GDC 11 Europe

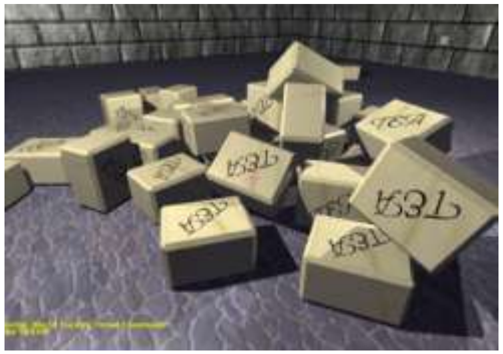
Game Developers Conference™ Europe 2011
August 15-17, 2011 | Cologne, Germany
www.GDCEurope.com

Hotspots, FLOPS, and uOps: To-The-Metal CPU Optimization

Levent Akyil
Intel Corp.

Motivation:

Fast Code == More Stuff == More Fun



Before

After

Source Code Does Not Provide The Complete Story

Example Optimization Tips:

"Use the return value optimization"

"Pass by reference instead of value"

"Use ++i instead of i++"

"cache intermediate computations"

"Unroll loops"

Has this guy studied your code?

Without context, Ad hoc source code tips may result in random trial and error.

This Class Goes Beyond the Source Code

Agenda:

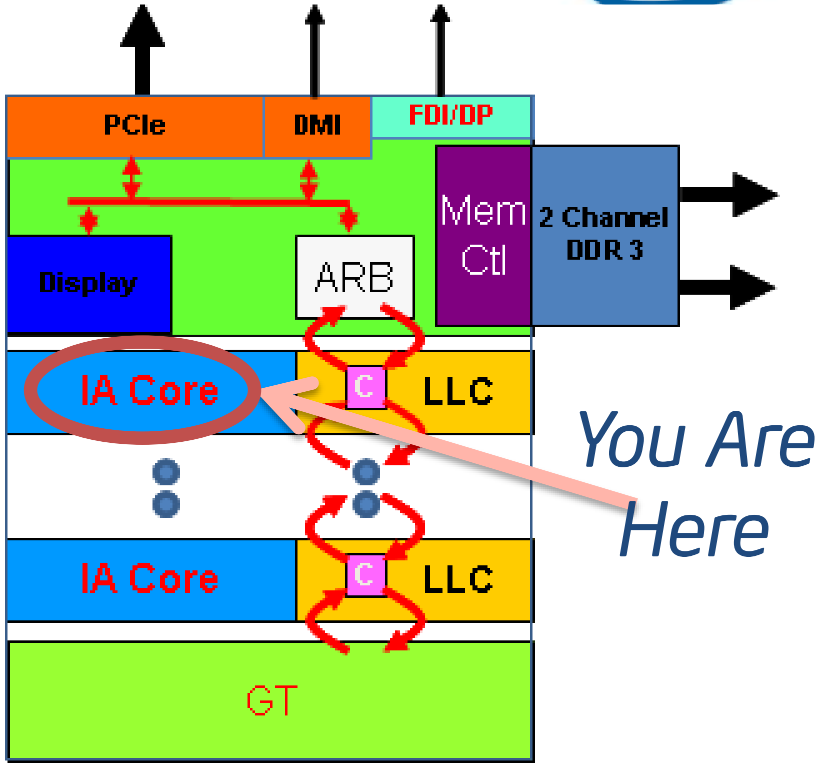
x86 Architecture

- Program Execution Flow
- AVX® SIMD
- Easy Effective code patterns

Performance Tuning Workflow

- Hotspot, Concurrency profiling
- Events and Intel® VTune™ Amplifier XE

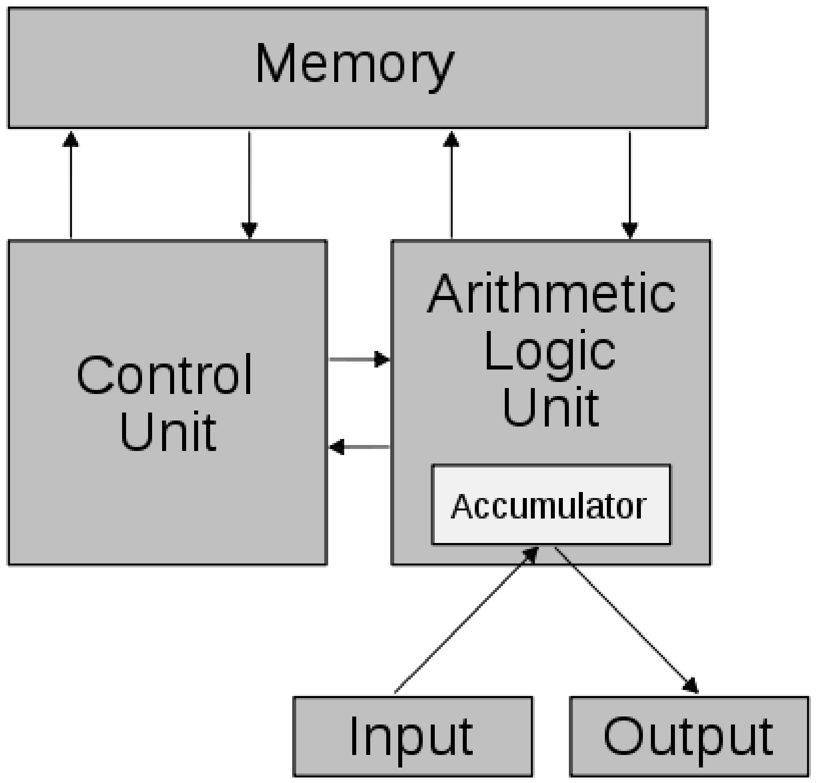
Walkthrough/Examples



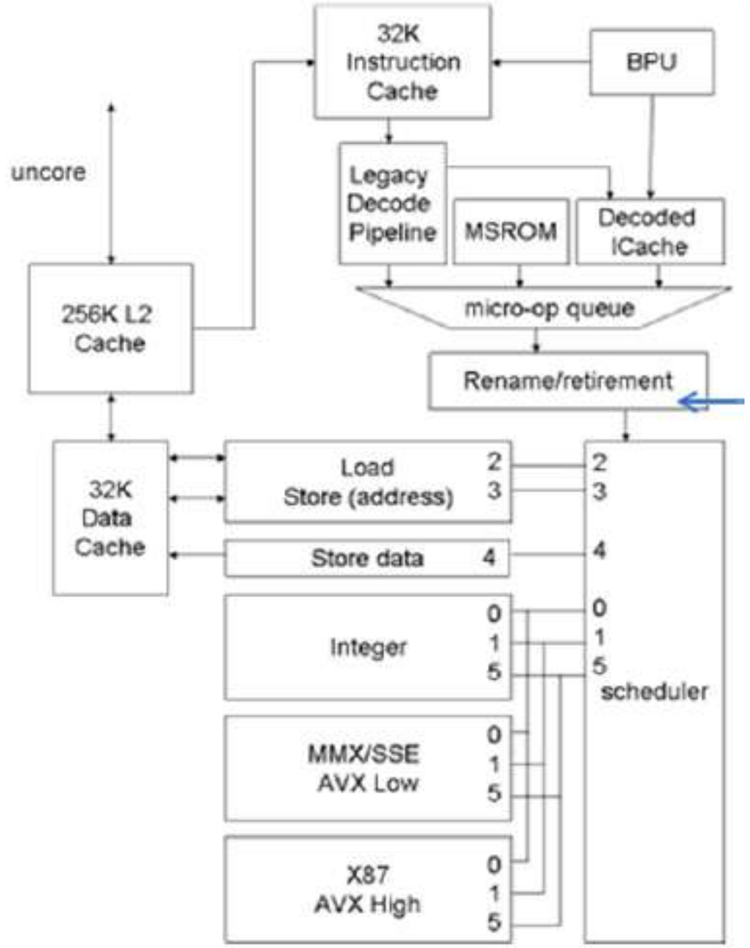
Intel® Microarchitecture
Codename SandyBridge

Take the Guesswork out of Optimization!

This is not your Grandfather's CPU



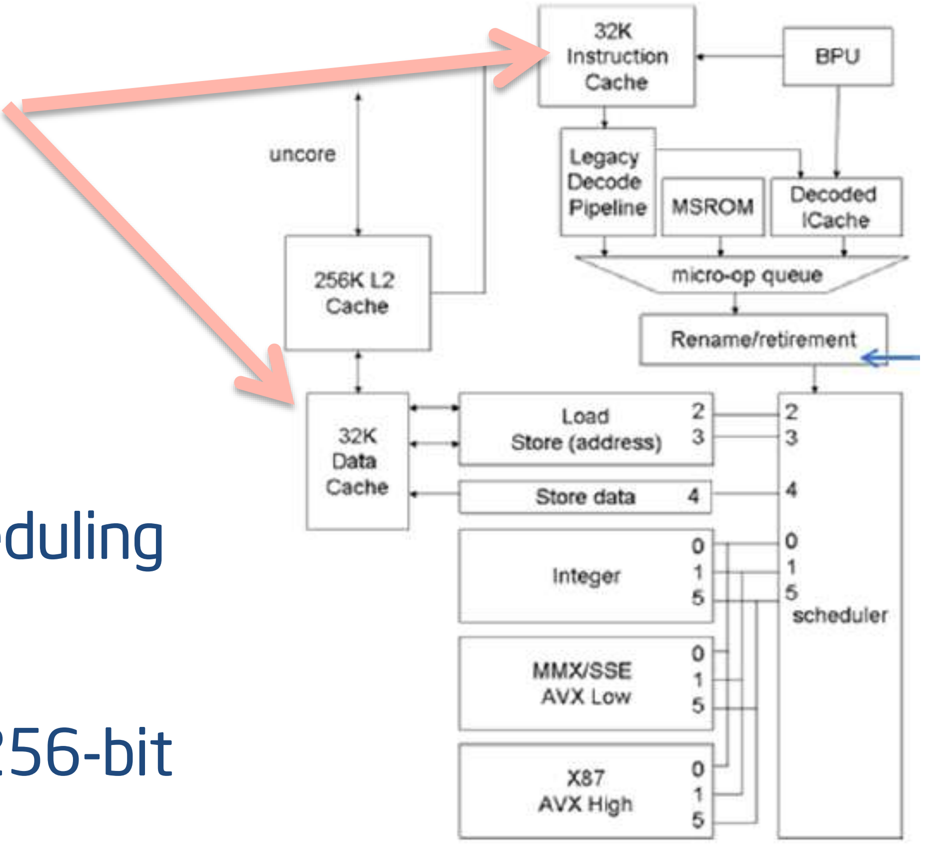
von Neumann architecture



Intel® Microarchitecture Codename SandyBridge

Key x86 Architecture Features for Developers To Know

- Cache (data and instr)
- Branch Prediction
- Out-of-order uOp Scheduling
- Wide Registers up to 256-bit



Intel® Microarchitecture
Codename SandyBridge

Cache Behavior Affects Performance

- Cost of Data Access increases with Distance from CPU

- Programming Tips:
  - Maximize work done on cached data
  - Work with Hardware Prefetch (arrays vs linked lists)

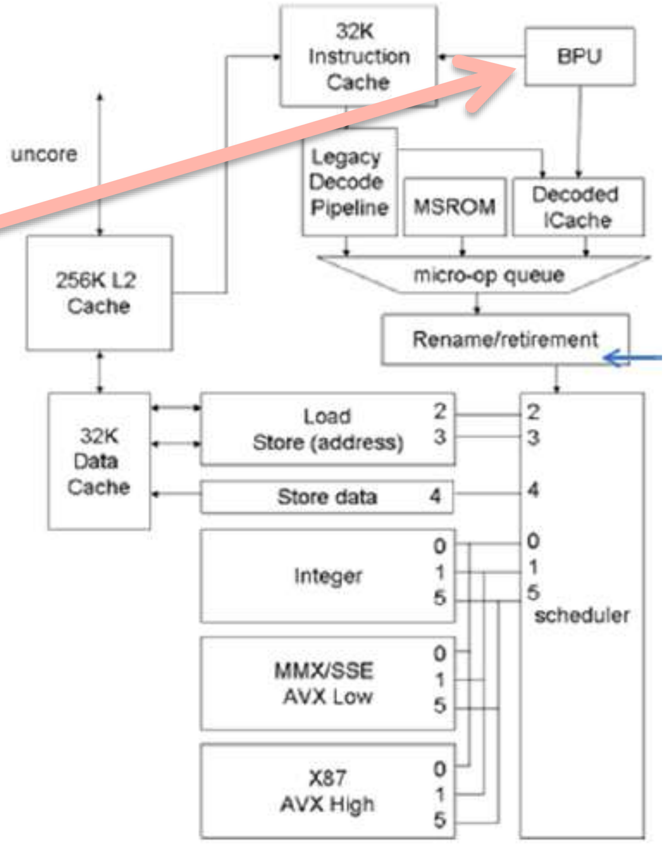
Cost of accessing data

Where Data Is Resident	Time to fetch data
Register	1 cycle
L1 Cache	4 cycles
L2 Cache	10 cycles
L3 Cache	40-75 cycles
Memory	60-100 ns

http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Key x86 Architecture Features for Developers To Know

- Cache (data and instr)
- **Branch Prediction**
- Out-of-order uOp Scheduling
- Wide Registers up to 256-bit



Intel® Microarchitecture Codename SandyBridge

BPU Helps (when predictable)

```
for(int i=0; i<4096; i++)
  m = max(m, x[i]);
```

Compiled using branch code

```
vmovss   xmm1,dword ptr[eax]
vcomiss  xmm0,xmm1
jbe      findmin+20h
vmovss   xmm0,xmm0,xmm1
add      eax,4
cmp      eax, (0BFF3C0h)
jl       findmax+12h
```

Compiled using max instruction

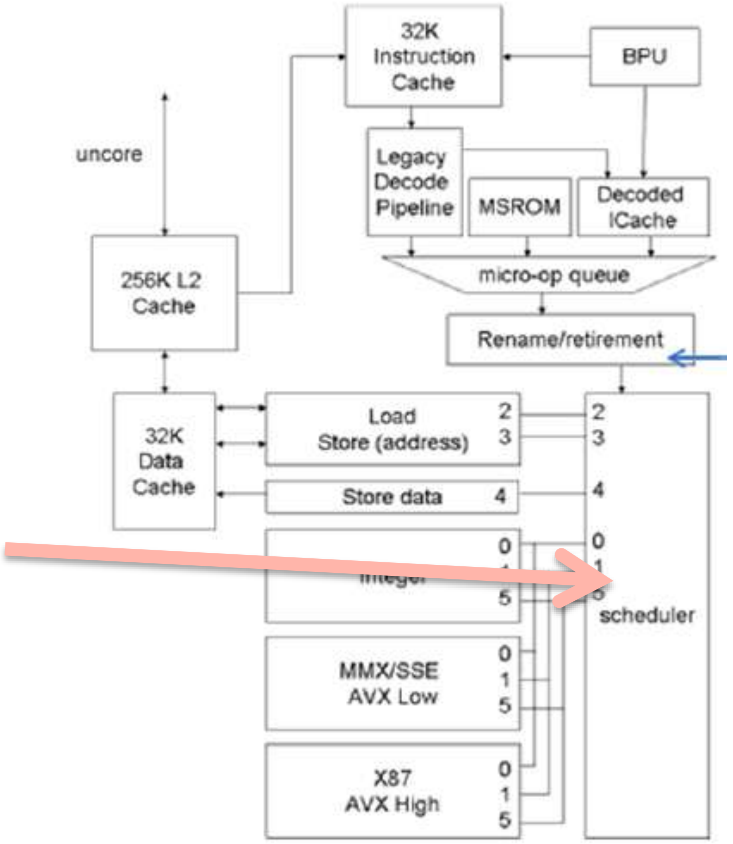
```
vmovss   xmm0,dword ptr[eax]
add      eax,4
vshufps  xmm0,xmm0,xmm0,0
vmaxss   xmm1,xmm1,xmm0
cmp      eax, (0BFF3C0h)
jl       findmax_intrin+16h
```

Array Ordering	branch	max uOp
Monotonic	2.1	3.0
Pathological	9.8	3.0
Random	2.2	3.0

cycles per iteration (lower is better)

Key x86 Architecture Features for Developers To Know

- Cache (data and instr)
- Branch Prediction
- **Out-of-order uOp Scheduling**
- Wide Registers up to 256-bit



Intel® Microarchitecture
Codename SandyBridge

Understanding Out-Of-Order Execution

How would you expect these 3 loops to perform?

Test	Code	Measured* CPU Cycles
SASPY	`for(int i=1; i<N; i++)` `s[i] = a * s[i-1] + y[i];`	
SAXPS	`for(int i=1; i<N; i++)` `s[i] = a * x[i] + s[i-1];`	
SAXPY	`for(int i=1; i<N; i++)` `s[i] = a * x[i] + y[i];`	

Comparison of 3 near-identical loops with different data access patterns

Similar code, yet significant differences in Performance

Test	Code	Measured* CPU Cycles

SASPY	for(int i=1; i<N; i++)
	  s[i] = a * s[i-1] + y[i];
		14.0

SAXPS	for(int i=1; i<N; i++)
	  s[i] = a * x[i] + s[i-1];
		9.0

SAXPY	for(int i=1; i<N; i++)
	  s[i] = a * x[i] + y[i];
		2.2

Assembly Code Comparison

```
for(int i=0;i<N;i++) // saxpy
  s[i] = a * x[i] + y[i];
```

```
for(int i=1;i<N;i++) // saspy
  s[i] = a * s[i-1] + y[i];
```

Assembly code:

```
vmovss xmm0,dword ptr X[eax]
vmulss xmm0,xmm0,dword ptr [A]
vaddss xmm0,xmm0,dword ptr Y[eax]
vmovss dword ptr S[eax],xmm0
add    eax,4
cmp    eax,1000h
jl     saxpy+7 (13A1041h)
```

```
vmovss xmm0,dword ptr S[eax]
vmulss xmm0,xmm0,dword ptr [A]
vaddss xmm0,xmm0,dword ptrY+4[eax]
vmovss dword ptr S+4 [eax],xmm0
add    eax,4
cmp    eax,0FFCh
jl     saspy+7 (13A10B5h)
```

Same instruction sequence!

Throughput != Latency

Instruction "cost" consists of two important timings:

- **Instruction Latency -** time to complete the operation and return result
- **Instruction Throughput -** frequency at which a new operation can be issued

Eg: while waiting 5 cycles for a multiplication to complete we can begin 5 other multiplication operations.

Operation	Latency	Throughput
+ - * rsqrt, rcp, hadd, min,max	3-5	1
div, sqrt	14	14
sin,cos	160-200	130
move load/store	>=1	.5
dot product	12	2

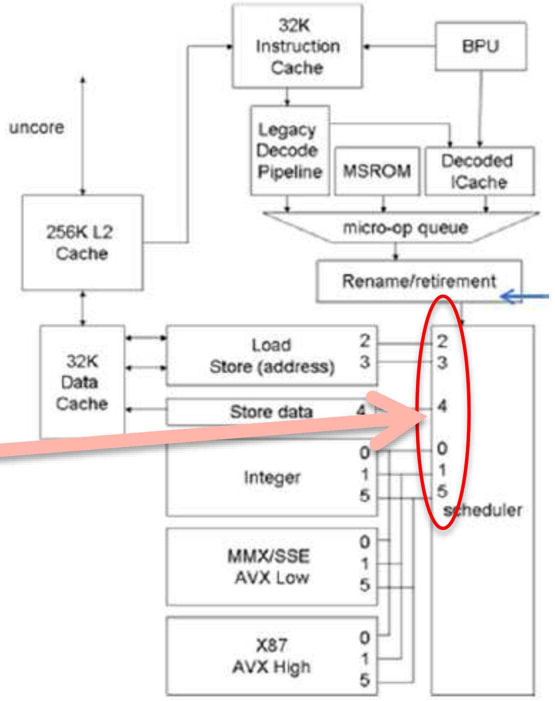
Note: This table is an extremely condensed version of the Intel® Architecture Manual

How x86 Instructions Get Processed

ASM instructions => Uops

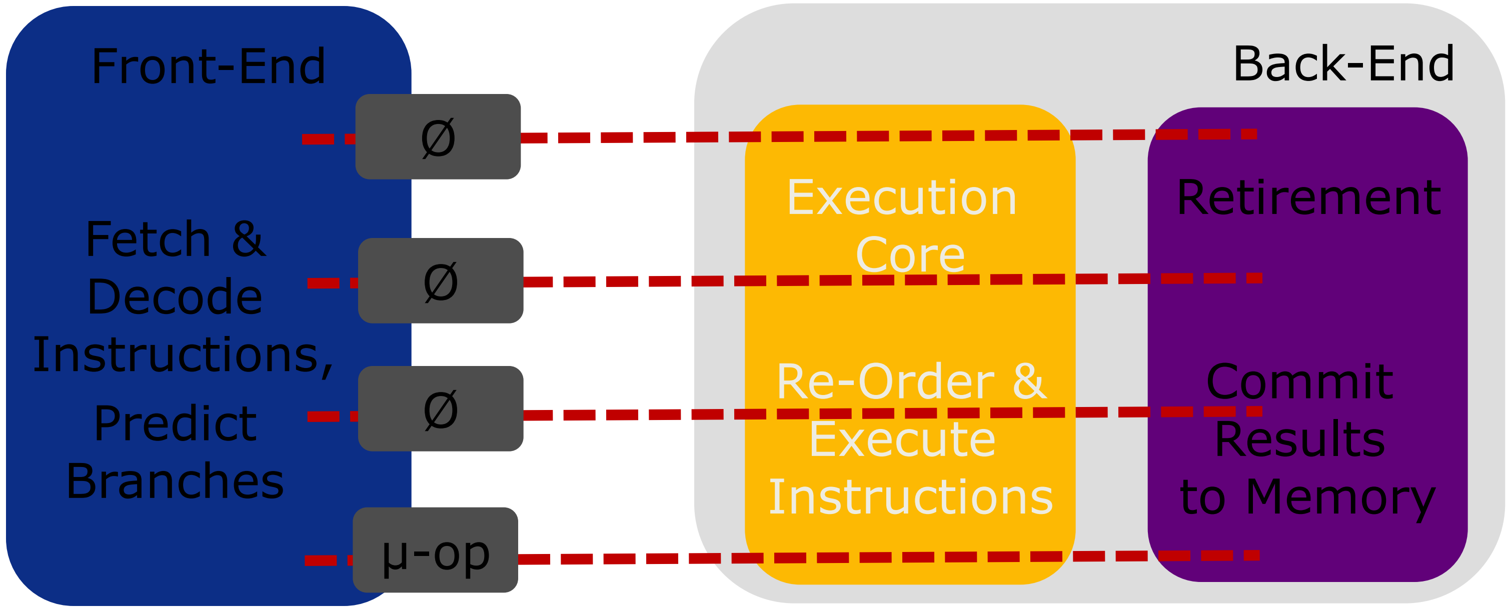
asm registers => physical registers

- uOps execute when ready
- Up to 1/port/cycle
- Data order dependent. i.e. Not instruction order dependent



Port #	0	1	2	3	4	5
Operations (uOps)	* /	+ -	Load	Load	Store	Shuffle

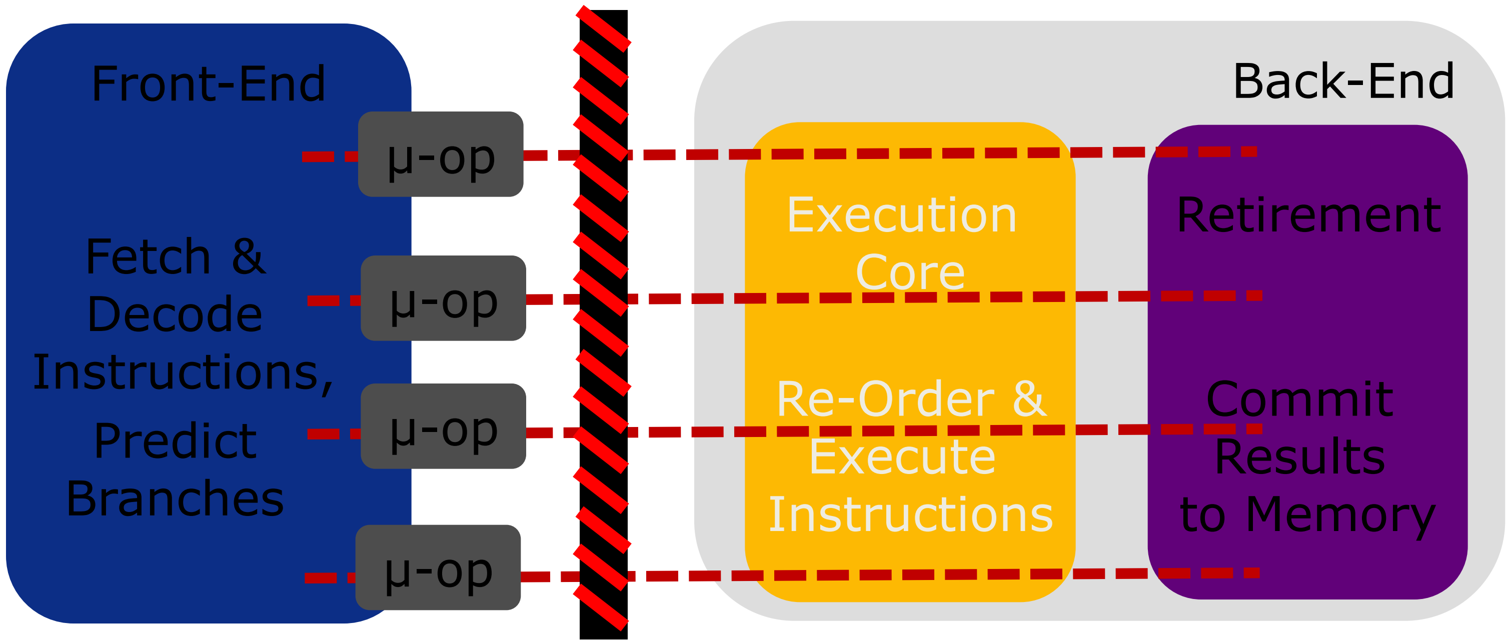
The Pipeline Slot Methodology, Illustrated


Front-End
Fetch & Decode Instructions, Predict Branches
Ø
Ø
Ø
µ-op
Back-End
Execution Core
Re-Order & Execute Instructions
Retirement
Commit Results to Memory

Case 1: Front-End does not provide micro-operations for all 4 pipeline slots

Front-End Bound

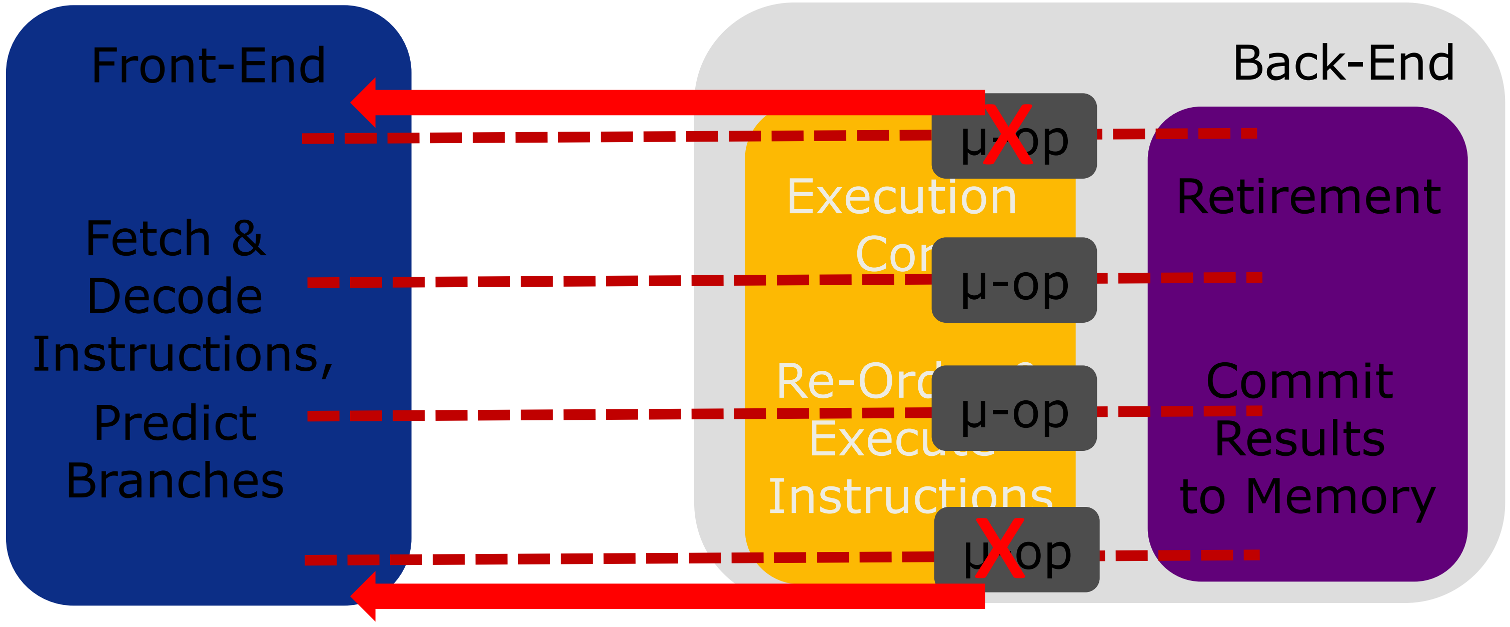
The Pipeline Slot Methodology, Illustrated



Case 2: Back-End cannot accept micro-operations for all 4 pipeline slots

Back-End Bound

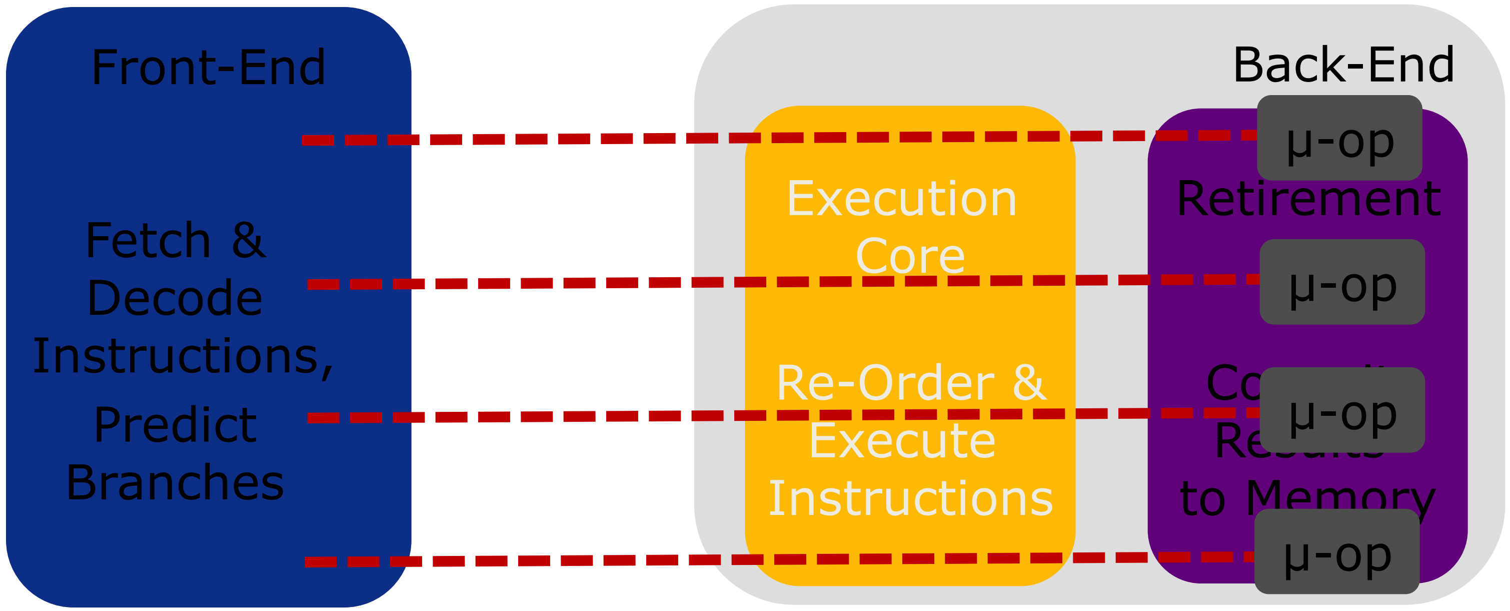
The Pipeline Slot Methodology, Illustrated



Case 3: Micro-operations make it to the Back-End, but then get removed from the pipeline

Cancelled

The Pipeline Slot Methodology, Illustrated



Case 4: Micro-operations make it to the Back-End, Execute, and then Retire

Retired

s=ax+y decomposed into uOps

Output from Intel Architecture Code Analyzer (IACA)

```
Intel(R) Architecture Code Analyzer
Loop Throughput: 2 Cycles; Loop Latency: 14 Cycles;

| Num  |   Ports pressure in cycles
| Uops |  0 |  1 |  2 |  3 |  4 |  5 | Assembly Code
------------------------------------------------------
|   1  |    |    |  1 |  _ |    |    | vmovss xmm0, ptr a
|   2^ |  1 |    |  _ |  1 |    |    | vmulss xmm0, xmm0, ptr x
|   2^ |    |  1 |  1 |  _ |    |    | vaddss xmm0, xmm0, ptr y
|   2^ |    |    |  _ |  1 |  1 |    | vmovss ptr s, xmm0
|   1  |  _ |  _ |    |    |    |  1 | add eax, 0x4
|   1  |  _ |  _ |    |    |    |  1 | cmp eax, 0x8000
|   0F |    |    |    |    |    |    | jl 0xffffffcc
-------------------------------------------------------
|Cycles|  1 |  1 |  2 |  2 |  1 |  2 |
-------------------------------------------------------
```

Note the range in throughput and latency

Data Dependencies Explain the Disparity in Performance

Test	Code	Measured* CPU Cycles

SASPY	`for(int i=1; i<N; i++)`
	`s[i] = a * s[i-1] + y[i];`
		14.0

SAXPS	`for(int i=1; i<N; i++)`
	`s[i] = a * x[i] + s[i-1];`
		9.0

SAXPY	`for(int i=1; i<N; i++)`
	`s[i] = a * x[i] + y[i];`
		2.2
		(1.6 unrolled)

Notes:

- Range matches predicted latency and throughput times.
- 2nd loop can begin multiplication early
- 3rd loop (no dependencies) benefits further with compiler-generated loop unroll

Reduce Dependencies to Maximize Throughput

Find Array Maximum	Code	Cycles

Standard Solution	`for(int i=0; i<N; i++)`
	`  m = max(m,x[i]);`
		3.0

Loop Unrolled	`for(int i=0; i<N; i+=2)`
	`  m = max(m,x[i]);`
	`  m = max(m,x[i+1]);`
		3.0

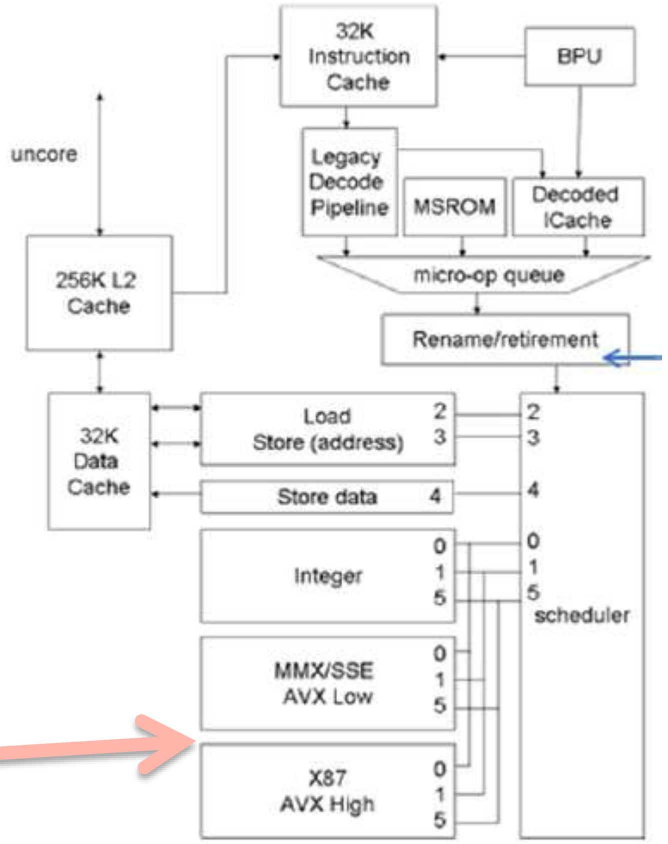
Dependence Reduced	`for(int i=0; i<N; i+=2)`
	`  m0 = max(m0,x[i]);`
	`  m1 = max(m1,x[i+1]);`
		1.6

Dependence Reduced Twice	`for(int i=0; i<N; i+=4)`
	`  m0 = max(m0,x[i]);`
	`  ...`
	`  m3 = max(m3,x[i+3]);`
		1.0

Note: x86 instruction VMAXSS has a latency of 3

Key x86 Architecture Features for Developers To Know

- Cache (data and instr)
- Branch Prediction
- Out-of-order uOp Scheduling
- **Wide Registers up to 256-bit**



Intel® Microarchitecture
Codename SandyBridge

Quick Review: "SSE" SIMD 128 bit (4 floats)

Arithmetic:
+ - * / sqrt

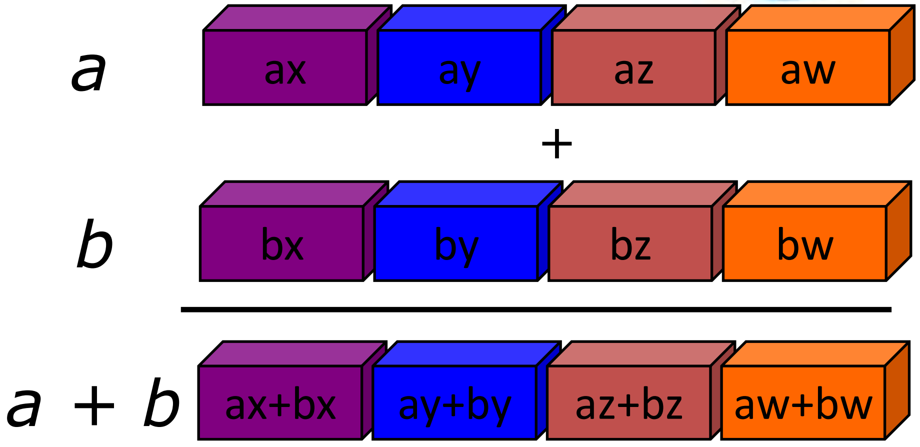
Bitwise:
& | ^

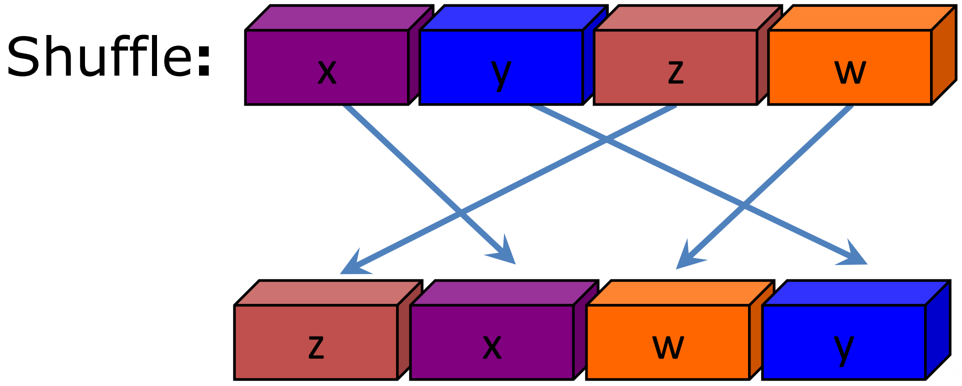
Transfer:
load/store
shuffle

Logical:
< > (returns mask)

Conditional:
blend (uses mask)

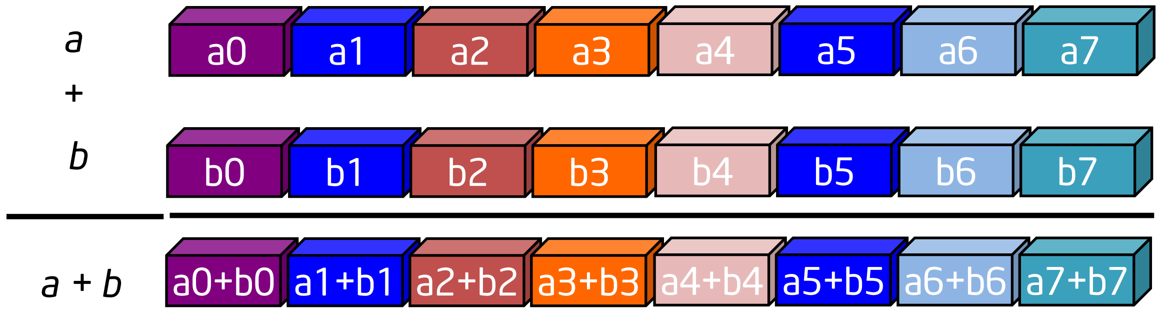
if-else statements handled by software predication





SIMD with AVX - up to 256 bit (8 floats)

- New instructions with 2nd generation Intel Core CPUs
- Supports 128 and 256-bit SIMD
- Non-destructive instructions



C/C++ Intrinsics

```
__m256 a,b,c;
...
c = _mm256_add_ps(a,b);
```

AVX SIMD applied to SAXPY

Test	Code	Measured* CPU Cycles / N

SAXPY
1 at a time
	`// float *s,*x,*y,a;`
	`for(int i=0; i<N; i++)`
	`  s[i] = a * x[i] + y[i];`
		2.2

SAXPY128
4 at a time
	`// __m128 *s,*x,*y,a;`
	`for(int i=0; i<N/4; i++)`
	`  s[i] = a * x[i] + y[i];`
		0.6

SAXPY256
8 at a time
	`// __m256 *s,*x,*y,a;`
	`for(int i=0; i<N/8; i++)`
	`  s[i] = a * x[i] + y[i];`
		0.3

*Measured time is total time divided by N (N==2048)

Array Maximum Example with AVX SIMD

Array Maximum	Code	Cycles Serial	SIMD4 version	SIMD8 version

Standard Solution	for(int i=0; i<N; i+=8)
	
	m=_mm256_max_ps((m256*)(x+i));
	//m = max(m,x[i]);
		3.0	0.73	0.36

Dependence Reduced	for(int i=0; i<N; i+=2)
	m0 = max(m0,x[i]);
	m1 = max(m1,x[i+1]);
		1.6	0.38	0.18

Dependence Reduced More	for(int i=0; i<N; i+=4)
	m0 = max(m0,x[i]);
	...
	m3 = max(m3,x[i+3]);
		1.0	0.26	0.13

Exploiting both SIMD and Instruction parallelism

SIMD Programming Patterns in Games

Typical “SSE” 4D Vector Class:

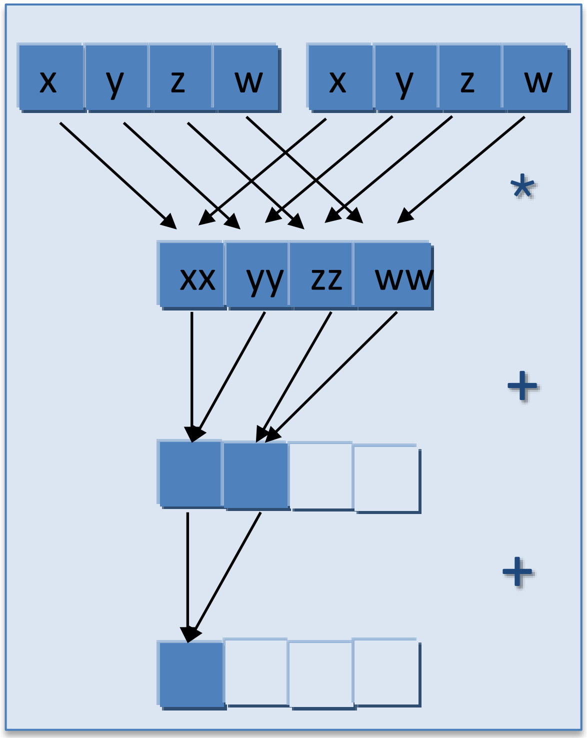
```
class Vec4
{
public:
  union {
    struct {float x,y,z,w;}
    _m128 v;
  }
};

inline Vec4 operator+(const Vec4 &a, const Vec4 &b)
{
  return Vec4(_mm_add_ps(a.v,b.v));
}
```

```
    ...
    Vec4 v = u + w ; // add two 4D vectors
    ...
```

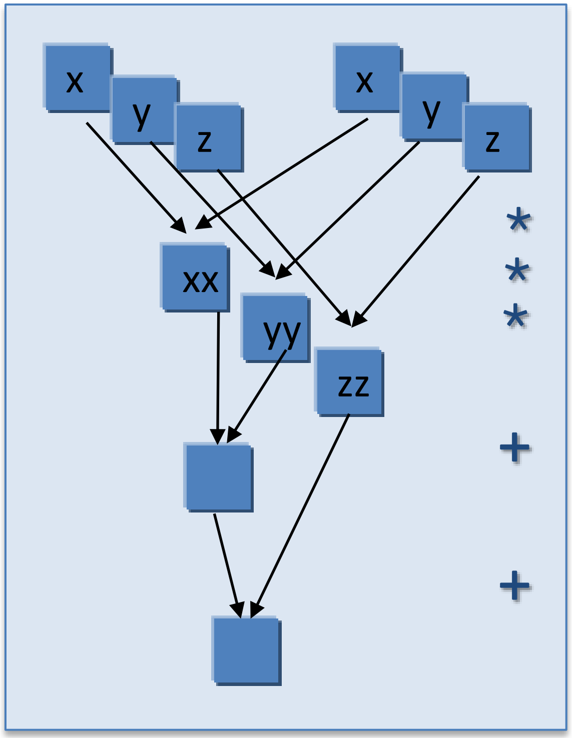
Scaling with 4D `xyzw` SIMD pattern

Real Results typically ~2X

- Not using all 4 flops at each operation.
- Often used for 3D data
- Shuffle overheads
- Instruction parallelism sometimes lost
- Pattern doesn't Scale to 256-bit (8float) SIMD



3D/4D dot product 128-bit SIMD



3D dot product serial

Another Way To Use SIMD is SOA

Array of Structures in Memory (AOS)

x0	y0	z0	w0	x1	y1	z1	w1	x2	…

Structure of Arrays in Memory (SOA)

x0	x1	x2	x3	x4	…
y0	y1	y2	y3	y4	…
z0	z1	z2	z3	z4	…
w0	w1	w2	w3	w4	…

C++ programming patterns for SOA

```
class Vec3<T>
{
public:
   T x;
   T y;
   T z;
};

Vec3<T> operator +(const Vec3<T> &a, const Vec3<T> &b)
{
          return Vec3<T>(a.x+b.x, a.y+b.y, a.z+b.z);
}
```

```
__m256 operator +(const __m256 &a, const __m256 &b)
{
          return _mm256_add_ps(a,b);
}
```

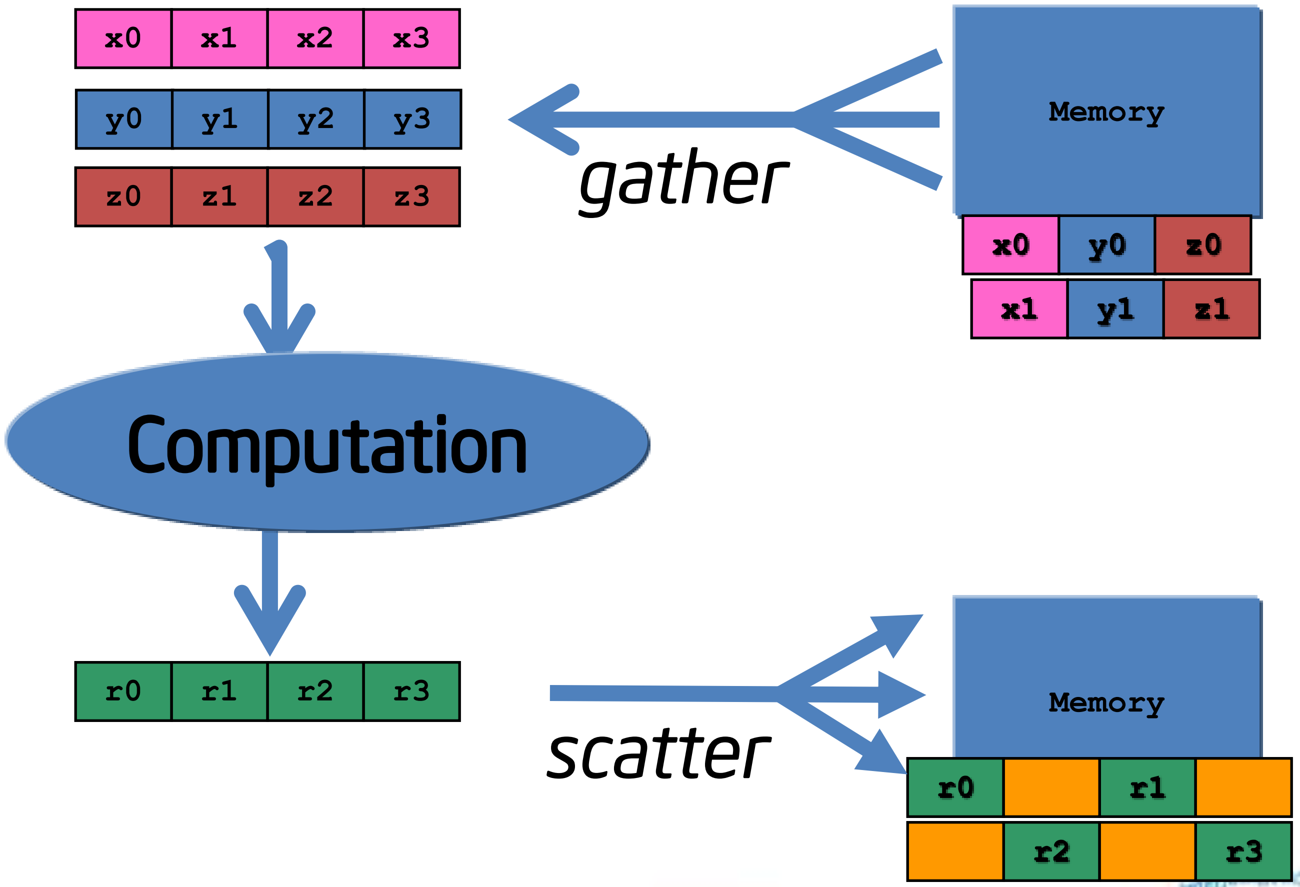
```
    ...
    Vec3<__m256> v = u + w;  // 8 vector additions at a time
    ...
```

Gather/Scatter - To SOA and Back!
Dealing With 'Real' Application Data

Gather
Input *xyz*

SIMD SOA
Parallelism

Scatter
Result *r*



AVX 256-bit Programming Patterns - Gather/Scatter

Technique

- Linear Traversal of an Array
  - Exploit regular access patterns
  - Use x86 shuffle for Transpose

Example

```
Vec3<float> v[];
for(int i=0; i<N; i+=8) {
  Vec3<__m256> u = trans8x3(v+i);
  u = Normalize(u); // 8 at a time
  trans3x8(v+i,u);
}
```

- Indexing/Indirection (Gather 8)
  - Use 4 float xyzw data pattern
  - Align Data (pad if necessary)
  - 4x8 transpose
  - SOA code patterns

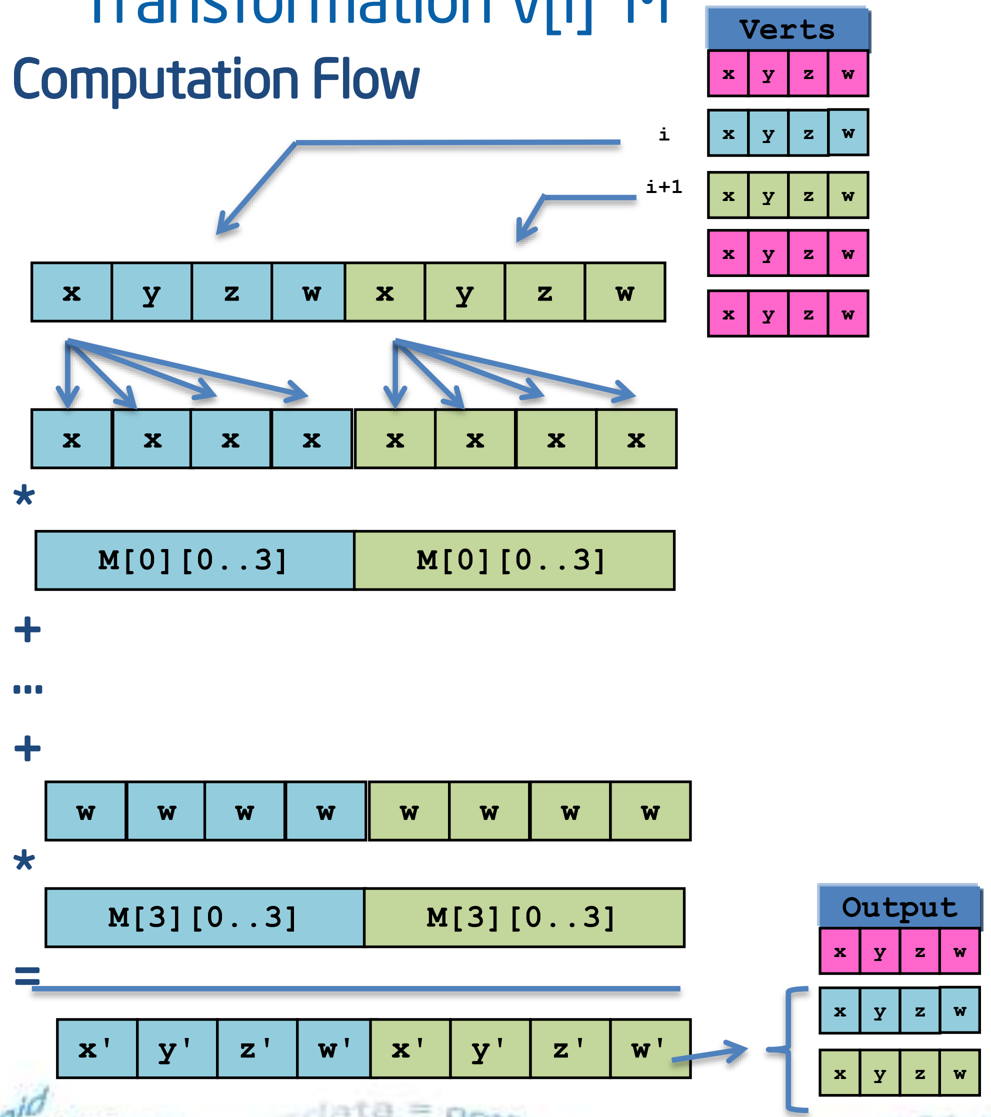
```
Vec4<float> v[];
for(int i=0; i<N; i+=8) {
  Vec8<__m128> g(v[k[i]],…,v[k[i+7]]);
  Vec4<__m256> u = trans8x4(g);
  r = MyCompute<__m256>(u); // do 8
  ...
}
```

- Indexing/Indirection (Gather 2)
  - Use 4 float xyzw data pattern
  - Use 256-bit as a way to Pair two 128-bit computations
  - AOS xyzw code patterns

```
Vec4<float> v[];
for(int i=0; i<N; i+=2) {
  __m256 u(v[k[i]],v[k[i+1]]);
  r = MyComputePair(u); // 2 at a time
  ...;
}
```

Pairing Two 128-bit Computations – Transformation v[i]*M

Computation Flow

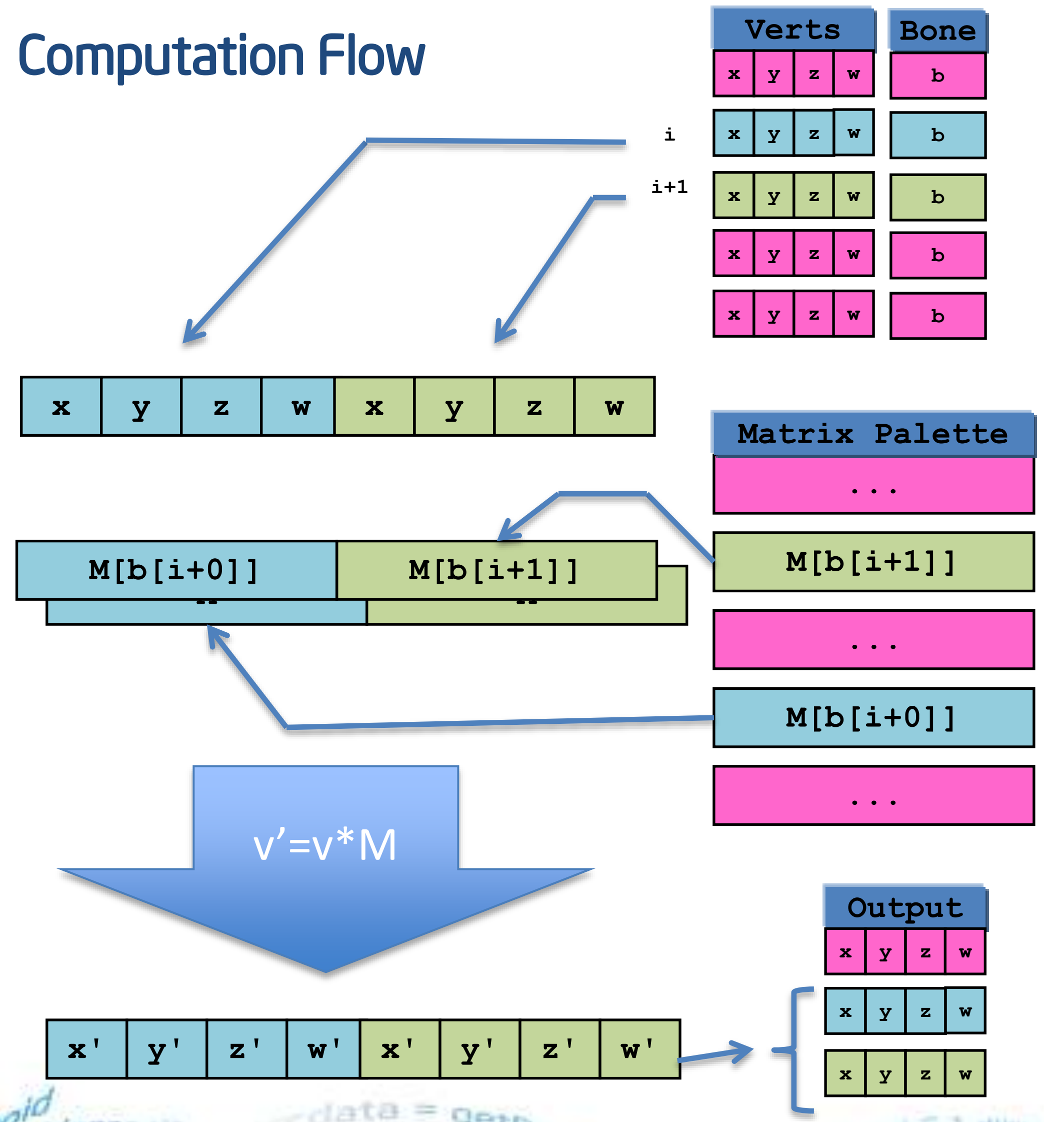


```
// Load M into both upper and lower halves of 256bit regs
__m256 m0 = _mm256_castps128_ps256(_mm_load_ps(M[0]));
__m256 m1 = _mm256_castps128_ps256(_mm_load_ps(M[1]));
__m256 m2 = _mm256_castps128_ps256(_mm_load_ps(M[2]));
__m256 m3 = _mm256_castps128_ps256(_mm_load_ps(M[3]));
m0 = _mm256_insertf128_ps(m0,_mm_load_ps(M[0]),1);
m1 = _mm256_insertf128_ps(m1,_mm_load_ps(M[1]),1);
m2 = _mm256_insertf128_ps(m2,_mm_load_ps(M[2]),1);
m3 = _mm256_insertf128_ps(m3,_mm_load_ps(M[3]),1);

for(int i=0;i<N;i+=2)  // transform two 4D points per iter
{
__m256 v = _mm256_load_ps(V[i]);
__m256 a = _mm256_mul_ps( m0 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 0,0,0,0)))    ;
a      = _mm256_add_ps( a ,_mm256_mul_ps( m1 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 1,1,1,1))))  ;
a      = _mm256_add_ps( a ,_mm256_mul_ps( m2 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 2,2,2,2))))  ;
a      = _mm256_add_ps( a ,_mm256_mul_ps( m3 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 3,3,3,3))))  ;
_mm256_store_ps(R[i],a);
}
```

Results: 1.7X speedup compared to 128 bit version

Pairing Two 128-bit Computations - Skinning Example

Computation Flow



```
for(int i=0;i<N;i+=2)  // single-bone skinning example
{
__m256 v = _mm256_load_ps(V[i]);
int b0 = Bone[i];
int b1 = Bone[i+1];
__m256 m0 = _mm256_castps128_ps256(_mm_load_ps(M[b0][0]));
__m256 m1 = _mm256_castps128_ps256(_mm_load_ps(M[b0][1]));
__m256 m2 = _mm256_castps128_ps256(_mm_load_ps(M[b0][2]));
__m256 m3 = _mm256_castps128_ps256(_mm_load_ps(M[b0][3]));
m0 = _mm256_insertf128_ps(m0,_mm_load_ps(M[b1][0]),1);
m1 = _mm256_insertf128_ps(m1,_mm_load_ps(M[b1][1]),1);
m2 = _mm256_insertf128_ps(m2,_mm_load_ps(M[b1][2]),1);
m3 = _mm256_insertf128_ps(m3,_mm_load_ps(M[b1][3]),1);

__m256 a =                 _mm256_mul_ps( m0 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 0,0,0,0)))   ;
a       = _mm256_add_ps( a ,_mm256_mul_ps( m1 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 1,1,1,1))))  ;
a       = _mm256_add_ps( a ,_mm256_mul_ps( m2 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 2,2,2,2))))  ;
a       = _mm256_add_ps( a ,_mm256_mul_ps( m3 ,
_mm256_shuffle_ps(v,v,_MM_SHUFFLE( 3,3,3,3))))  ;
_mm256_store_ps(R[i],a);
}
```

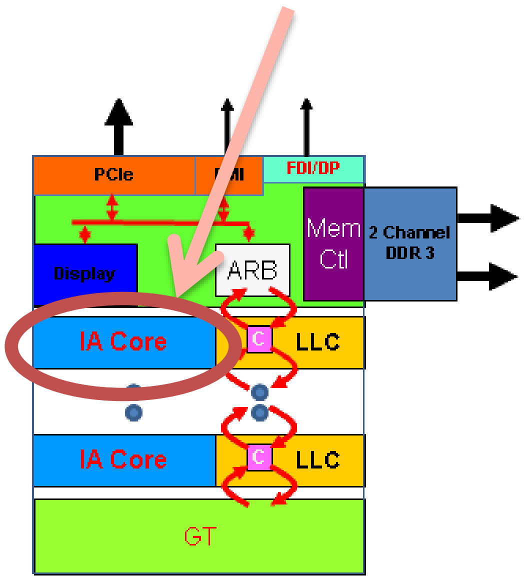
Results: 1.4X speedup compared to 128 bit version

Performance tuning

Agenda:

- x86 Architecture
  - Program Execution Flow
  - AVX® SIMD
  - Easy Effective code patterns
- **Performance Tuning Workflow**
  - **Hotspot profiling**
  - **Events and vTune® performance guided analysis**
- Walkthrough/Examples

You Are Here



Intel® Microarchitecture Codename SandyBridge

Take the Guesswork out of Optimization!

Code profiling and performance tuning

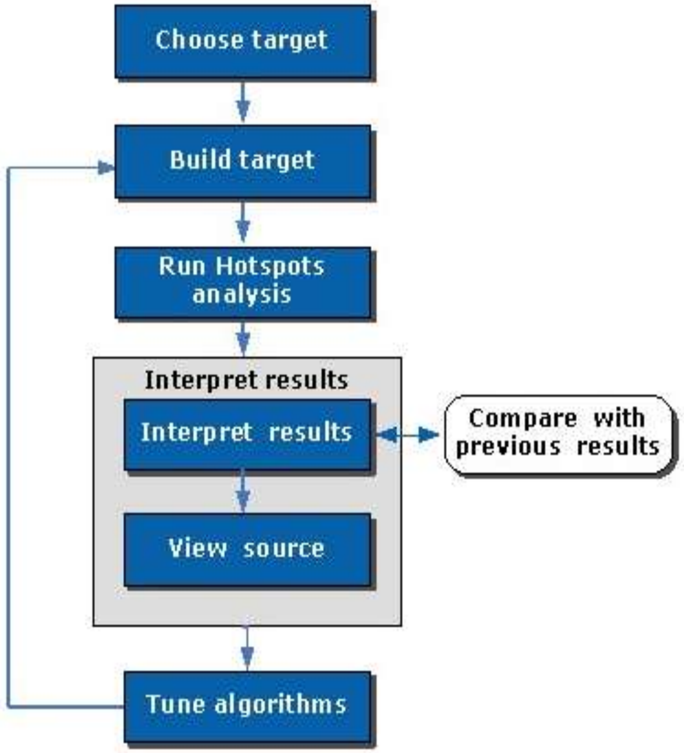
- Goal: Make programs run *faster*
- For video games, it's low latency
  - Finish drawing in a bounded amount of time
- Where do I start optimizing?
  - Limited time, maximize effort
- Solution: Use code profiling tools for performance tuning

2 kinds of performance tuning

- **Algorithmic**
  - Applies to all architectures
  - Generally improves code elegance and conciseness
  - Also includes the quality of parallel decompositions, CPU usage, and other multithreading issues
- **Hardware**
  - Architecture-specific (though commonalities exist)
  - Tends to obfuscate code (e.g. matrix blocking)
  - Requires architectural understanding

Both are essential!

Analysis and tuning workflow



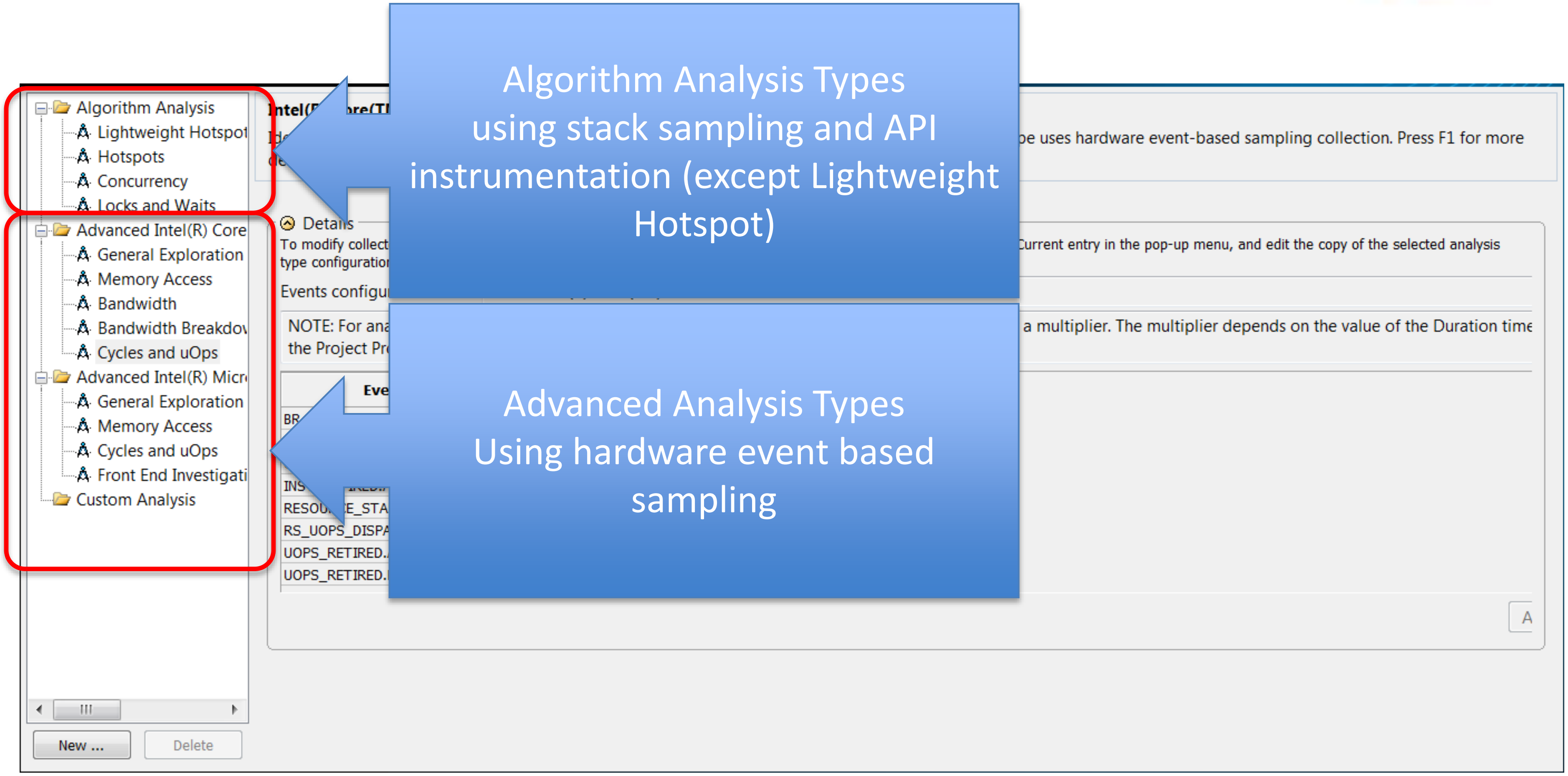
Performance tuning is an iterative process

Intel® VTune™ Amplifier XE

- Helps analyze code performance
  - Multi-threaded and hardware bottlenecks
  - Find hotspots, analyze thread performance
  - Compare before and after performance
- Available for Windows and Linux
  - Integrates with Microsoft Visual Studio
  - Also standalone GUI for both Windows & Linux

http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

Types of analysis

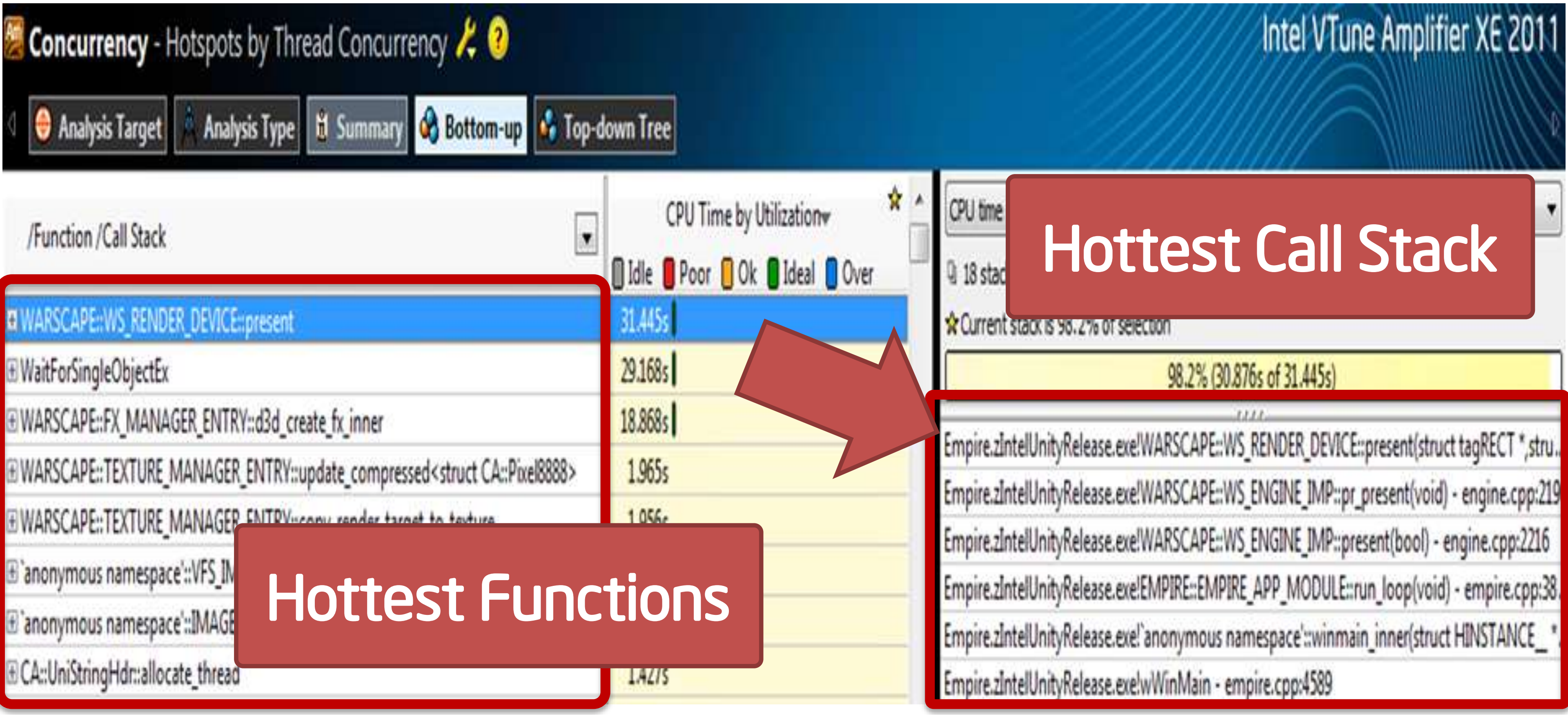


Algorithm Analysis
Lightweight Hotspot
Hotspots
Concurrency
Locks and Waits
Advanced Intel(R) Core
General Exploration
Memory Access
Bandwidth
Bandwidth Breakdov
Cycles and uOps
Advanced Intel(R) Micr
General Exploration
Memory Access
Cycles and uOps
Front End Investigati
Custom Analysis

Algorithm Analysis Types using stack sampling and API instrumentation (except Lightweight Hotspot)

Advanced Analysis Types Using hardware event based sampling

Intel VTune Amplifier XE
Algorithmic Analysis: Hotspots



/Function /Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)
WARSCAPE::WS_RENDER_DEVICE::present	31.445s
WaitForSingleObjectEx	29.168s
WARSCAPE::FX_MANAGER_ENTRY::d3d_create_fx_inner	18.868s
WARSCAPE::TEXTURE_MANAGER_ENTRY::update_compressed<struct CA::Pixel8888>	1.965s
WARSCAPE::TEXTURE_MANAGER_ENTRY::copy_render_target_to_texture	1.956s
`anonymous namespace'::VFS_IM	
`anonymous namespace'::IMAGE	
CA::UniStringHdr::allocate_thread	1.427s

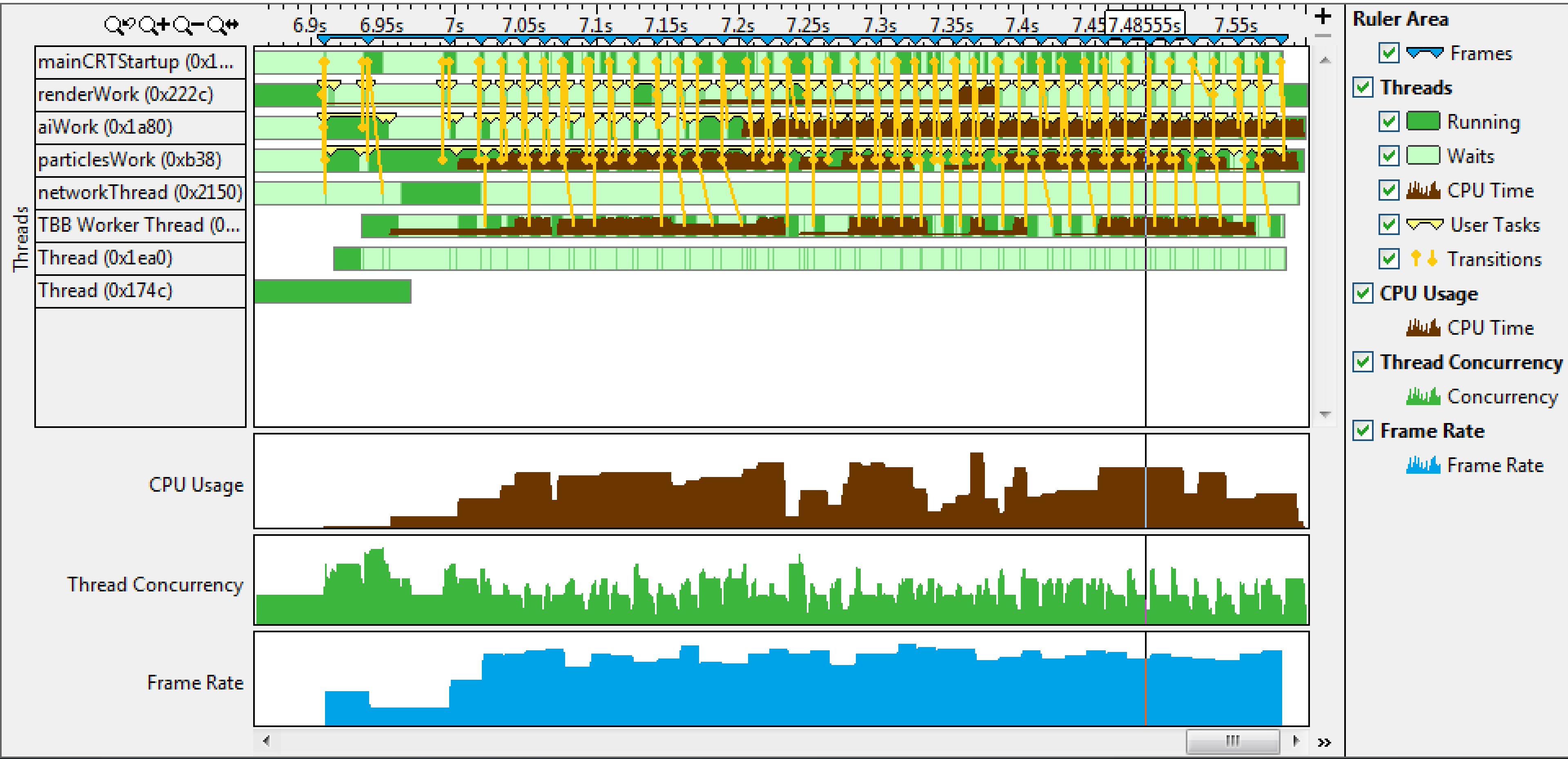
CPU time
18 stac
Current stack is 98.2% of selection
98.2% (30.876s of 31.445s)

```
Empire.zIntelUnityRelease.exe!WARSCAPE::WS_RENDER_DEVICE::present(struct tagRECT *,stru
Empire.zIntelUnityRelease.exe!WARSCAPE::WS_ENGINE_IMP::pr_present(void) - engine.cpp:219
Empire.zIntelUnityRelease.exe!WARSCAPE::WS_ENGINE_IMP::present(bool) - engine.cpp:2216
Empire.zIntelUnityRelease.exe!EMPIRE::EMPIRE_APP_MODULE::run_loop(void) - empire.cpp:38
Empire.zIntelUnityRelease.exe!`anonymous namespace'::winmain_inner(struct HINSTANCE_ *
Empire.zIntelUnityRelease.exe!wWinMain - empire.cpp:4589
```

Quickly identify what is important

Intel VTune Amplifier XE
Algorithmic Analysis
Concurrency and Frame Analysis

Frames

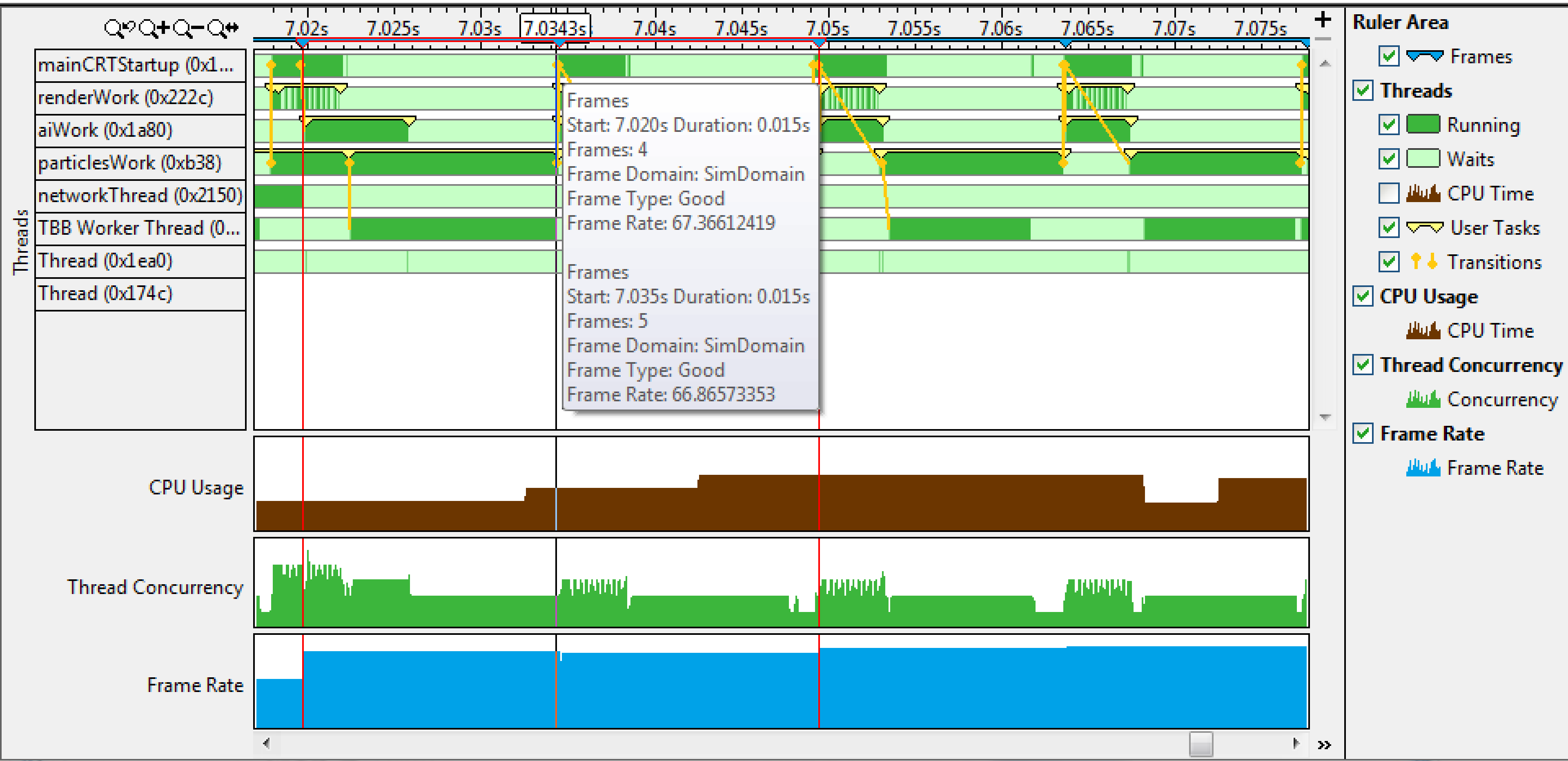


Intel VTune Amplifier XE

Algorithmic Analysis

Concurrency and Frame Analysis

Frames



Frames
Start: 7.020s Duration: 0.015s
Frames: 4
Frame Domain: SimDomain
Frame Type: Good
Frame Rate: 67.36612419

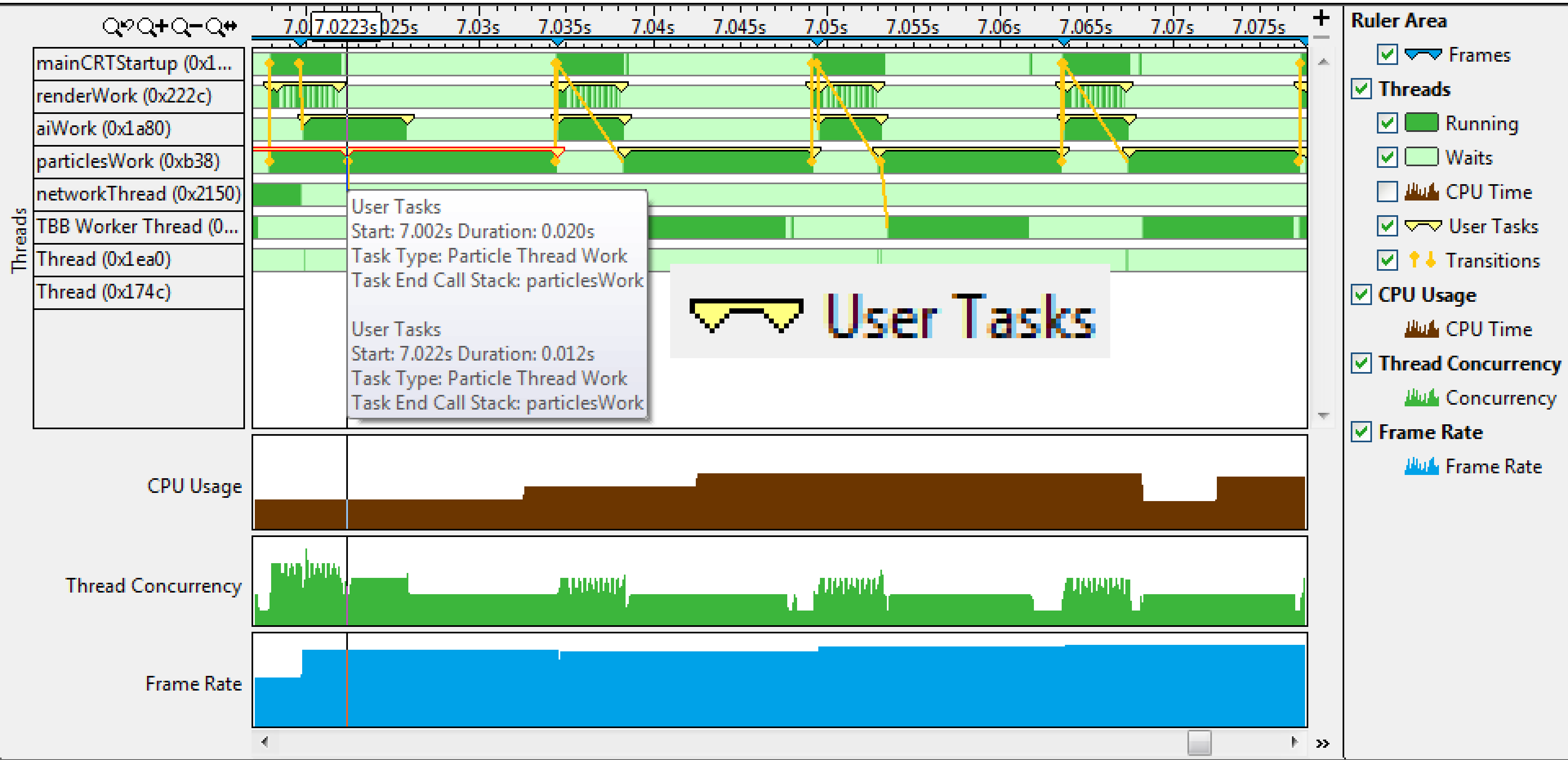
Frames
Start: 7.035s Duration: 0.015s
Frames: 5
Frame Domain: SimDomain
Frame Type: Good
Frame Rate: 66.86573353

Intel VTune Amplifier XE

Algorithmic Analysis

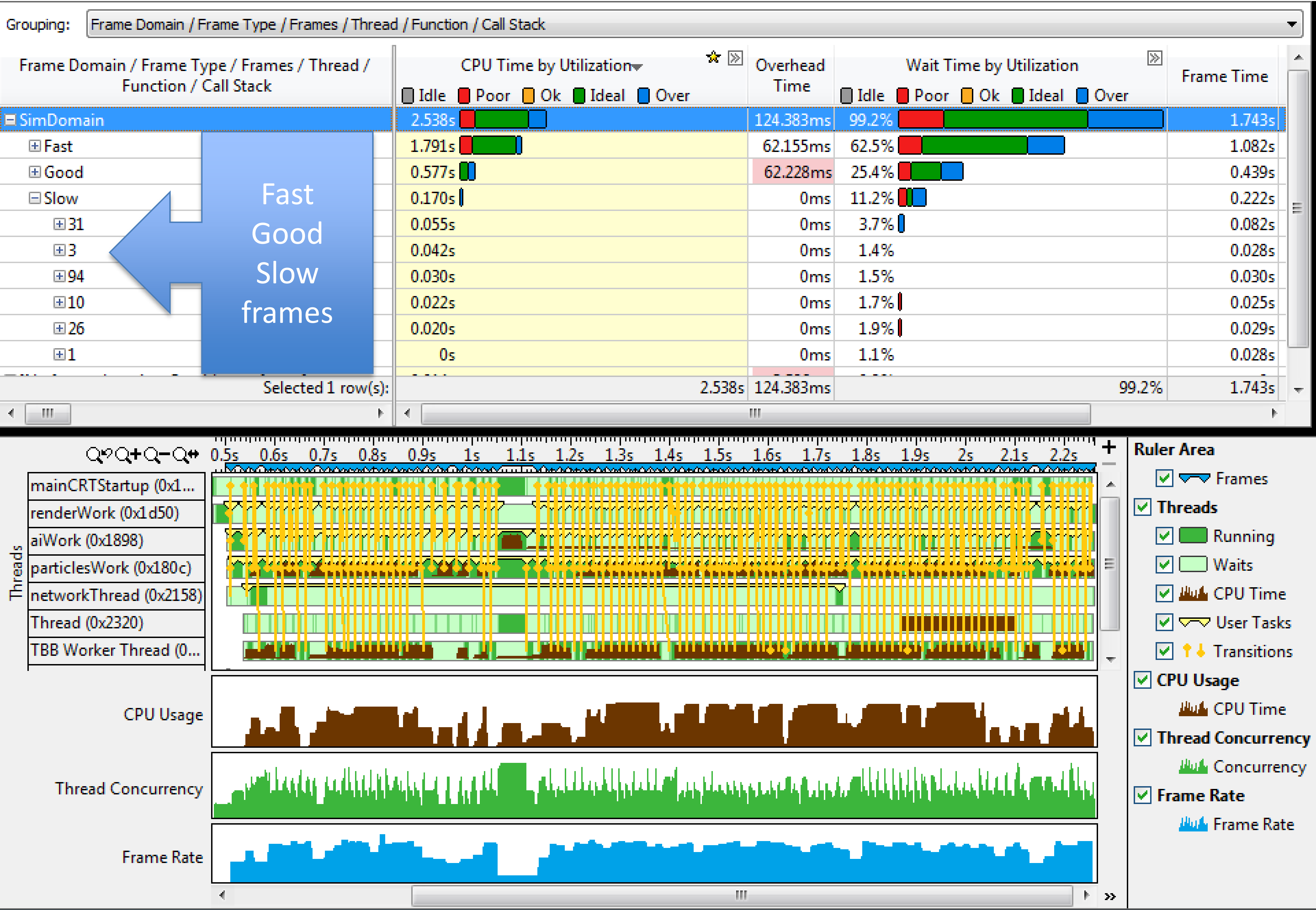
Concurrency and Frame Analysis

Frames



User Tasks
Start: 7.002s Duration: 0.020s
Task Type: Particle Thread Work
Task End Call Stack: particlesWork

User Tasks
Start: 7.022s Duration: 0.012s
Task Type: Particle Thread Work
Task End Call Stack: particlesWork



Grouping: Frame Domain / Frame Type / Frames / Thread / Function / Call Stack

Frame Domain / Frame Type / Frames / Thread / Function / Call Stack	CPU Time by Utilization (🔲 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Overhead Time	Wait Time by Utilization (🔲 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Frame Time
⊟ SimDomain	2.538s	124.383ms	99.2%	1.743s
⊞ Fast	1.791s	62.155ms	62.5%	1.082s
⊞ Good	0.577s	62.228ms	25.4%	0.439s
⊟ Slow	0.170s	0ms	11.2%	0.222s
⊞ 31	0.055s	0ms	3.7%	0.082s
⊞ 3	0.042s	0ms	1.4%	0.028s
⊞ 94	0.030s	0ms	1.5%	0.030s
⊞ 10	0.022s	0ms	1.7%	0.025s
⊞ 26	0.020s	0ms	1.9%	0.029s
⊞ 1	0s	0ms	1.1%	0.028s
Selected 1 row(s):	2.538s	124.383ms	99.2%	1.743s

Fast
Good
Slow
frames

Threads:
mainCRTStartup (0x1...
renderWork (0x1d50)
aiWork (0x1898)
particlesWork (0x180c)
networkThread (0x2158)
Thread (0x2320)
TBB Worker Thread (0...

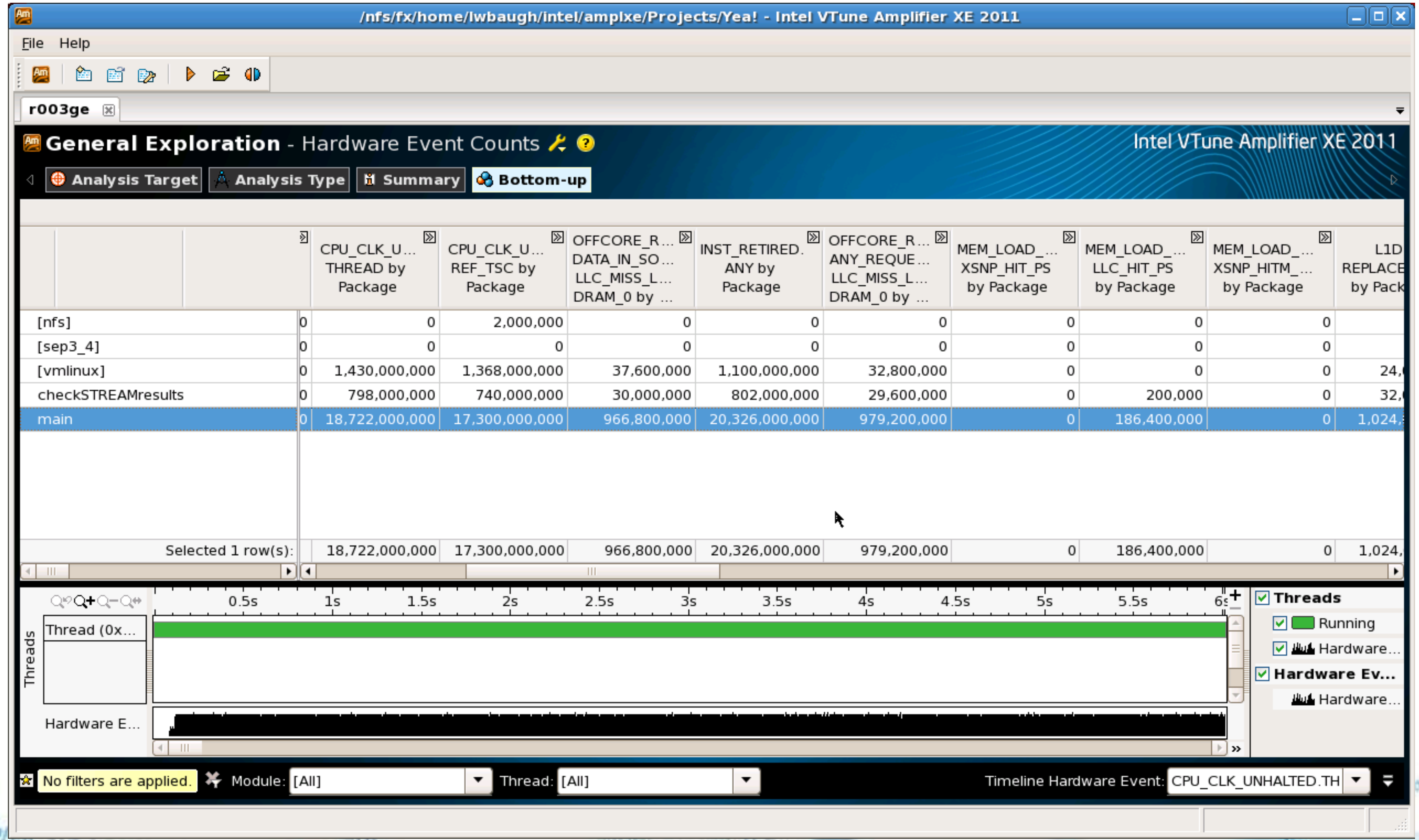
Hardware event-based sampling

- Performance Monitoring Unit (PMU) counters
  - + Offer a unique and powerful view into the CPU
    - reveal architectural bottlenecks in uninstrumented code running at full speed
    - hundreds of events offer insights into every part of the microarchitecture
  - – Methodology to use event counters in a top-down optimization methodology, but beyond the scope of this class
- At the level of functions and higher, raw events aren't that useful
  - Who cares if I experienced 2,166,000,000 DTLB misses? What matters is how much it cost me!

Hardware event-based sampling

- VTune™ Amplifier XE helps make PMU-based performance tuning easier
  - Several predefined analysis types help you focus on specific problems
  - Even better, VTune™ Amplifier XE shows *metrics* over PMU event counts. Instead of 2.17B DTLB misses, we can see what proportion of the time the app was dealing with DTLB overhead…
- What does that look like?

Hardware event-based sampling -- Example



/nfs/fx/home/lwbaugh/intel/amplxe/Projects/Yea! - Intel VTune Amplifier XE 2011

File Help

r003ge

General Exploration - Hardware Event Counts

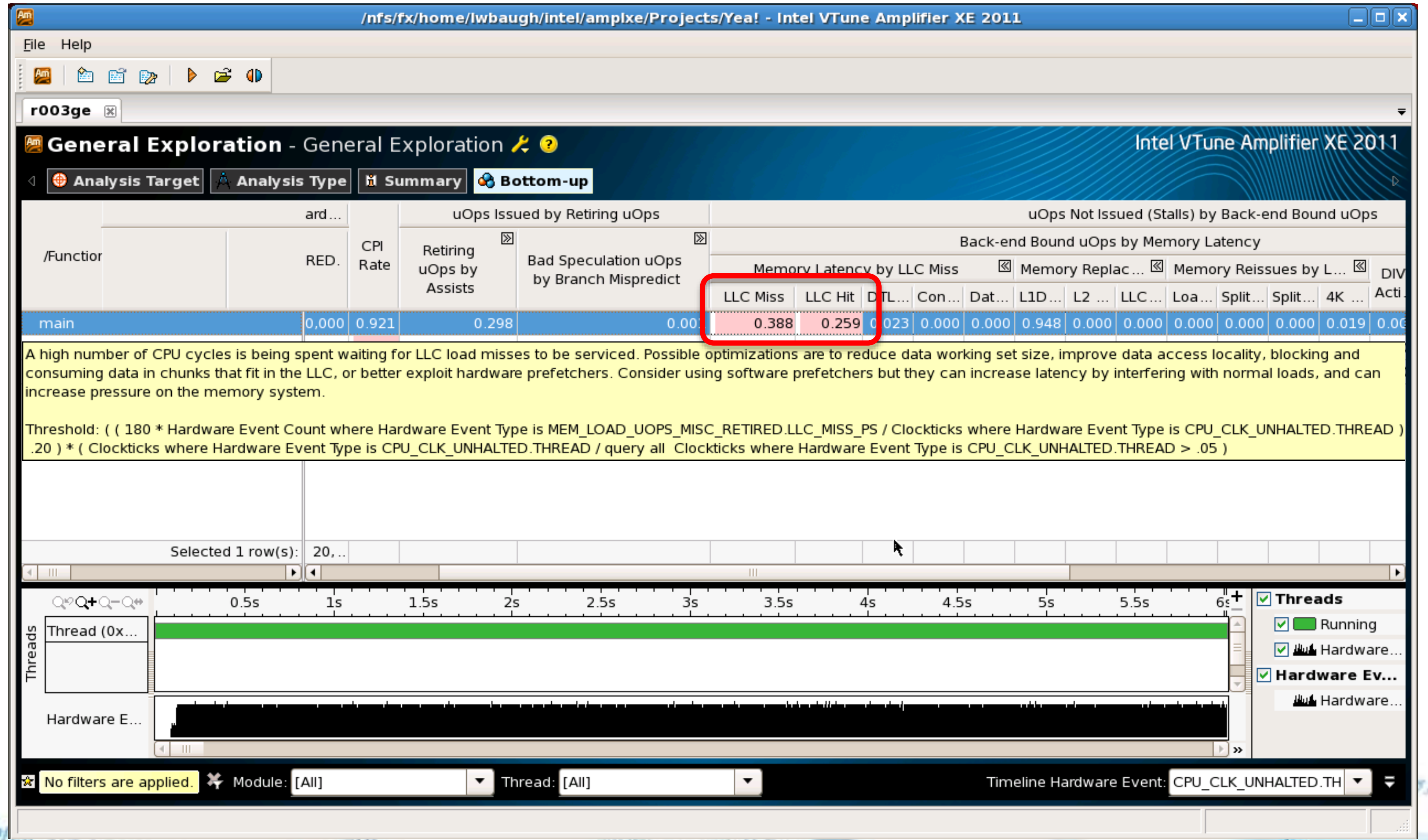
Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Summary | **Bottom-up**

		CPU_CLK_U... THREAD by Package	CPU_CLK_U... REF_TSC by Package	OFFCORE_R... DATA_IN_SO... LLC_MISS_L... DRAM_0 by ...	INST_RETIRED. ANY by Package	OFFCORE_R... ANY_REQUE... LLC_MISS_L... DRAM_0 by ...	MEM_LOAD_... XSNP_HIT_PS by Package	MEM_LOAD_... LLC_HIT_PS by Package	MEM_LOAD_... XSNP_HITM_... by Package	L1D REPLACE by Pack
[nfs]	0	0	2,000,000	0	0	0	0	0	0	
[sep3_4]	0	0	0	0	0	0	0	0	0	
[vmlinux]	0	1,430,000,000	1,368,000,000	37,600,000	1,100,000,000	32,800,000	0	0	0	24,
checkSTREAMresults	0	798,000,000	740,000,000	30,000,000	802,000,000	29,600,000	0	200,000	0	32,
main	0	18,722,000,000	17,300,000,000	966,800,000	20,326,000,000	979,200,000	0	186,400,000	0	1,024,
Selected 1 row(s):		18,722,000,000	17,300,000,000	966,800,000	20,326,000,000	979,200,000	0	186,400,000	0	1,024,

No filters are applied. Module: [All] Thread: [All] Timeline Hardware Event: CPU_CLK_UNHALTED.TH

Hardware event-based sampling -- Example


/nfs/fx/home/lwbaugh/intel/amplxe/Projects/Yea! - Intel VTune Amplifier XE 2011
File Help
r003ge
General Exploration - General Exploration
Intel VTune Amplifier XE 2011
Analysis Target
Analysis Type
Summary
Bottom-up
Threads
Running
Hardware...
Hardware Ev...
Thread (0x...
Hardware E...
0.5s 1s 1.5s 2s 2.5s 3s 3.5s 4s 4.5s 5s 5.5s 6s

/Function	ard... RED.	CPI Rate	uOps Issued by Retiring uOps: Retiring uOps by Assists	Bad Speculation uOps by Branch Mispredict	Memory Latency by LLC Miss: LLC Miss	LLC Hit	DTL...	Con...	Dat...	L1D...	L2 ...	LLC...	Loa...	Split...	Split...	4K ...	DIV Acti...
main	0.000	0.921	0.298	0.00	0.388	0.259	0.023	0.000	0.000	0.948	0.000	0.000	0.000	0.000	0.000	0.019	0.00

uOps Not Issued (Stalls) by Back-end Bound uOps — Back-end Bound uOps by Memory Latency — Memory Latency by LLC Miss | Memory Replac... | Memory Reissues by L...

A high number of CPU cycles is being spent waiting for LLC load misses to be serviced. Possible optimizations are to reduce data working set size, improve data access locality, blocking and consuming data in chunks that fit in the LLC, or better exploit hardware prefetchers. Consider using software prefetchers but they can increase latency by interfering with normal loads, and can increase pressure on the memory system.

Threshold: ((180 * Hardware Event Count where Hardware Event Type is MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS / Clockticks where Hardware Event Type is CPU_CLK_UNHALTED.THREAD) .20) * (Clockticks where Hardware Event Type is CPU_CLK_UNHALTED.THREAD / query all Clockticks where Hardware Event Type is CPU_CLK_UNHALTED.THREAD > .05)

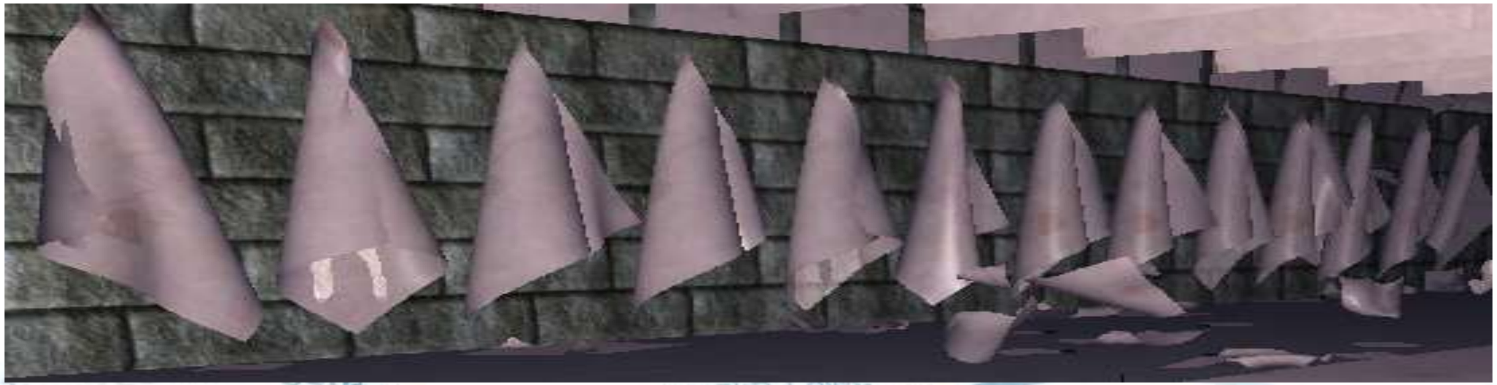
Selected 1 row(s): 20,...

No filters are applied. Module: [All] Thread: [All] Timeline Hardware Event: CPU_CLK_UNHALTED.TH

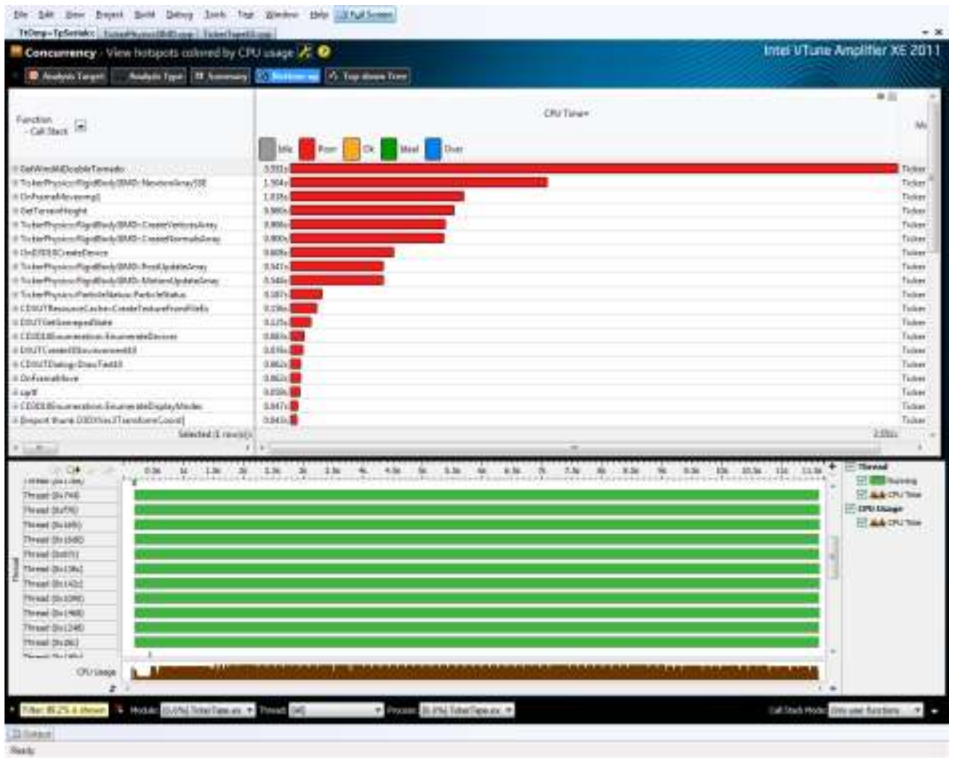
AVX Cloth Sample

Maximizing Throughput and exploiting 8-wide SIMD in practice

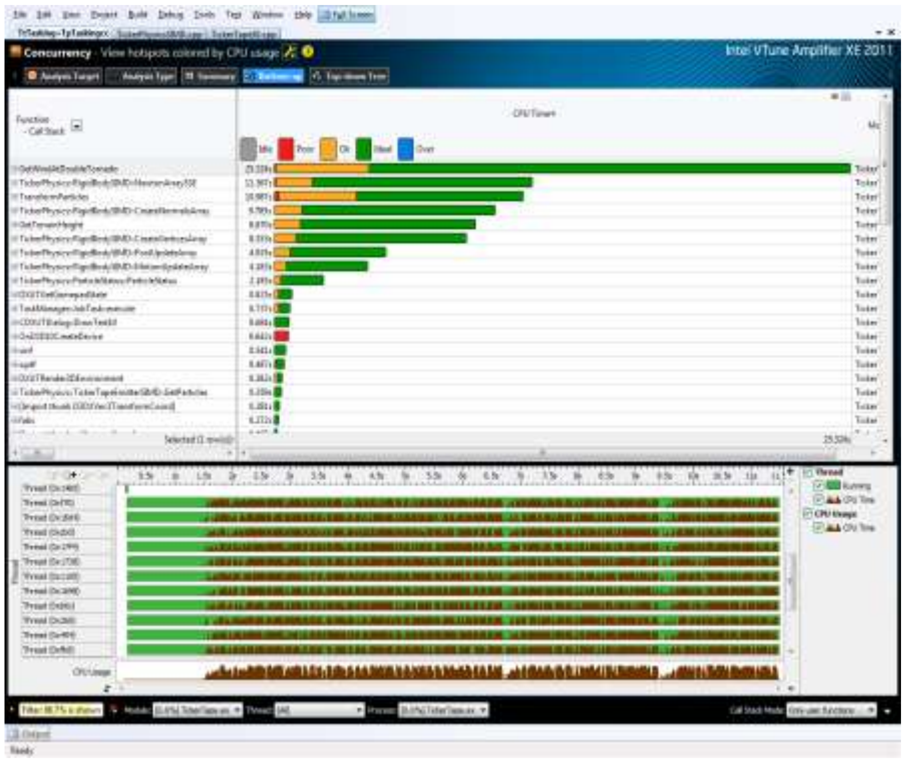
- Cloth Simulation Background
- Distance Constraint Update (Key Hotspot)
- Aligned Data and picked working-set sizes to fit cache
- Ordering the constraints to avoid data dependency
- Mapping to 8-float SIMD with SOA
- AVX transpose to AOS vertex buffer



Demo of VTune Amplifier XE on AVX Cloth


Concurrency - View hotspots colored by CPU usage
Intel VTune Amplifier XE 2011
Idle
Poor
Ok
Ideal
Over

Before optimization


Concurrency - View hotspots colored by CPU usage
Intel VTune Amplifier XE 2011
Idle
Poor
Ok
Ideal
Over

After optimization

Tools help in tuning performance

- Not automatically.. yet!
  - This is a Hard Problem: what does ‘run faster’ mean? What behavior is correct and what incorrect?
- They show where code is slow, and why it’s slow
  - Hotspots, the fundamental unit of performance tuning
- Performance tuning is an iterative process
  - Phases of analysis, using tools like VTune Amplifier XE, alternate with phases of contemplation and code editing
- In the end, developers must decide their goal

Fast Code == More Stuff == More Fun

- Optimize code to harness modern CPU power
  - Saturate execution ports with work every cycle
- Do big pieces of work – Consider an AVX build of your app
  - do 8 at a time and fully utilize AVX SIMD
  - SOA if possible (static or on-the-fly data transpose)
  - Pair 4D SIMD patterns otherwise
- Sanity check the source code (and perhaps assembly) for obvious inefficiencies
  - With timing or VTune Amplifier analysis, verify program flow is optimal
  - Watch for cache misses, branch prediction misses, port underutilization

Resources: Programming with AVX

For AVX support

- Intel 2nd Generation Core family, AMD's upcoming CPU
- Windows 7 SP1
- Visual Studio 2010 SP1
- Intel® Composer XE (Intel Compiler 12.0)

Resources

- AVX Cloth demo
- Intel VTune Amplifier XE Performance Profiler

Legal Disclaimers

Optimization Notice