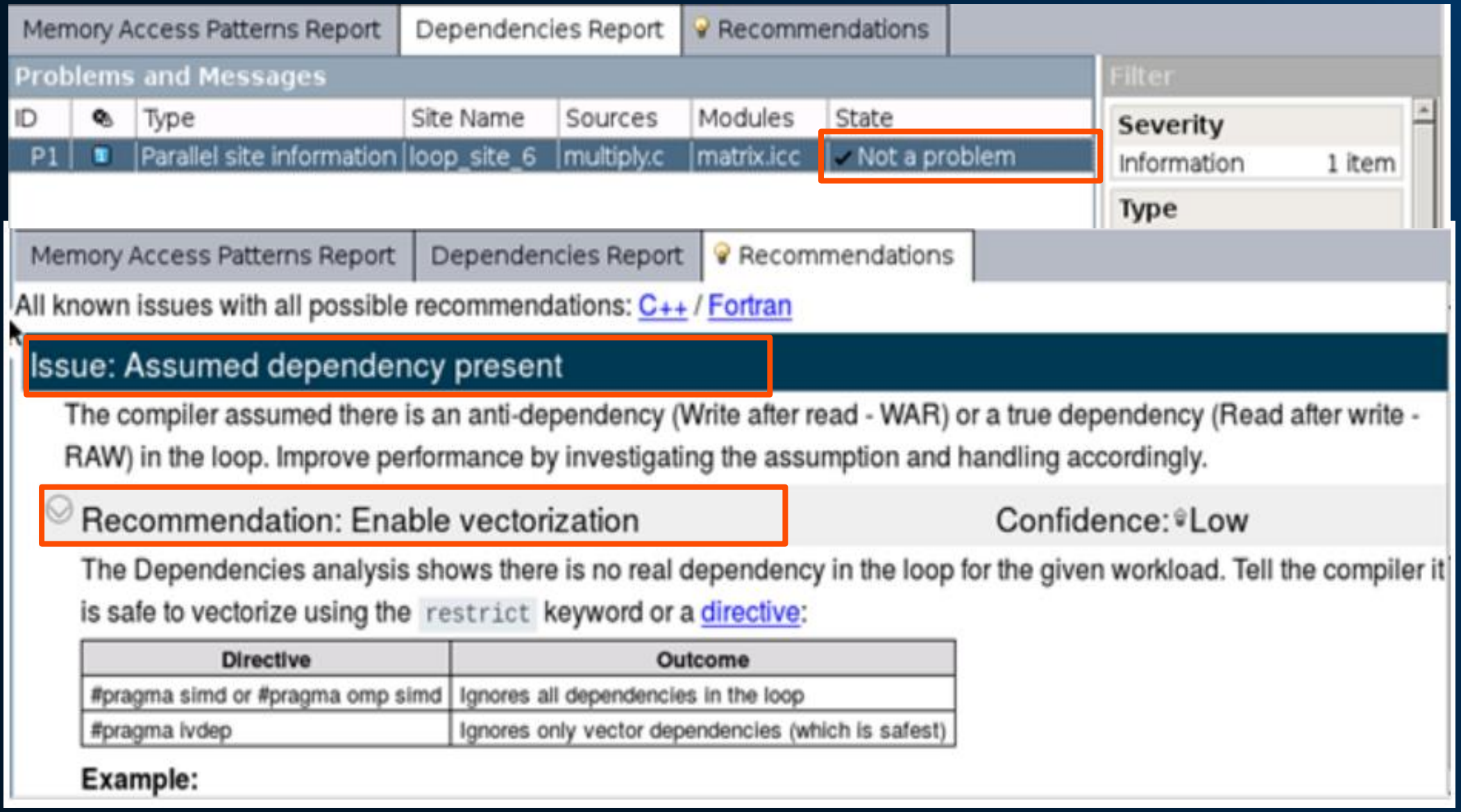
Main loop

15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details (line: 35, column: 3)

15346: vector dependence: assumed FLOW dependence between Frame[i*22400+j] (36:4) and Factor[i*22400+j] (36:4) (line: 35, column: 3)

```
        for (int j = 0; j < COLUMNS; j++) {
            Frame[i * COLUMNS + j] = Frame[i * COLUMNS + j] * Factor[i * COLUMNS + j] + FactorScalar;
        }
        __itt_task_end(omp__domain);
    }

}
```

INTEL ADVISOR REPORT



INTEL COMPILER REPORT

```
//#pragma intel optimization_parameter target_arch=AVX2
#pragma intel optimization_parameter target_arch=COMMON-AVX512
__declspec(noinline) void ProcFrameCase5(float*Frame, float*Factor)
{
#pragma omp parallel for  schedule(static)
```

1 optimization note

Main loop

```
    for (int i = 0; i < LINES; i++) {
        __itt_task_begin(omp__domain, __itt_null, __itt_null, omp__task);
        //ivdep: Instructs the compiler to ignore assumed vector dependencies
#pragma ivdep
```

3 optimization notes

Peeled loop for vectorization

Main loop

15300: LOOP WAS VECTORIZED (line: 38, column: 3)

15442: entire loop may be executed in remainder (line: 38, column: 3)

15448: unmasked aligned unit stride loads: 1 (line: 38, column: 3)

15449: unmasked aligned unit stride stores: 1 (line: 38, column: 3)

15450: unmasked unaligned unit stride loads: 1 (line: 38, column: 3)

15475: --- begin vector cost summary --- (line: 38, column: 3)

15476: scalar cost: 9 (line: 38, column: 3)

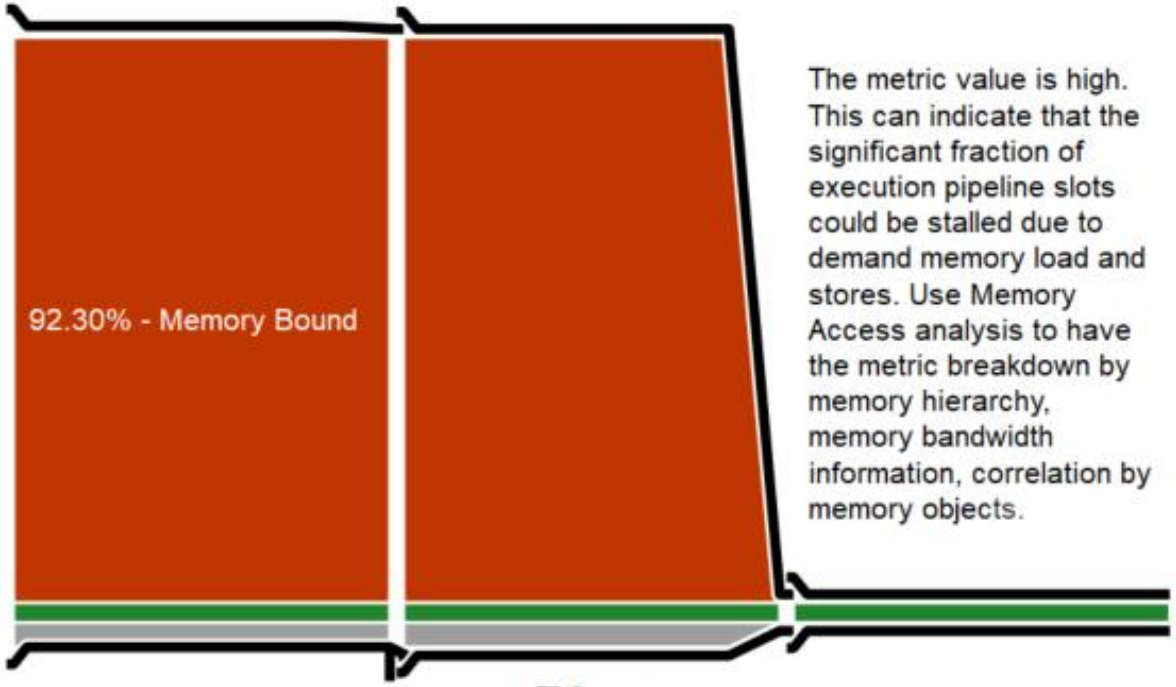
15477: vector cost: 0.560 (line: 38, column: 3)

15478: estimated potential speedup: 15.910 (line: 38, column: 3)

```
        for (int j = 0; j < COLUMNS; j++) {
            Frame[i * COLUMNS + j] = Frame[i * COLUMNS + j] * Factor[i * COLUMNS + j] + FactorScalar;
        }
        __itt_task_end(omp__domain);
    }
}
```

MICROARCHITECTURE EXPLORATION

Clockticks:	1,964,087,500,000	
Instructions Retired:	238,575,000,000	
CPI Rate:	8.233	
MUX Reliability:	0.994	
Retiring:	3.2%	of Pipeline Slots
Front-End Bound:	0.5%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	96.3%	of Pipeline Slots
Memory Bound:	92.3%	of Pipeline Slots
L1 Bound:	0.8%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	1.4%	of Clockticks
DRAM Bound:	69.0%	of Clockticks
Memory Bandwidth:	67.1%	of Clockticks
Memory Latency:	5.6%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	4.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	3.1%	of Clockticks
Cycles of 0 Ports Utilized:	0.0%	of Clockticks
Cycles of 1 Port Utilized:	3.1%	of Clockticks
Cycles of 2 Ports Utilized:	1.8%	of Clockticks
Cycles of 3+ Ports Utilized:	1.6%	of Clockticks
Vector Capacity Usage (FPU):	100.0%	
Total Thread Count:	56	
Paused Time:	2.753s	
Frame Count:	300	



The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

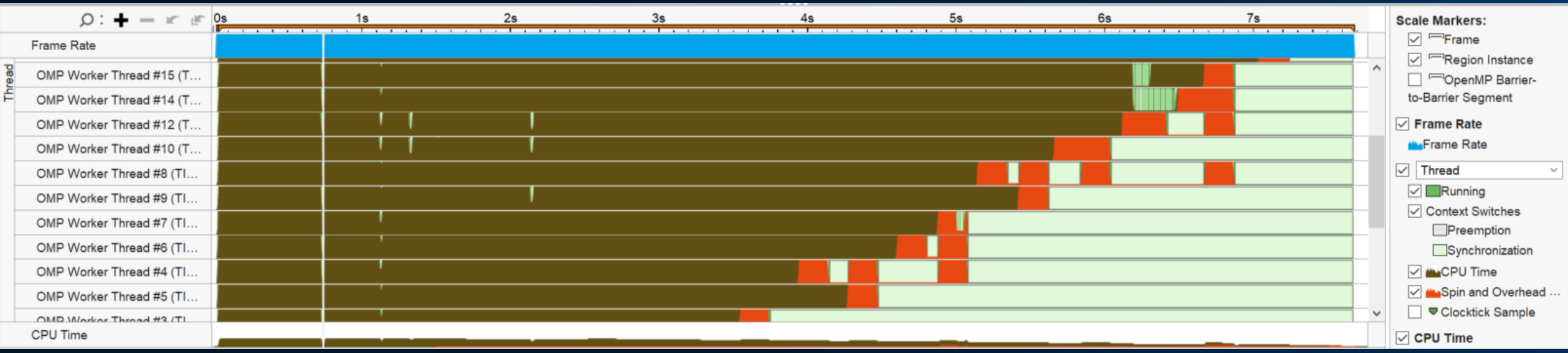
```
#pragma omp parallel for reduction (+:sum)
for (int i = 2; i <= NUM; i++){
   sum += isPrime(i);
}
```

THREADING ANALYSIS

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time ▼	Serial CPU Time	OpenMP Potential Gain					
			Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics
▶ proc1ompparallel:22@unknown:25:28	7.622s	0s	2.231s	0s	0s	0s	0s	0s



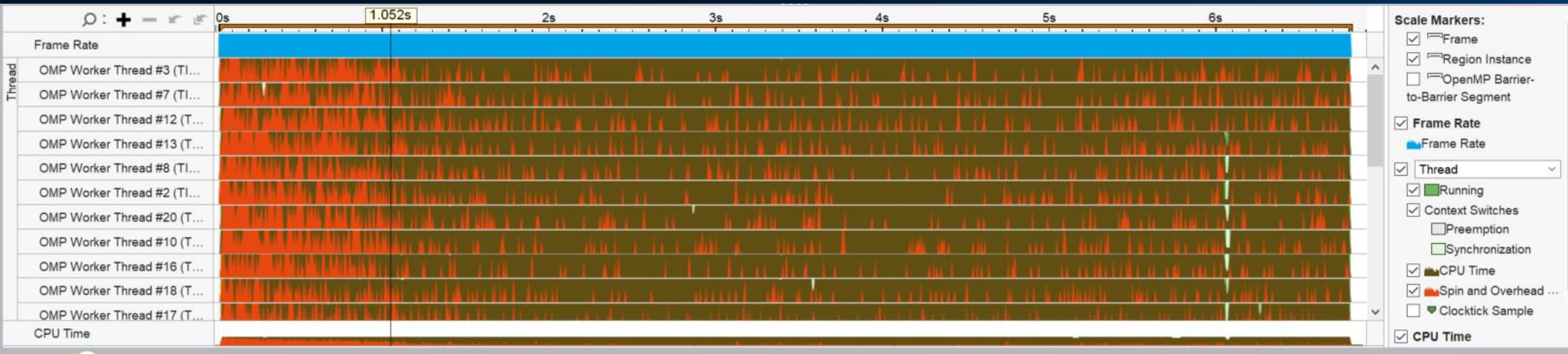
```
#pragma omp parallel for reduction (+:sum)
schedule(dynamic)
for (int i = 2; i <= NUM; i++){
   sum += isPrime(i);
}
```

THREADING ANALYSIS

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: OpenMP Region / OpenMP **13% Boost** nction / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time ▼	Serial CPU Time	OpenMP Potential Gain					
			Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics
proc2ompparallel:22@unknown:39:42	6.775s	0s	0.000s	0s	0s	1.291s	0s	0s



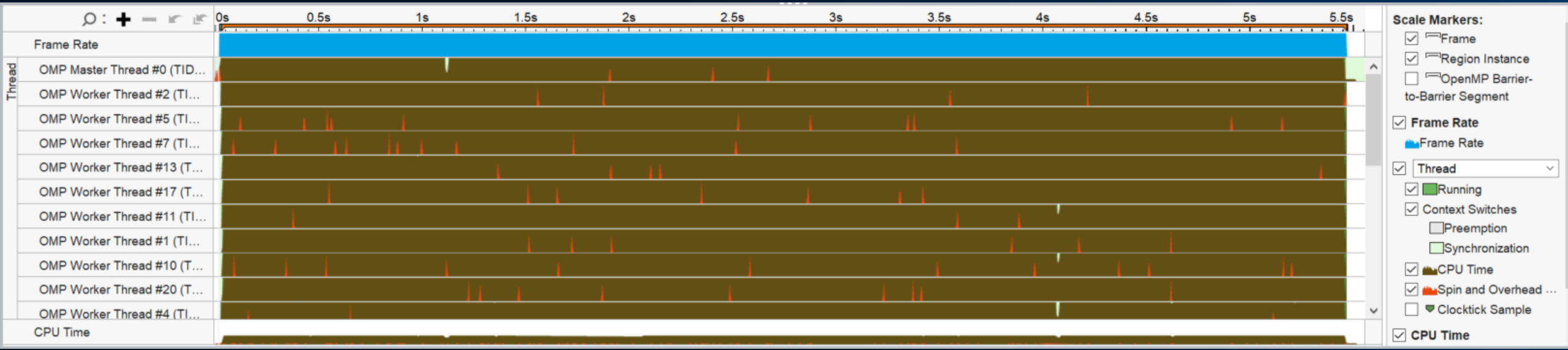
```
#pragma omp parallel for reduction (+:sum)
schedule(dynamic, 20)
for (int i = 2; i <= NUM; i++){
   sum += isPrime(i);
}
```

THREADING ANALYSIS

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

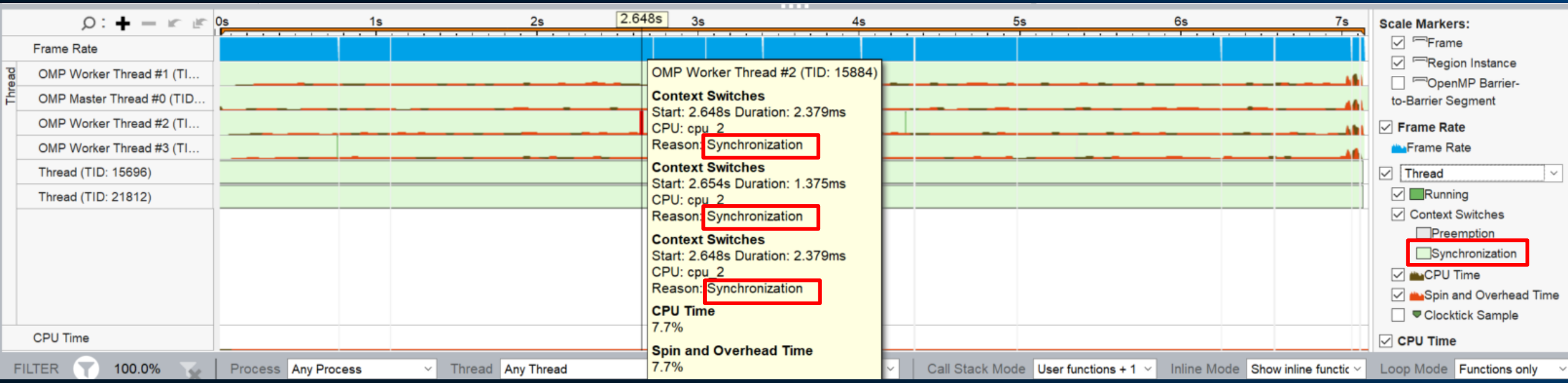
Grouping: OpenMP Region / **40% Boost (+24%)** Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time ▼	Serial CPU Time	OpenMP Potential Gain					
			Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics
proc3ompparallel:22@unknown:53:56	5.430s	0s	0.000s	0s	0.000s	0.033s	0s	0s



```
#pragma omp parallel for reduction (+:sum)
schedule(dynamic, 20)
for (int i = 2; i <= NUM; i++){
   sum += isPrime(i);
   printf("Interesting Info %d\n", id);
}
```

THREADING ANALYSIS

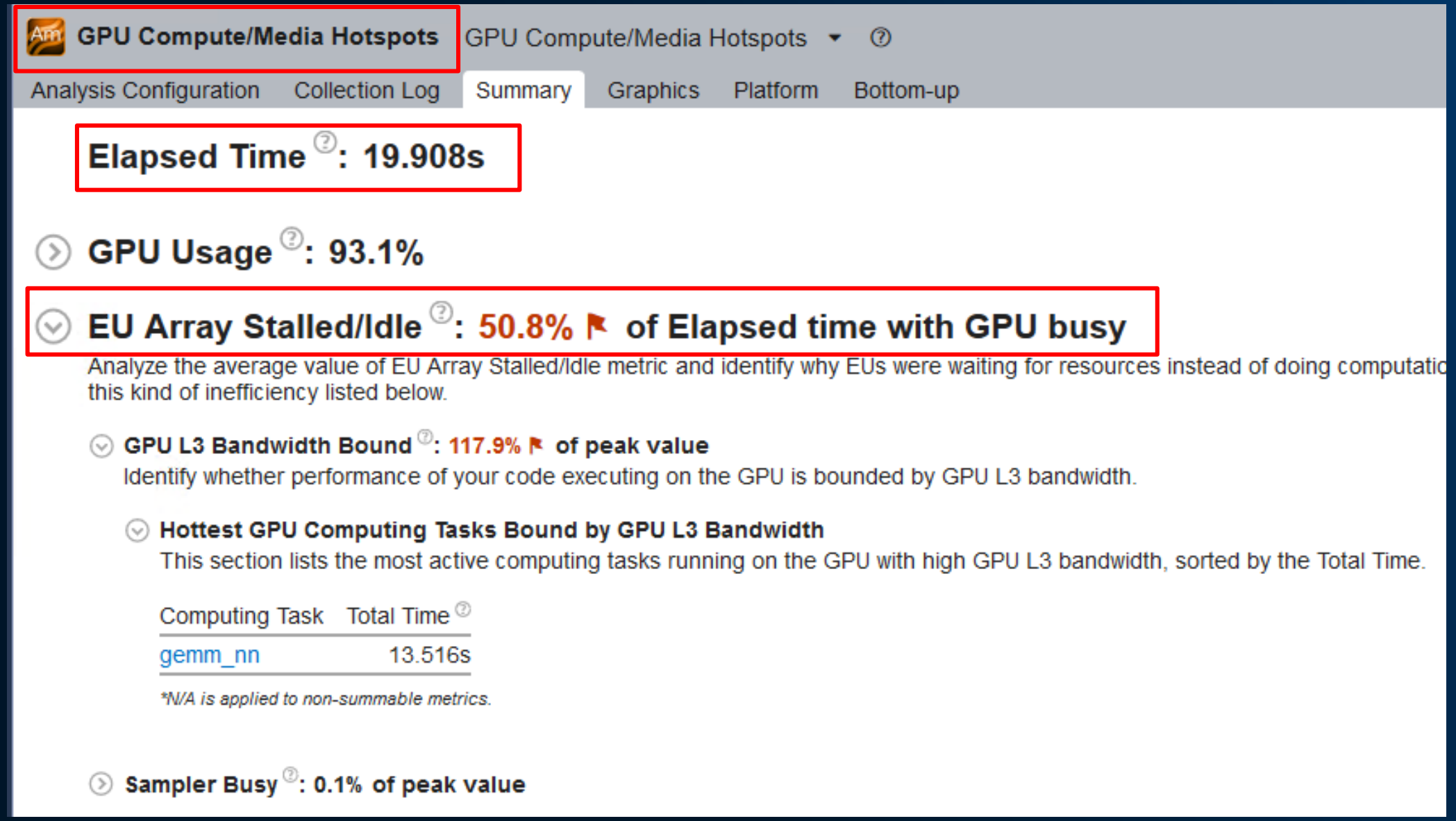


Analysis Configuration | Collection Log | Summary | **Bottom-up** | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

Function / Call Stack	Serial CPU Time ▼	CPU Time			Inactive
		Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Overhead Time	Inactive Sync Wait Time
_stdio_common_vfprintf	0s	0.669s	1.878s	0s	24.754
func@0x18008ae4e	0s	0s	0s	0s	0.000
NtDeviceIoControlFile	0s	0s	0s	0s	0.005
exit	0s	0s	0s	0s	0.027

GPU COMPUTE / MEDIA HOTSPOTS



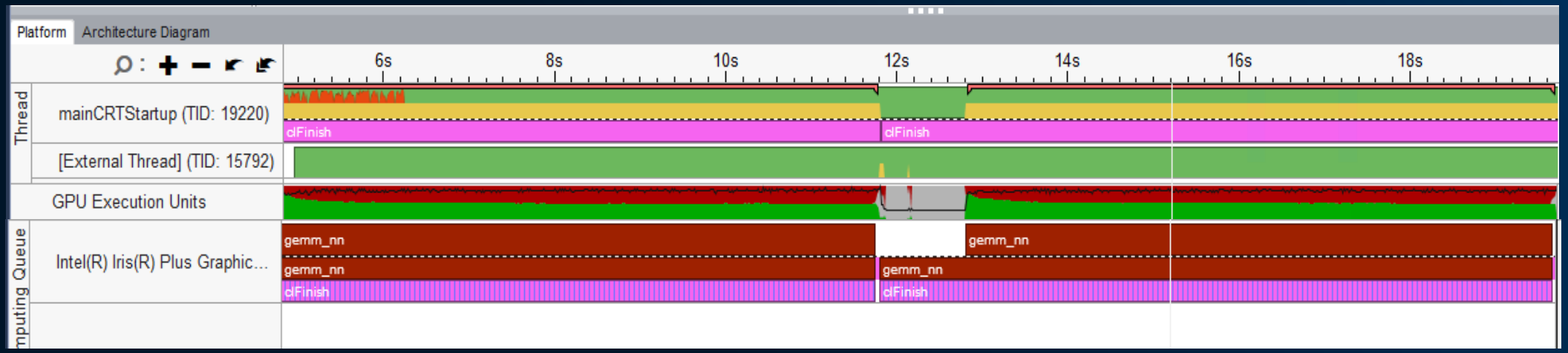
GPU COMPUTE / MEDIA HOTSPOTS

GPU Compute/Media Hotspots GPU Compute/Media Hotspots

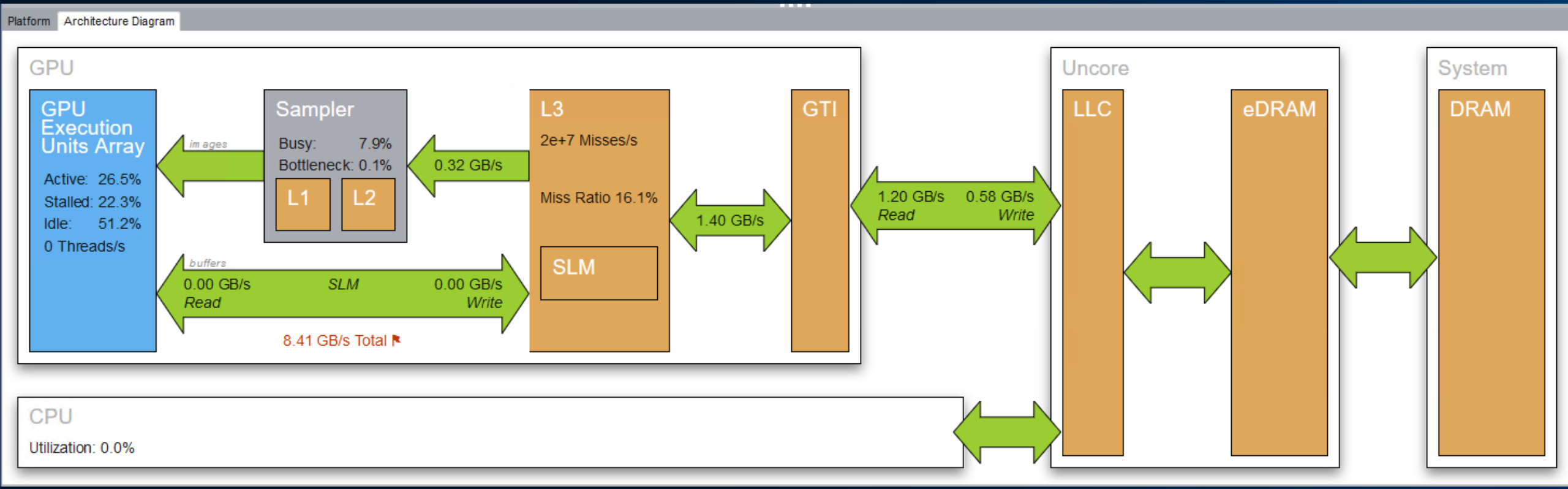
Analysis Configuration | Collection Log | Summary | Graphics | Platform | Bottom-up

Grouping: Computing Task / Instance

Computing Task / Instance	Work Size		Computing Task					Data Transferred		EU Array		
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width	SVM U...	Size	Total, GB/sec	Active	Stalled	Idle
gemm_nn	3968 x 31	8 x 1	13.516s	6.758s	2	8			0.000	49.4%	50.6%	0.0%



GPU COMPUTE / MEDIA HOTSPOTS



```
size_t input_size[2] = {3968, 3968};
size_t block_size[2] = {1, 128};
size_t global_size[2] = {
    input_size[0] / block_size[0],
    input_size[1] / block_size[1]
};
size_t local_size[2] = {8, 1};
err = clEnqueueNDRangeKernel(
            oclobjects.queue, executable.kernel, 2,0,
            global_size,
            local_size,
            0, 0, 0);
```

```
size_t input_size[2] = {3968, 3968};
size_t global_size[2] = {3968, 31};
size_t local_size[2] = {16, 1};
err = clEnqueueNDRangeKernel(
            oclobjects.queue, executable.kernel,
            2,0,
            global_size,
            local_size,
            0, 0, 0
        );
```

GPU COMPUTE / MEDIA HOTSPOTS

20% Boost

Elapsed Time: 16.517s

GPU Usage: 90.4%

15% Boost

EU Array Stalled/Idle: 43.9% of Elapsed time with GPU busy

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computati
this kind of inefficiency listed below.

GPU L3 Bandwidth Bound: 165.4% of peak value

Identify whether performance of your code executing on the GPU is bounded by GPU L3 bandwidth.

Hottest GPU Computing Tasks Bound by GPU L3 Bandwidth

This section lists the most active computing tasks running on the GPU with high GPU L3 bandwidth, sorted by the Total Time.

Computing Task	Total Time
gemm_nn	10.121s

Analysis Configuration | Collection Log | Summary | Graphics | Platform | Bottom-up

Grouping: Computing Task / Instance

34% Boost

Computing Task / Instance	Work Size		Computing Task					Data Transferred		EU Array		
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width	SVM U...	Size	Total, GB/sec	Active	Stalled	Idle
gemm_nn	3968 x 31	16 x 1	10.121s	5.060s	2	16			0.000	56.6%	42.7%	0.7%

```
for (int iter = 1; iter < N_ITER; iter++) {
    double**matrix = (double**)malloc(N*sizeof(double*));
    for (int i = 0; i < N; i++) {
        matrix[i] = (double*)malloc(N*sizeof(double));
        for (int j = 0; j < N; j++) {
            matrix[i][j] = (i * 0.2 + j);
        }
    }
    double result = process(matrix);
    printf("Result = %d\n", result);
}
```

DEMO

```
for (int iter = 1; iter < N_ITER; iter++) {
    double**matrix = (double**)malloc(N*sizeof(double*));
    for (int i = 0; i < N; i++) {
        matrix[i] = (double*)malloc(N*sizeof(double));
        for (int j = 0; j < N; j++) {
            matrix[i][j] = (i * 0.2 + j);
        }
    }
    double result = process(matrix);
    printf("Result = %d\n", result);
    free_mat(matrix);
}
```

DEMO

NEXT TIME YOU LOOK AT YOUR CODE, MAKE SURE THAT -

- You are not doing extra work
  - Unless there's a good reason
- Your memory is always close by
  - And you don't have to share
- You are using vectorized instructions
- Your threads are busy working

Developer: Call to Action/More Resources

Download Intel® System Studio

- Free Community License
- Free/Discounted Versions for Students /Academia

Access Developer Resources

- Product site
- Get Started/Documentation
- Code Samples/Training
- Tech.Decoded webinars, how-to videos/articles
- Expert Community Support Forum

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Paid Licenses include Priority Support

Software

THANK YOU!
STAY IN TOUCH

Shiran.Afergan@intel.com

Legal Disclaimer & Optimization Notice

Performance results are based on testing as of August 2017 to September 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

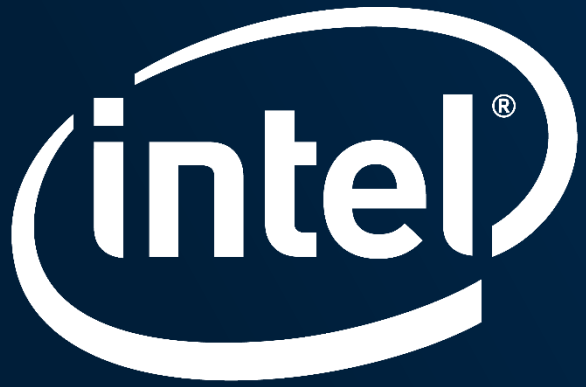
INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2018, Intel Corporation. All rights reserved. Intel, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Software