# Massive Parallelism in Graphics Processors

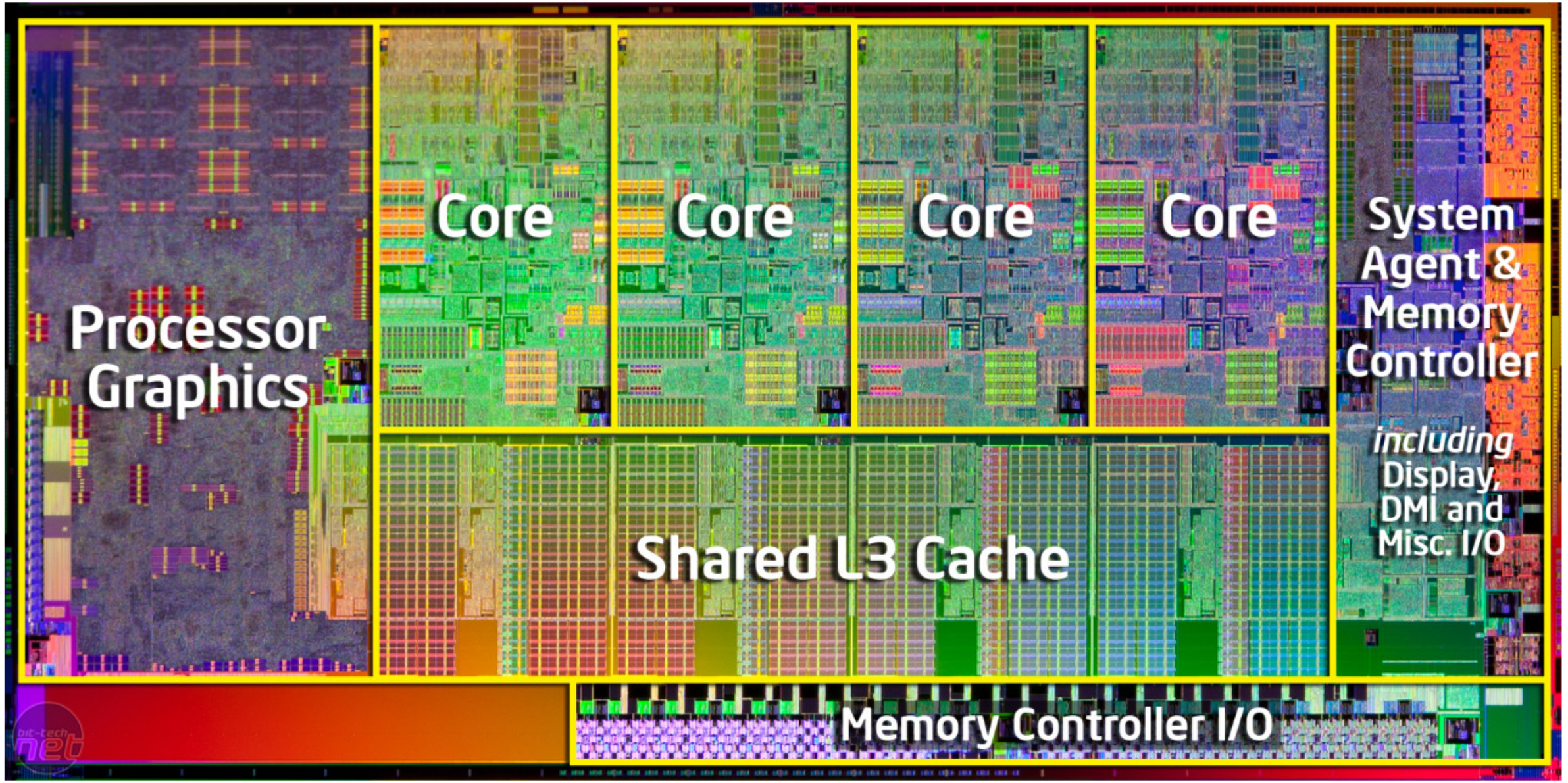
**Tomas Akenine-Möller**
**Department of Computer Science, Lund University**
**Advanced Rendering Technology, Visual & Parallel Computing, Intel**
**2014-11-27**


LUND UNIVERSITY

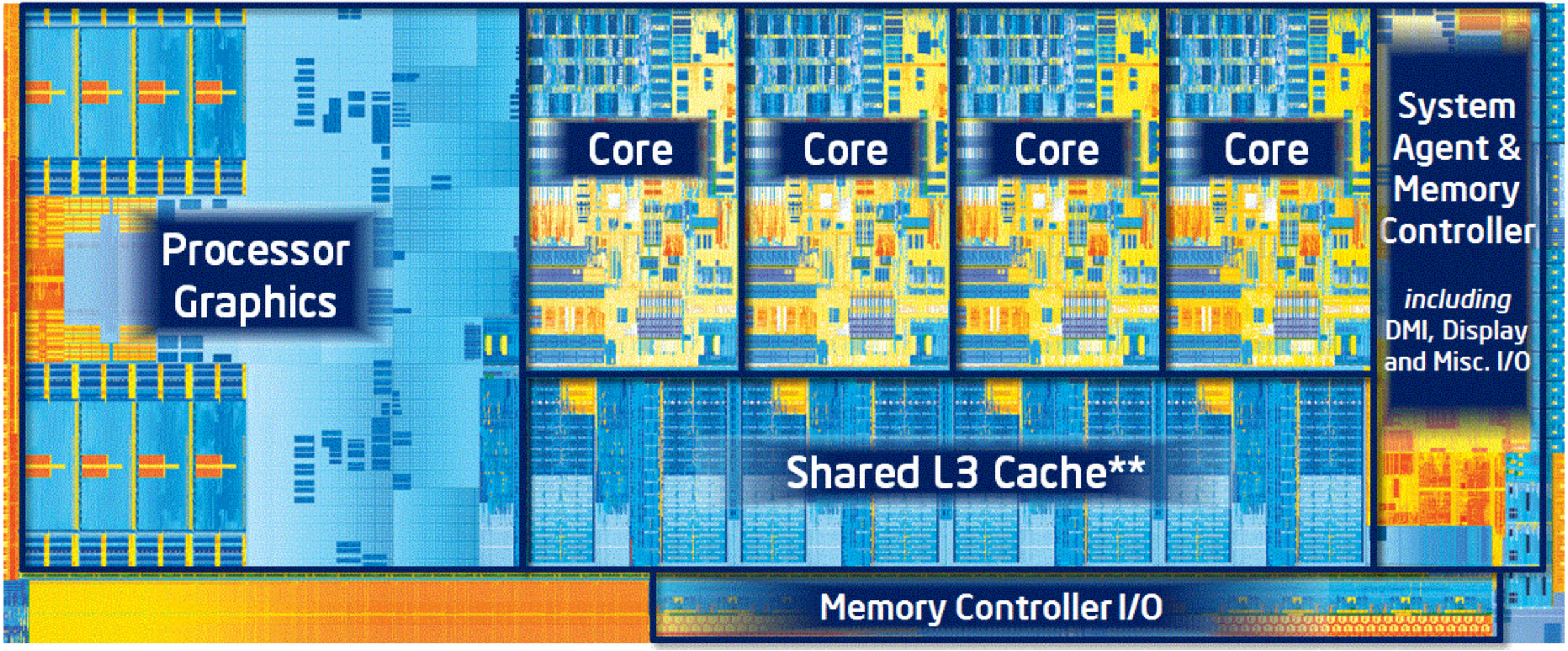
# Who am I?

- MSc in Computer Science & Engineering, LTH
- PhD in Computer Engineering (Graphics, really), Chalmers
  - Per Stenström as research mentor
- Researcher at Ericsson Mobile Platforms (co-developed ETC1)
- Postdoc UC Berkeley, UC San Diego
- Moved to LTH in 2004
  - Professor in Computer Graphics, 2007, LTH
- Swiftfoot Graphics AB, CEO
  - Acquired by Intel in December, 2008
  - I work at Intel 50% of my time (tech lead) -- 7-person group + OTC (~50 ppl)
- When starting research group in Lund, lots of thinking
  - Start with real HW for graphics? Continue with “just” SW for graphics?
  - Somewhere in between?

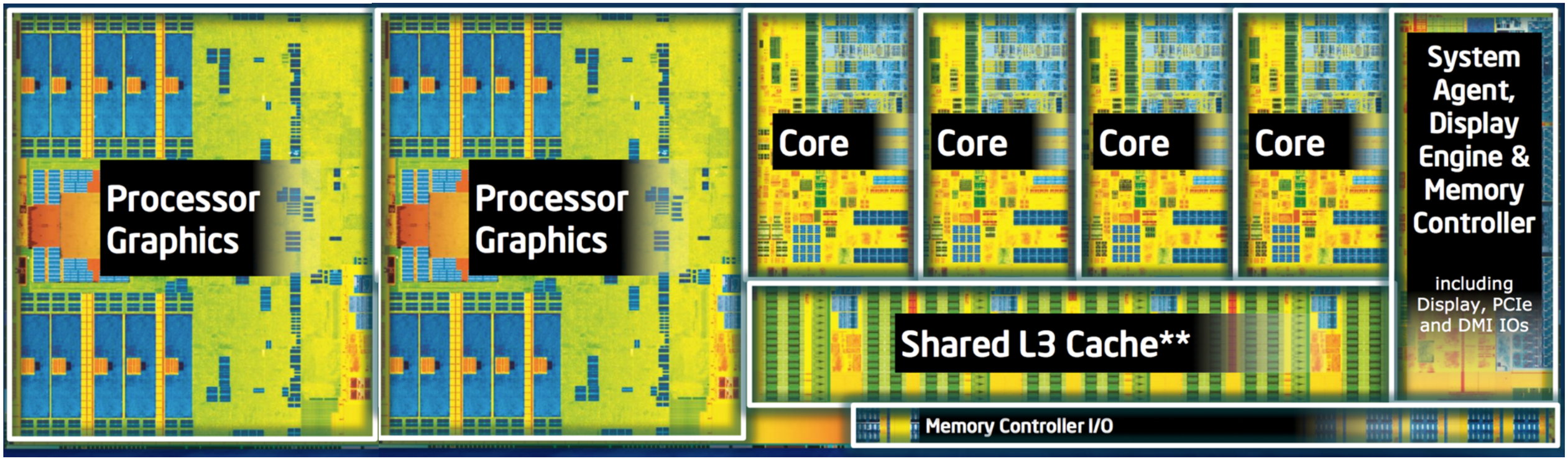
# Sandy Bridge, January 2011



# Ivy Bridge, April 2012



# Haswell, September 2012

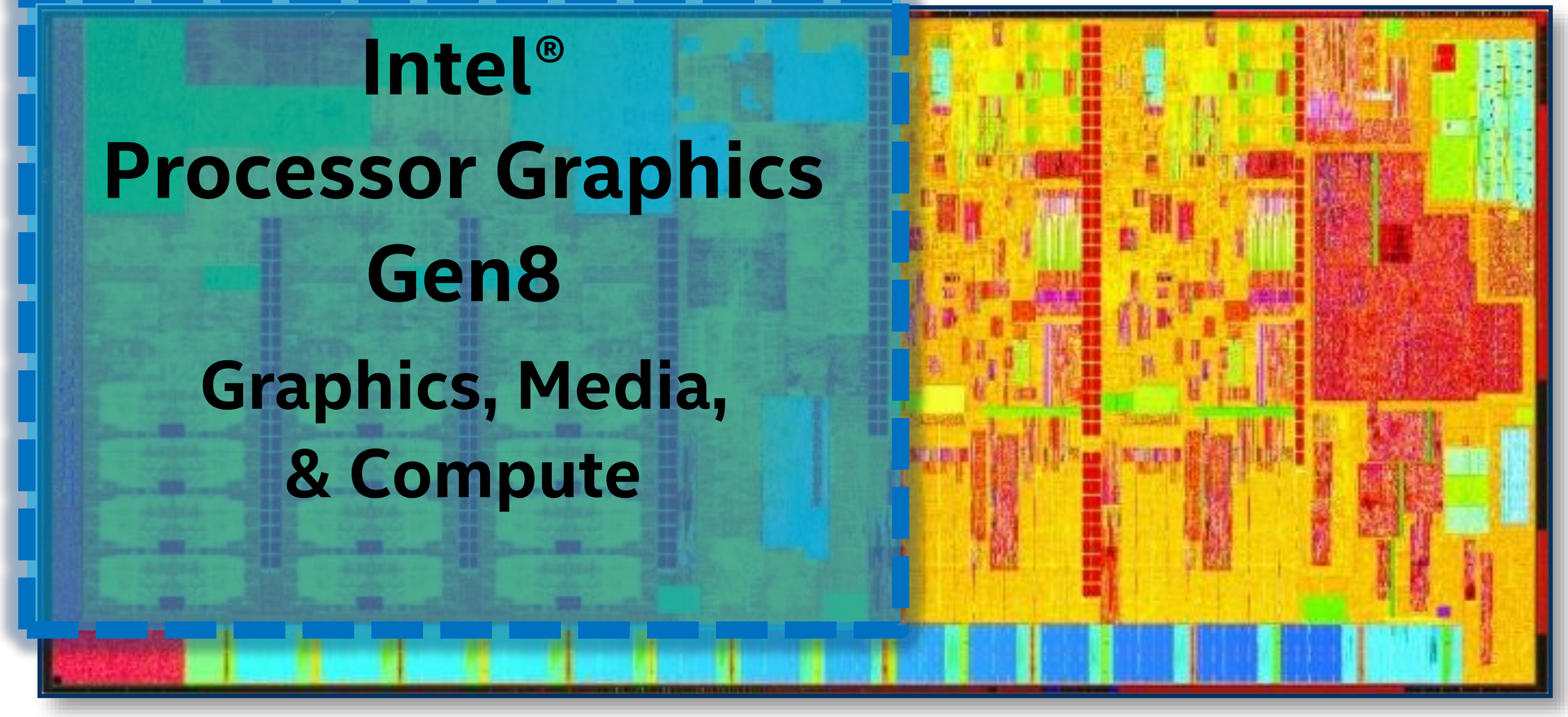



GT2

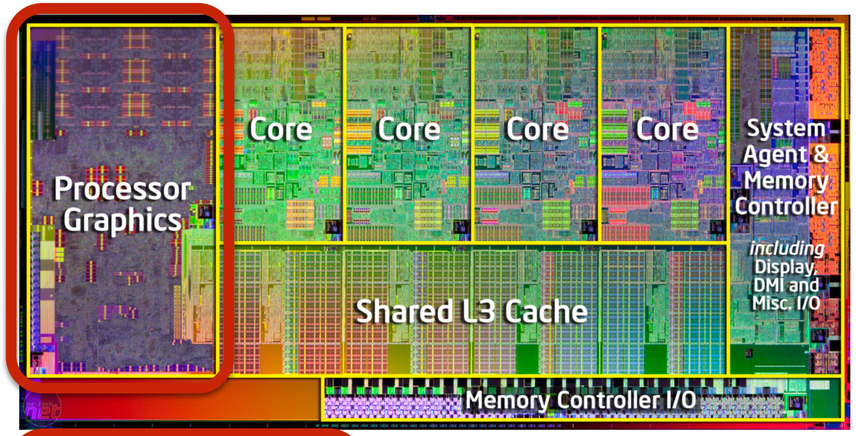
GT3

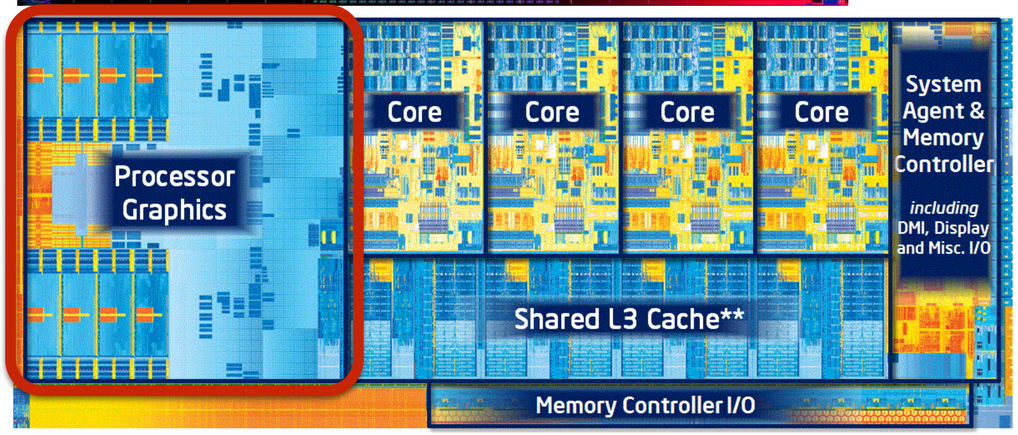
# Broadwell, fall 2014

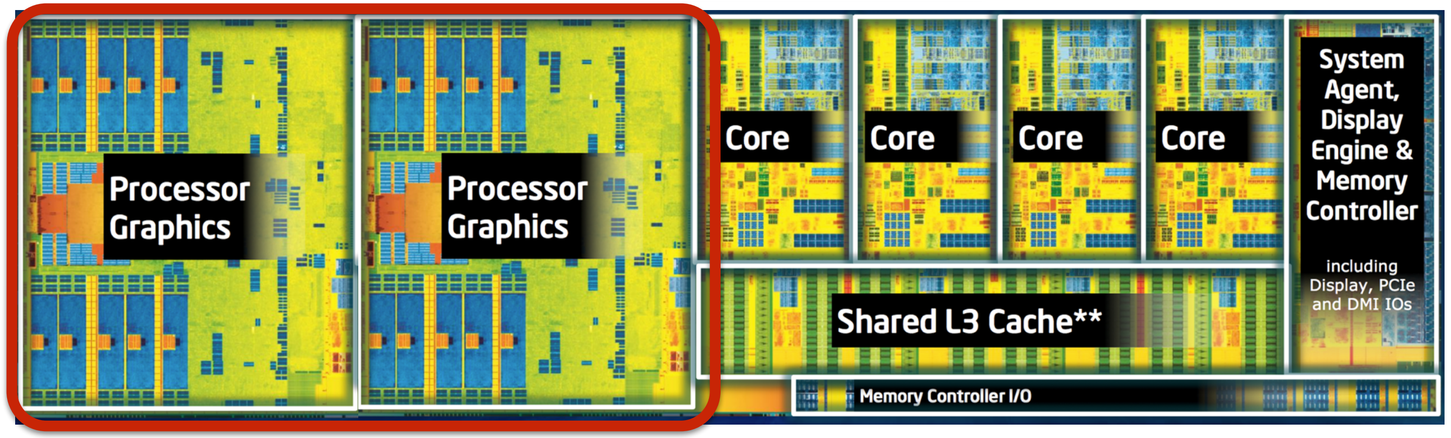
## Intel® Core™ M:



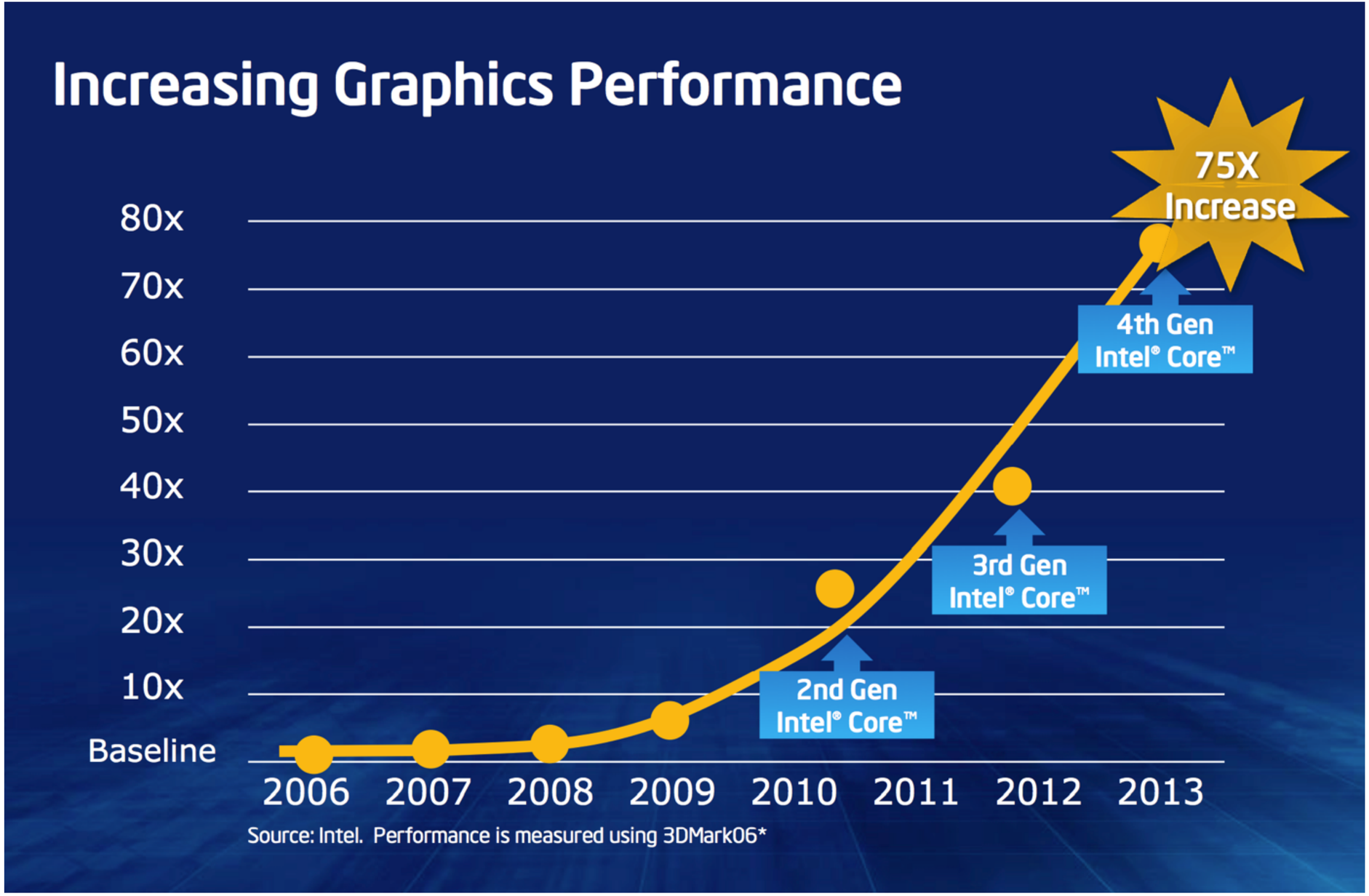
# Sandy, Ivy, Haswell




- Of course, cannot compare like this
  - Different process node, area, etc, but…
- …we can see where this is going
- However, must be able to scale down to tablets and phones as well!
  - Also, there are many different products based on these designs, so comparison may be even more off :(



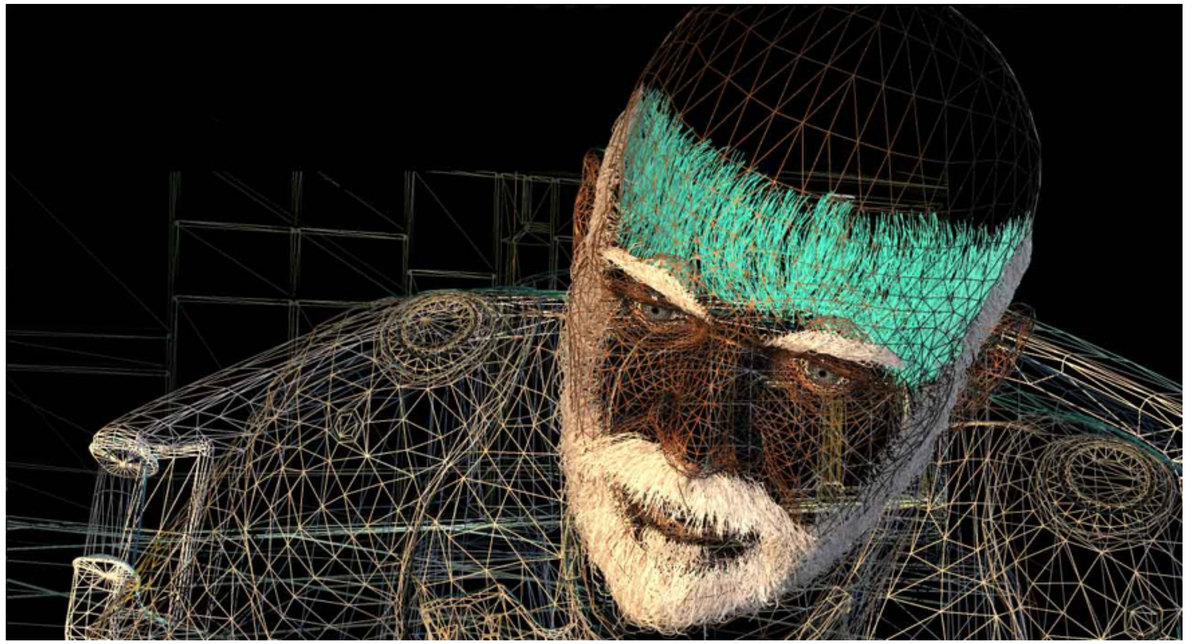

# Integrated graphics processor is a key compute resource today

# The Graphics Processor

- Made for graphics:
  - Fixed-function units for graphics
    - Rasterization of triangles
    - Texture sampler
    - …and more!
  - But also, lots of general compute units


The Division, Ubisoft, 2014

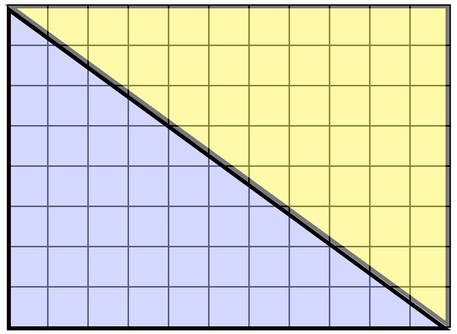
# Drawing triangles...



Images from 2002

- Everything is made of triangles (or lines and points)
- For all pixels inside a triangle
  - Execute a user-defined program (**pixel shader**)
  - Not dependent on other pixels
    - Except possibly on the result from earlier triangles covering the same pixel

# Drawing two triangles covering the screen



- Graphics processor executes a user-defined program per pixel
- So, isn't this just a double for-loop?

```
for (int y=0; y < height; y++)
{
      for (int x=0; x < width; x++)
      {
          pixelShader(x,y);
      }
}
```

# Example pixel shader: 3x3 box filter

```
uniform sampler2D tx;  // the texture
varying vec2 txCoord;  // texture coordinates

void main(void)
{
    vec4 sum = vec4(0.0);    // 4 components

    for (int x = -1 ; x <= 1; x++)
        for (int y = -1; y <= 1; y++)
            sum += texture2D(tx, vec2(txCoord.x + x, txCoord.y + y));

    gl_FragColor = sum * (1.0 / 9.0); // output for pixel shader
}
```

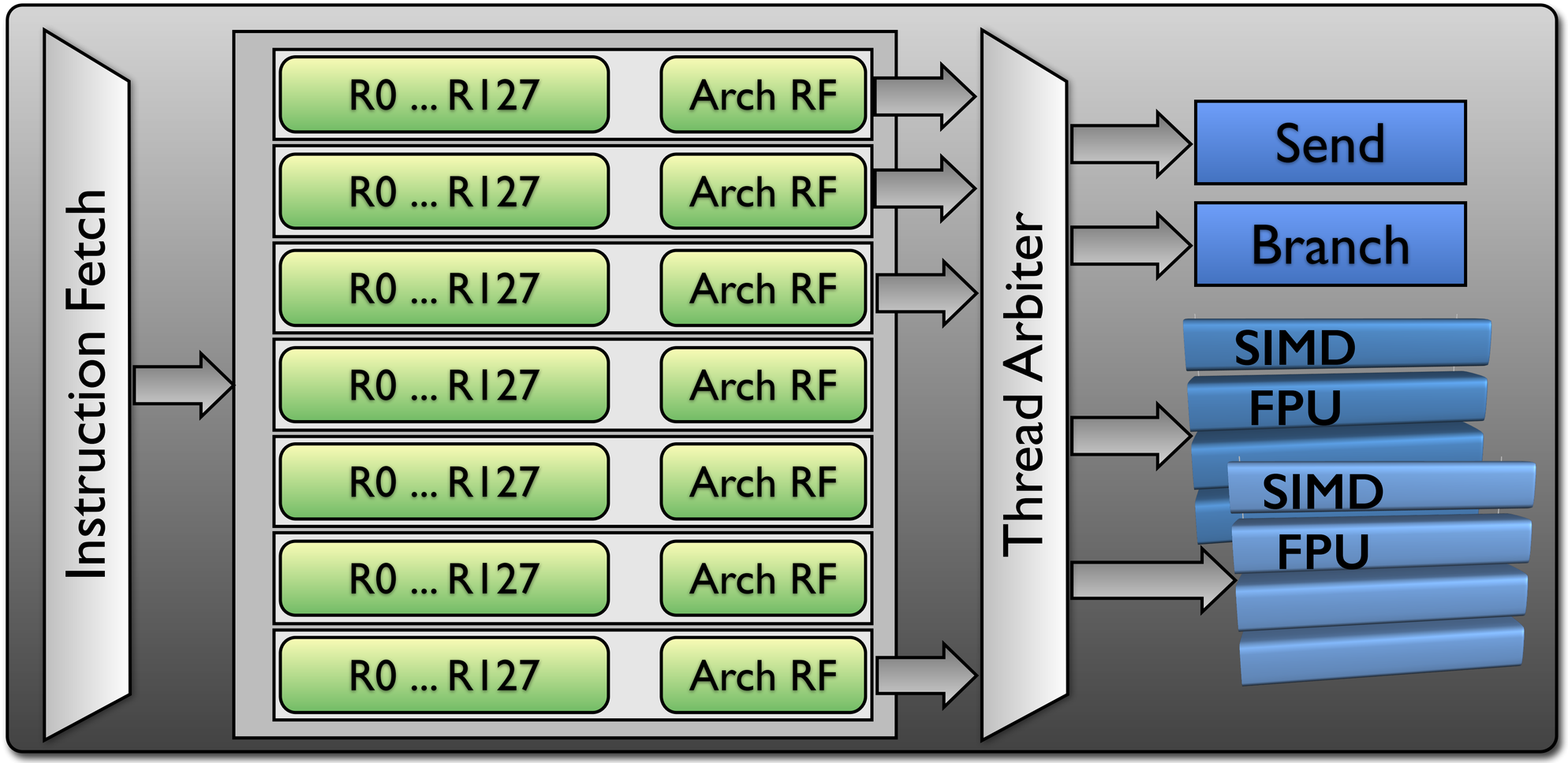
# Intel Gen8 graphics architecture

- Much of my talk will explain “Generation 8”
- Starting to appear in Broadwell products (14 nm)
- Scalable architecture
- Lots of parameters may change from product to product
- Lots of improvements per generation

- At Intel, I work on the following topics:
  - Reducing memory traffic using compression & caching
  - Reducing triangles to render using culling
  - Long term research, such as stochastic rasterization and ray tracing
    - Check out our publications to learn more about this

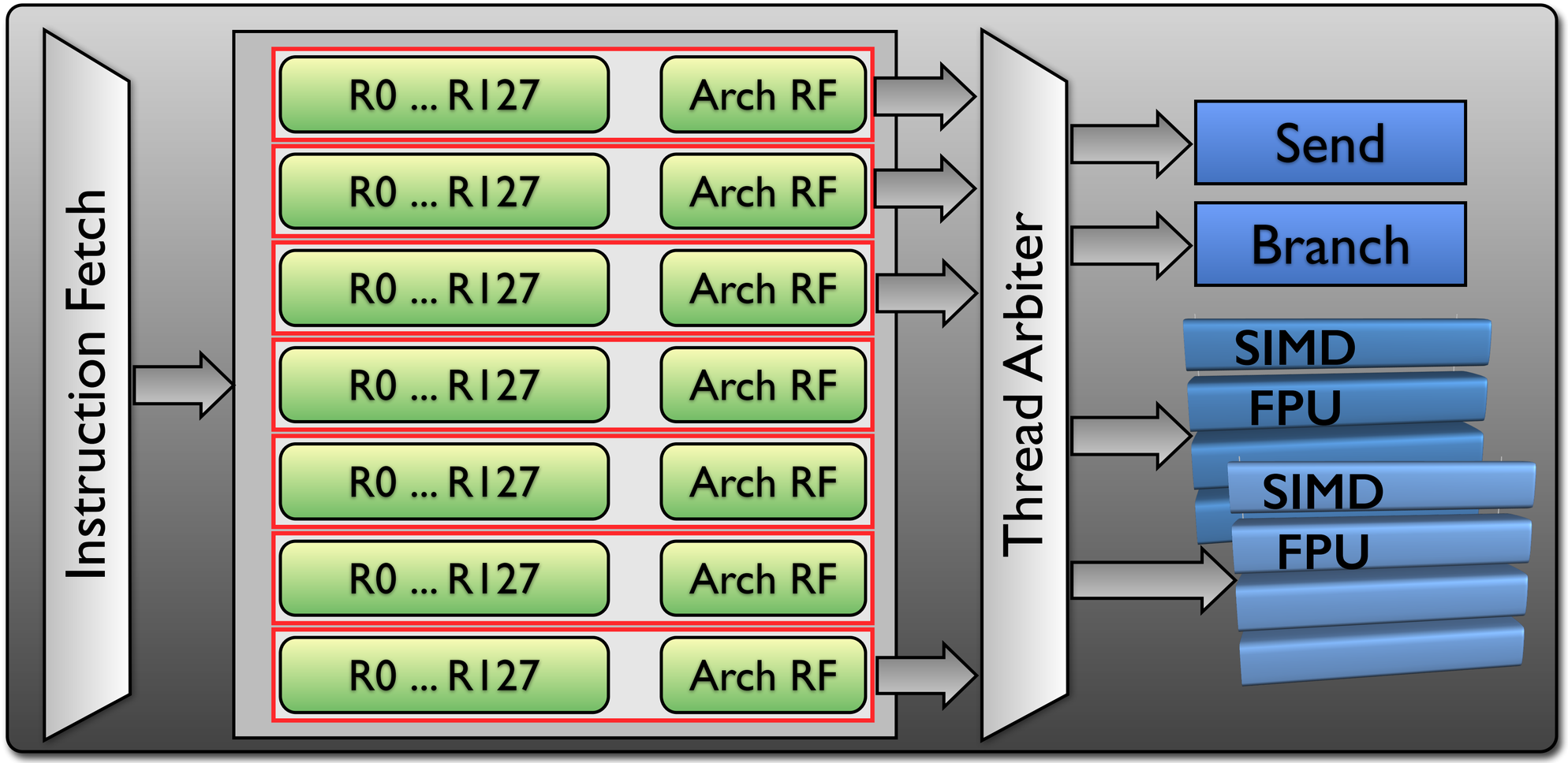
# The EU: Execution Unit

- The workhorse in GEN



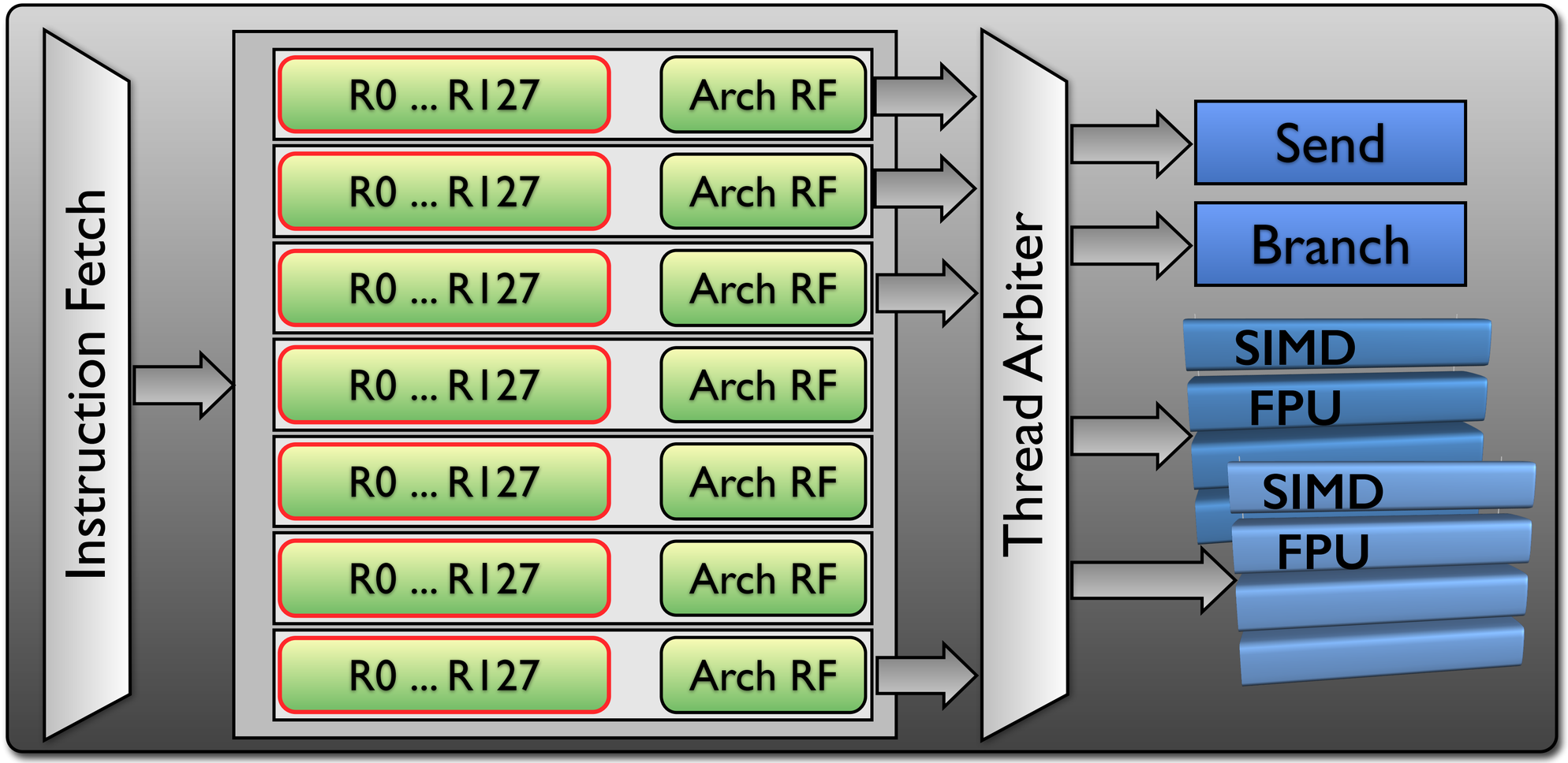
# The EU: Execution Unit

- Seven HW threads per EU
  - Zero cycle thread switching
- Threads can execute different instances of the same kernel
- ...but can also execute different kernels



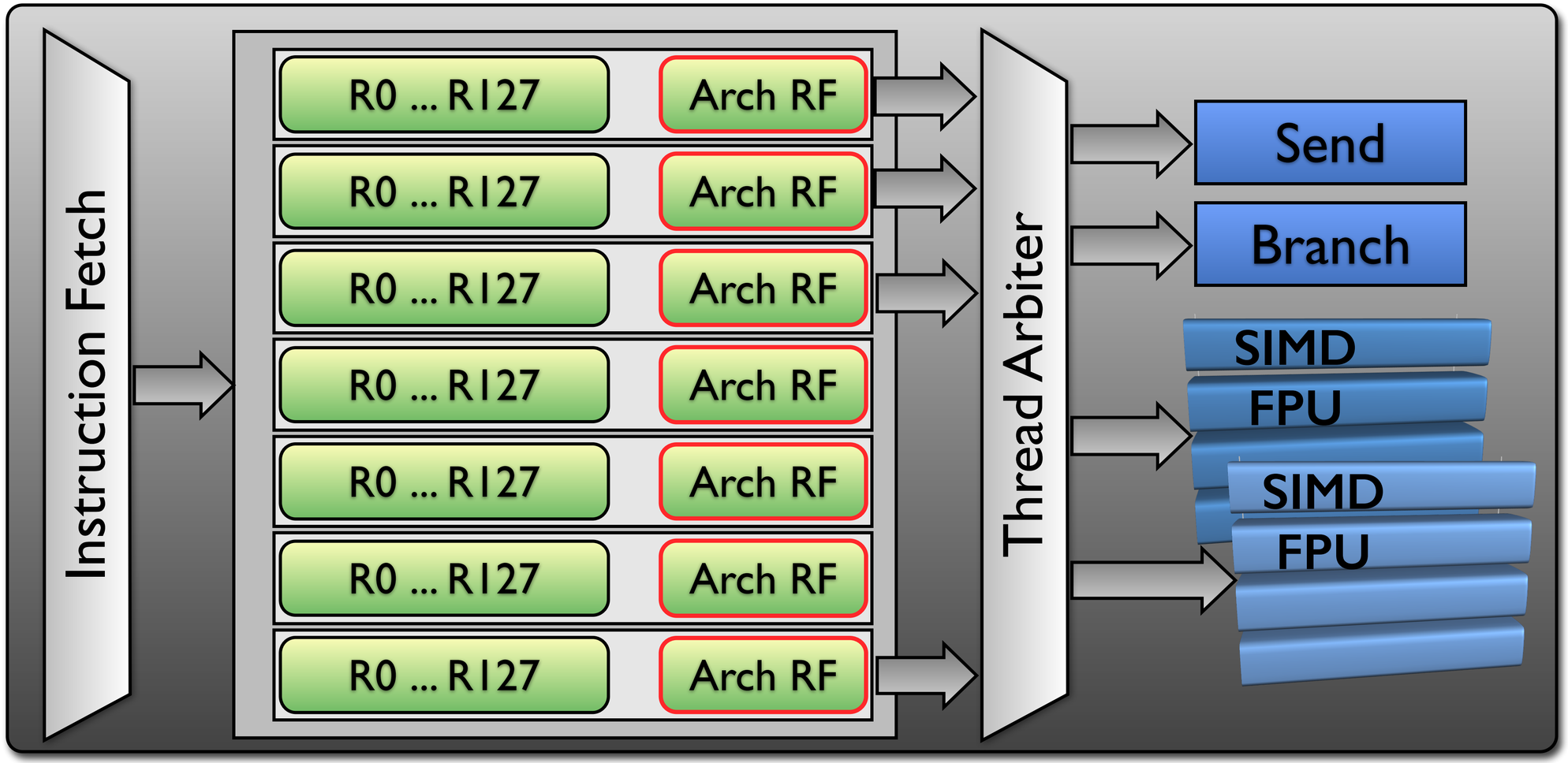
# The EU: Execution Unit

- 128 registers per thread. 1 register = SIMD8x32b = 32B
  - Can use as four 64b doubles, eight 32b ints/floats, or sixteen 16b shorts/half-floats
- 32B x 128 = 4kB for registers per thread
- 4kB x 7 = 28kB per EU for the register file



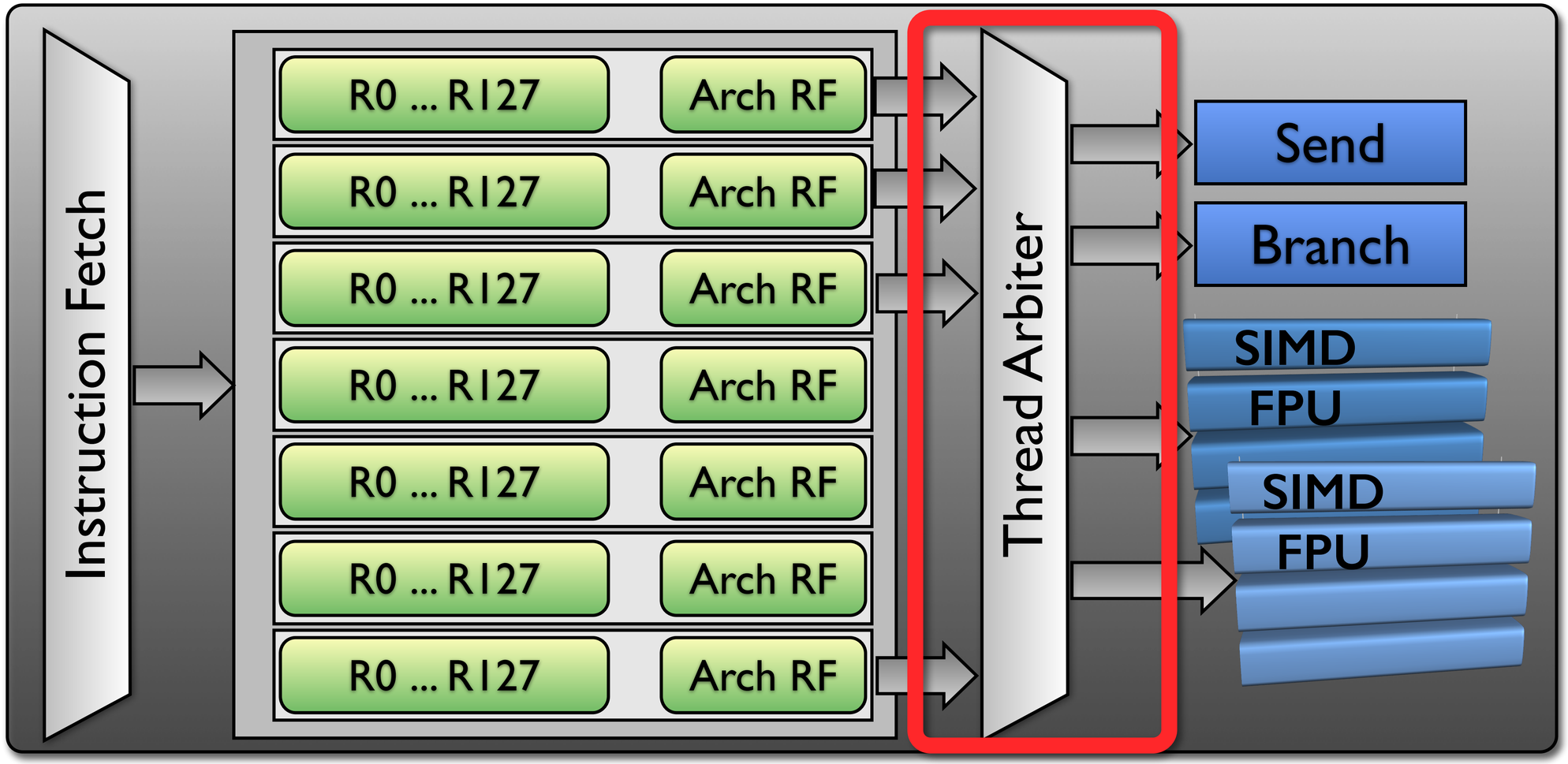
# The EU: Execution Unit

- Architecture registers per thread
  - Program counters, accumulators, index & predicate registers



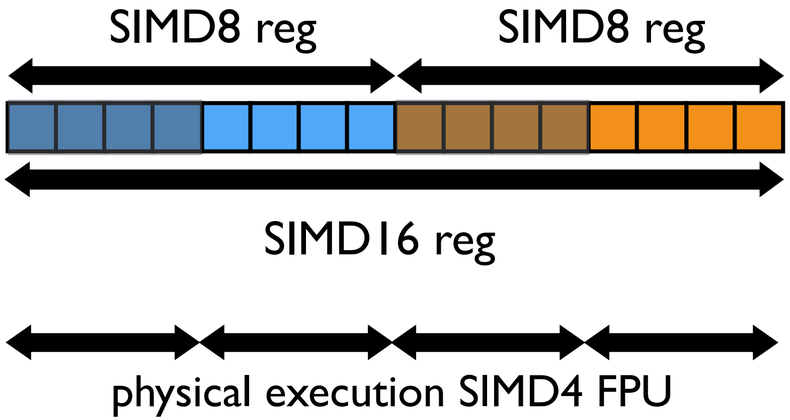
# The EU: Execution Unit

- Simultaneous multi-threading (SMT) instruction dispatcher
- Thread arbiter picks instructions from runnable threads
- Dispatches to functional units (Send, Branch, FPUs)
- Can co-issue instructions from up to four threads **each cycle.** Possible to saturate FP throughput using 2 threads



# Instructions

- 2 or 3 src register and 1 dst register
- Instructions are **variable** width SIMD
- To optimize register footprint & compute density, Gen is logically programmable as $2^n$-wide SIMD
  - *n* can be 0-5, i.e., 1, 2, 4, 8, 16, and 32 wide SIMD
  - SIMD width can change at any time without penalty
- However, physically, FPU is 4-wide SIMD, 32-bit lanes

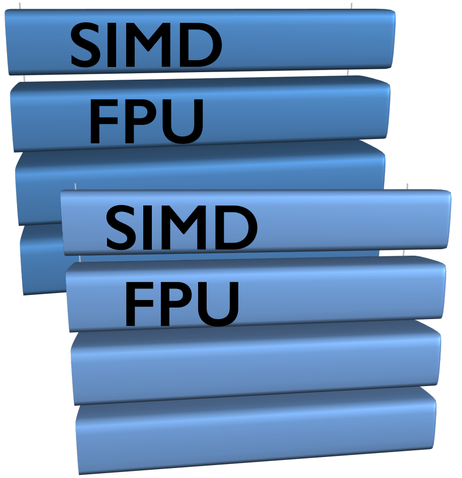


# Send


- **Send** = I/O instruction for Gen
- Memory read/writes
- Scatters/gathers
- Texture sampling per SIMD lane
- Synchronization, atomic operations, fences, etc.

- Note: when data is not in cache, the thread does **not stall** the machine
  - Rather, another thread is switched in
  - Perfect for hiding latency of memory fetches etc.

# FPU




- Minimum latency for instructions is 2 clocks
- 2 clocks for SIMD 1,2,4,8-float ops
- 4 clocks for SIMD 16 float ops
- 8 clocks for SIMD 32 float ops
- FPU is fully pipelined across  threads
  - Instructions complete every cycle
- 2x int performance in Gen8 compared to Gen 7.5
- Math support:
  - Int/fp divide, quotient, remainders
  - Reciprocals, sqrt, rsqrt
  - sin, cos, exp, pow, log,...
    - Piecewise linear approximation for transcendental functions
- Also, native 16-bit floating point support (2x vs fp32)
- Only one of the FPUs can do double precision math

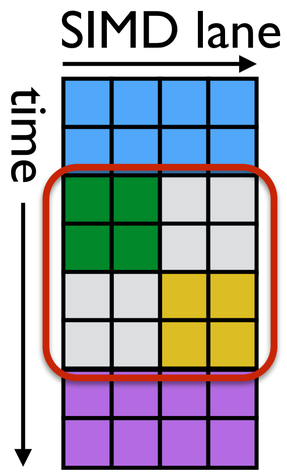
# Branch

Branch

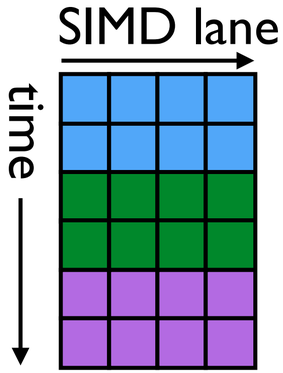
- Scalar control flow
  - Program counter per HW thread
- SIMD control flow
  - PC per SIMD lane
    - if {} else {}
    - while
    - call / return
    - goto / join
- Every instruction can be predicated
  - Enable/disable individual SIMD lanes

```
x=detta();
if(x)
    ditten();
else
    datten();
drutten();
```

**x sometimes true:**




**x always true:**



# Theoretical EU peak capacity

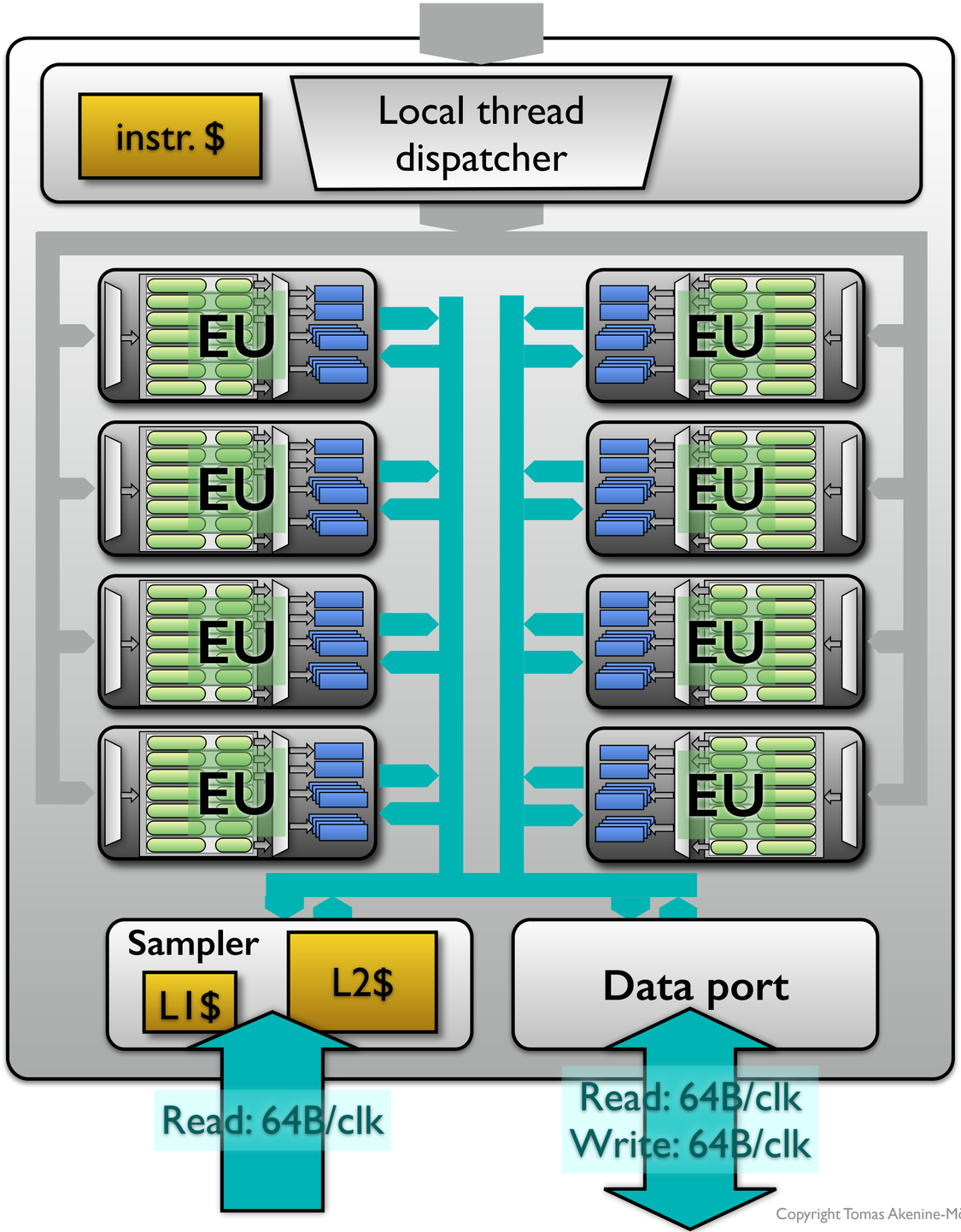
- 2 FPUS  per EU
- **MAD** = fused **M**ul + **AD**d per clock
- Physical SIMD width = 4
- 2 x 2 x 4 = 16 floating-point operations per clock
- My laptop (47W) has a 1.3 GHz Gen 7.5

**1.3 x 16 = 20.8 GFLOPS per EU**

- So, how do we scale up towards **1 TFLOPS**? 50 EUs? How?

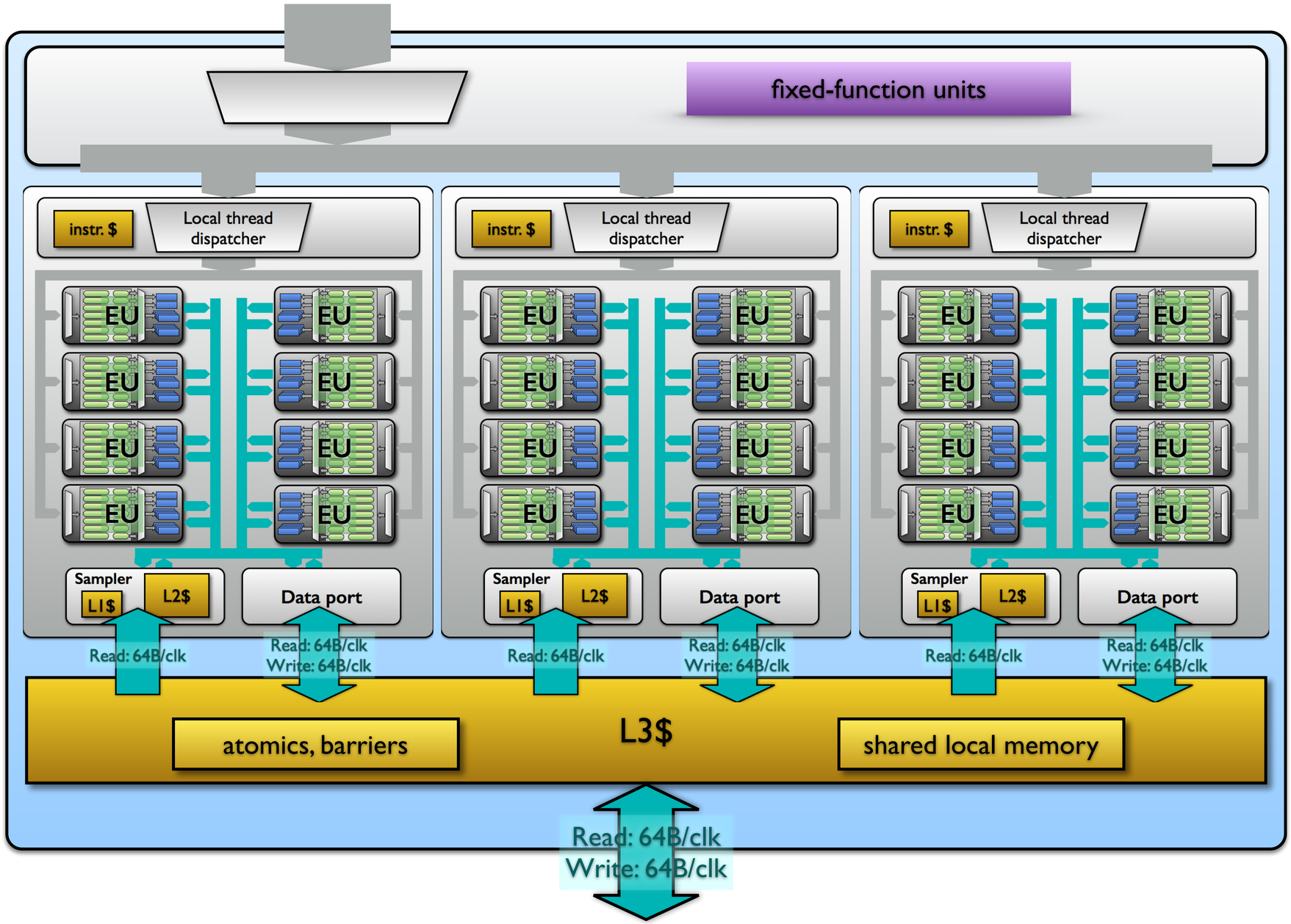
# Subslice

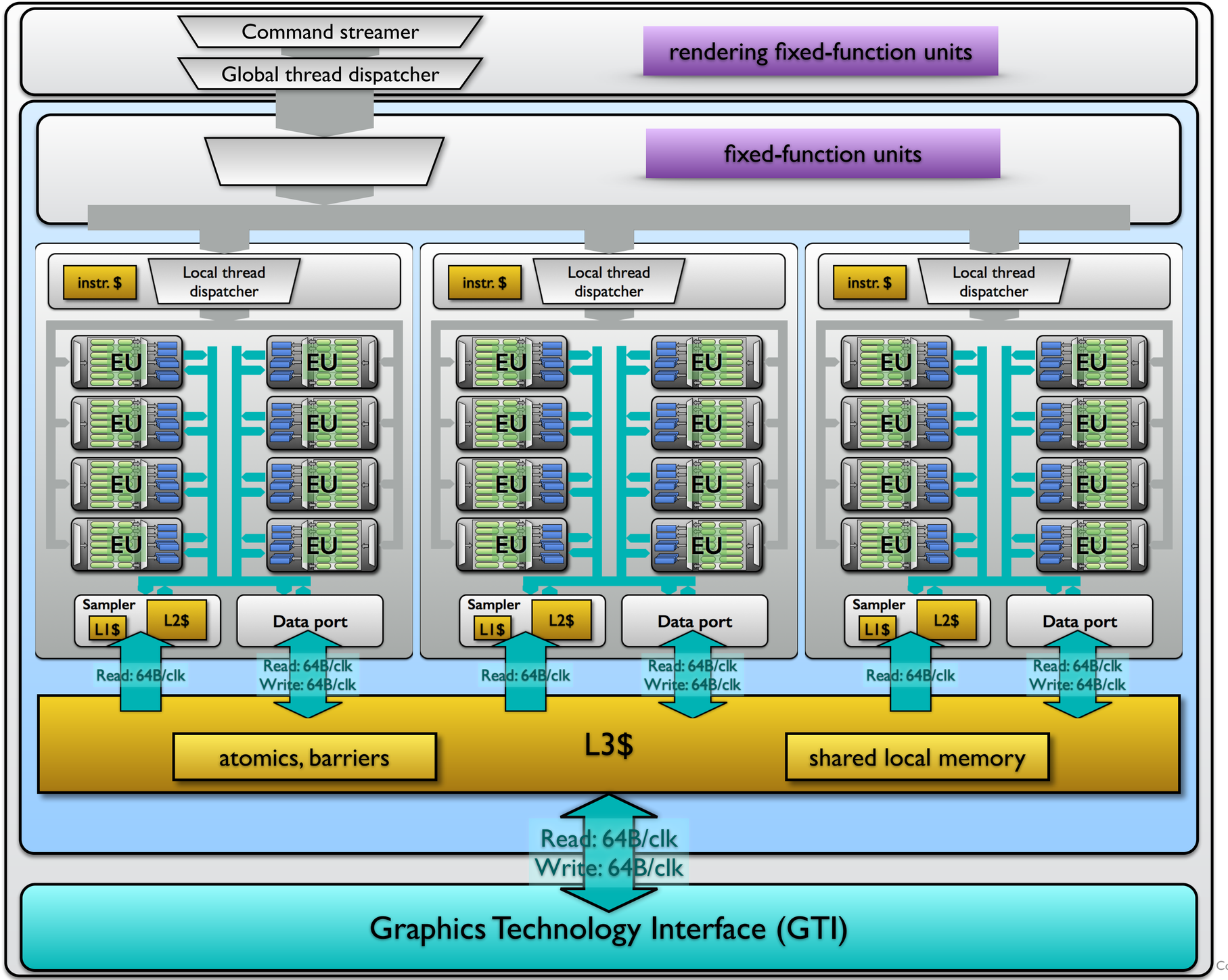
- Array of EUs
  - Here, 8 EUs: 20.8 x 8 = **166.4** GFLOPS
- Plus:
  - Local thread dispatcher & instr $
  - Texture sampler + L1 + L2
    - Decompression, filtering, addressing
  - Data port: general purpose load/store
    - Dynamically coalesces scattered memory operations into fewer ops
- 7 x 8 = 56 HW threads



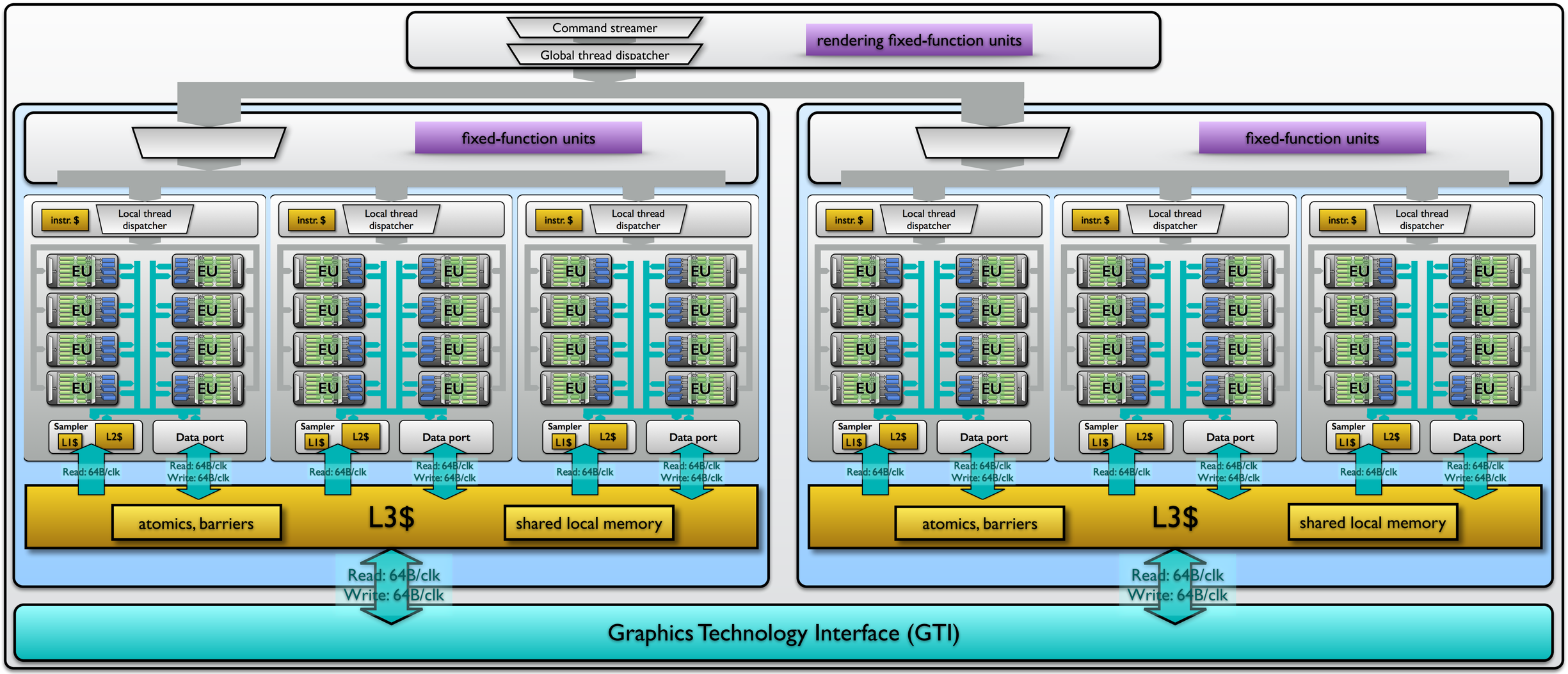
# Slice

- 3 subslices
  - 3 x 8 = 24 EUs
  - 3 x 20.8 x 8 = **499.2** GFLOPS
- L3$
  - Subslices have dedicated interface to L3
  - 64B $line size
  - Monolithic, but distributed
  - L3$ = 384-768kB/slice, for example
- 7 x 8 x 3= 168 HW threads per slice
- Shared local memory 64kB/slice
  - Higher banking than rest of L3
- HW barriers, 32b atomics





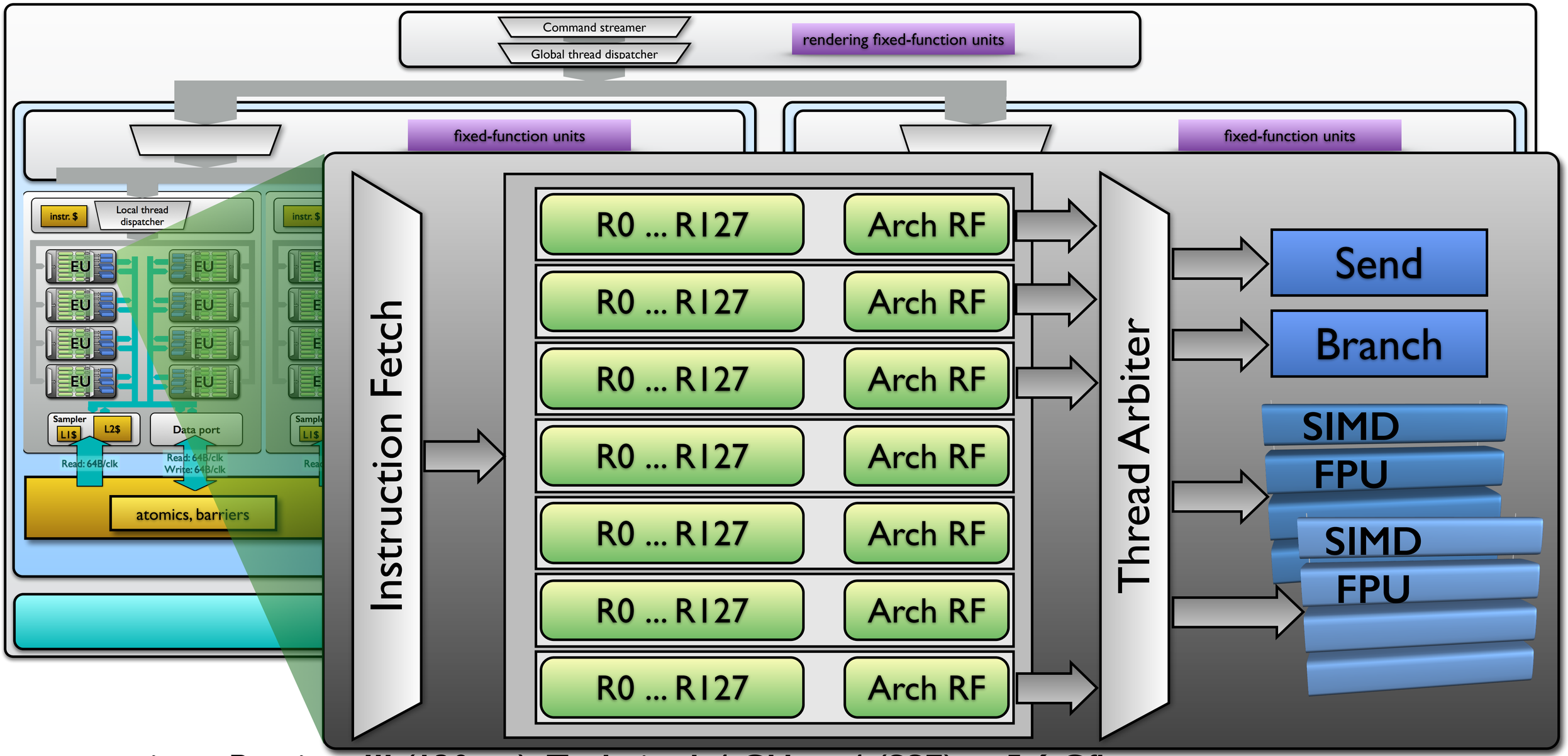
# Another example configuration




- Peak compute: 20.8 x 8 x 6 = **998.4 GFLOPS ≈ 1 TFLOPS**
- My laptop: 40 EUs x 4-wide-SIMD x 2 FPUs x 2 (MAD) x 1.3 GHz = **832 GFLOPS**
  - The four CPU cores in my laptop: **332.8 / 460.8 GFLOPS** (without/with Turbo)

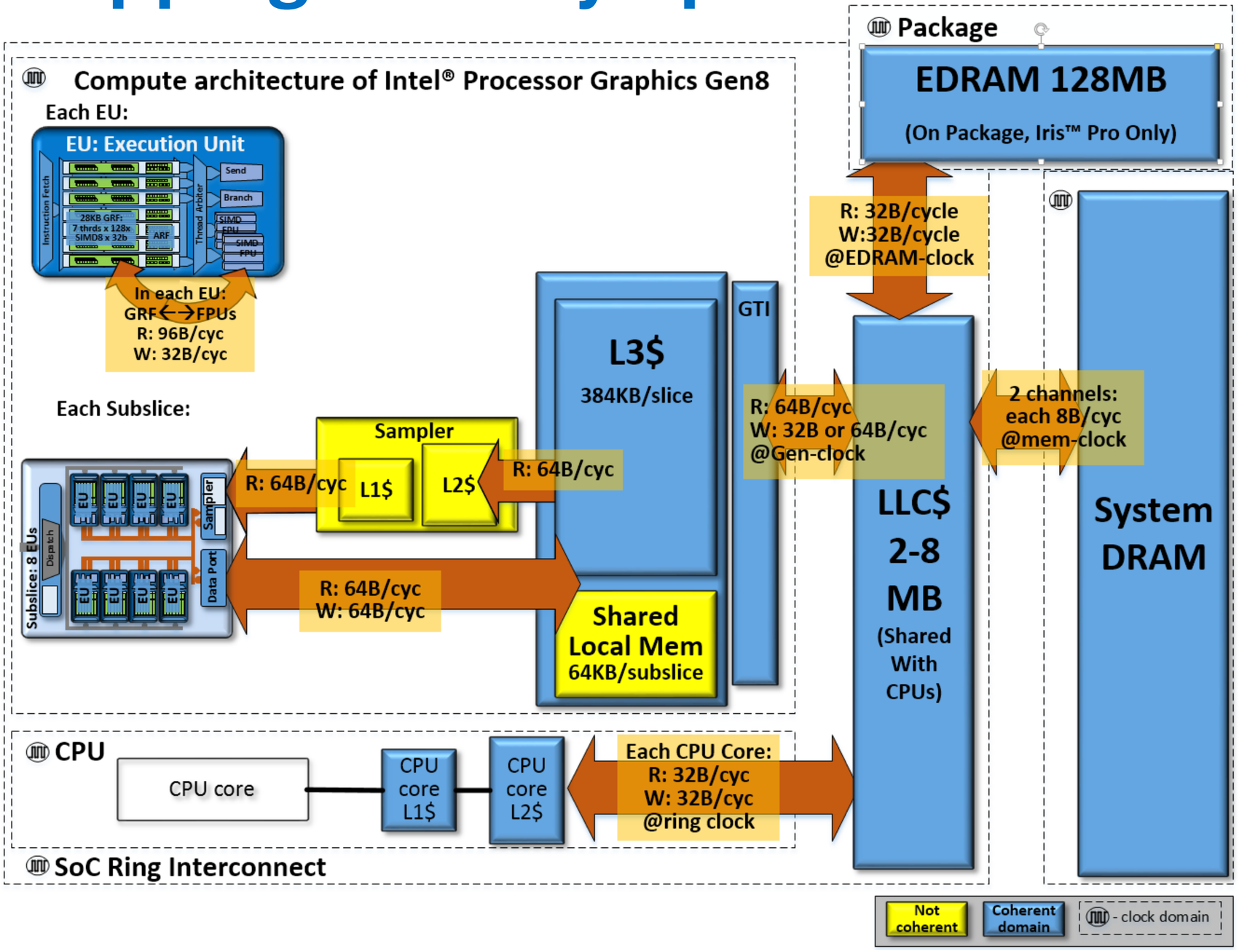
Challenge to exploit both CPU and GPU resources

# Back to the EU




For comparison: Pentium III (130nm), Tualatin, 1.4 GHz x 4 (SSE) = 5.6 Gflops

# Mapping Memory Spaces to Memory Hierarchy




**EDRAM details for Iris Pro 5200:**
51.2 GB/s x 2
Hitrate often > 95% (AnandTech)

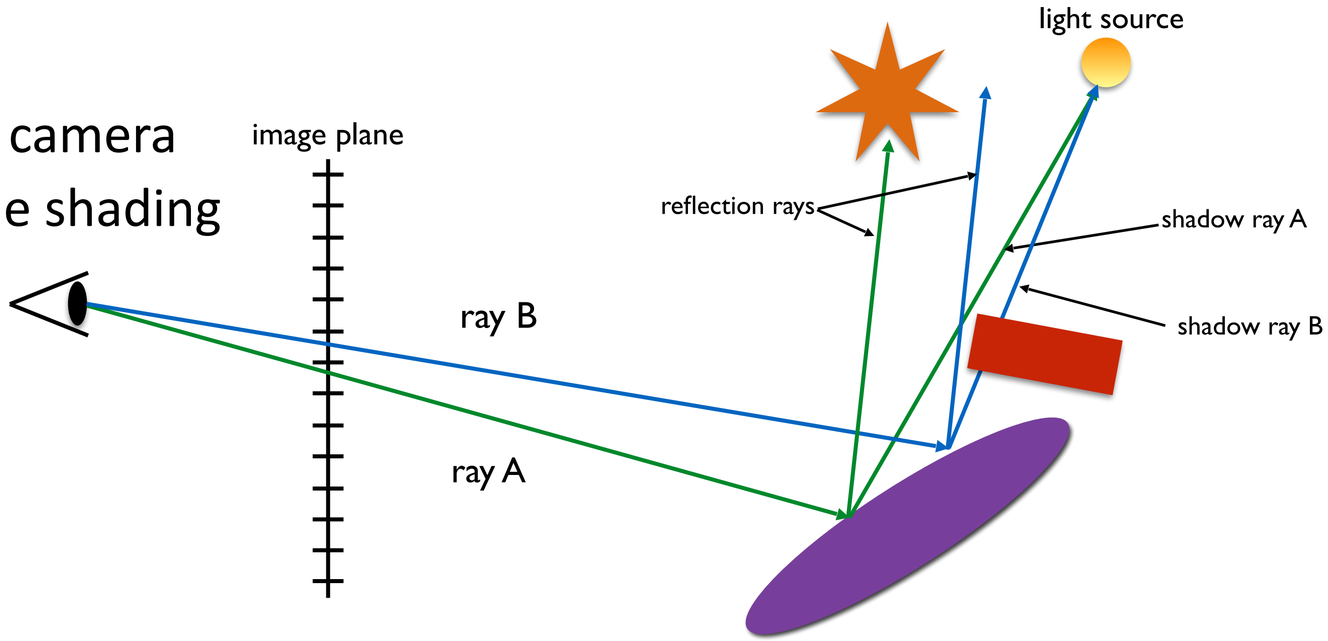
# Shared Virtual Memory

- Gen7.5 has shared ***physical*** memory
  - Can do “zero copy” from CPU to GPU and vice versa
  - But cannot share data structures with pointers in them
    - Needs to be done using indices
- Gen8 has shared ***virtual*** memory
  - Seamless pointer sharing enabled between GPU and CPU
  - Hardware-supported byte-level CPU & GPU coherency
  - Fine-grained sharing
  - New feature in OpenCL 2.0
  - To increase programmer productivity

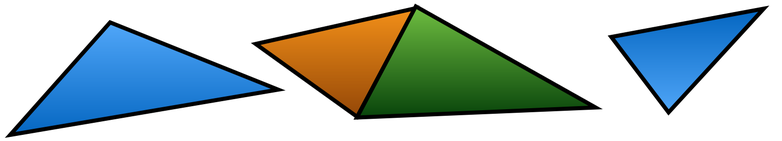
# Example Application

# Ray Tracing

- My laptop: Macbook Pro, Haswell Core i7 (4960HQ)
  - **332.8/460.8** Gflops (without/with Turbo, 2.6/3.6 GHz)
  - **832** Gflops on the GT3e (graphics processor)
- Our goal in this research project:
  - Attempt to exploit all this compute for a ray tracer
- Ray tracing:
  - Follow rays through a virtual camera
  - Shoot more rays to determine shading
  - Recursively
  - If careful, can get images with **global illumination**



# 3D scene representation?

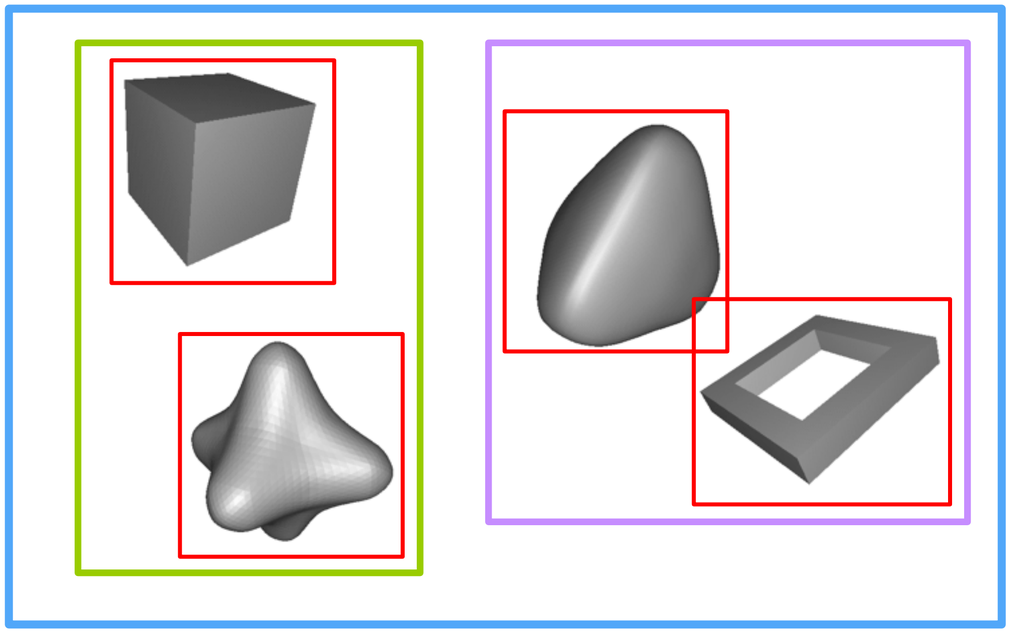


- Usually triangles, even for ray tracing
- What to do if your scene has 10 million triangles?
  - Test a ray against every triangle and keep only the closest intersection?

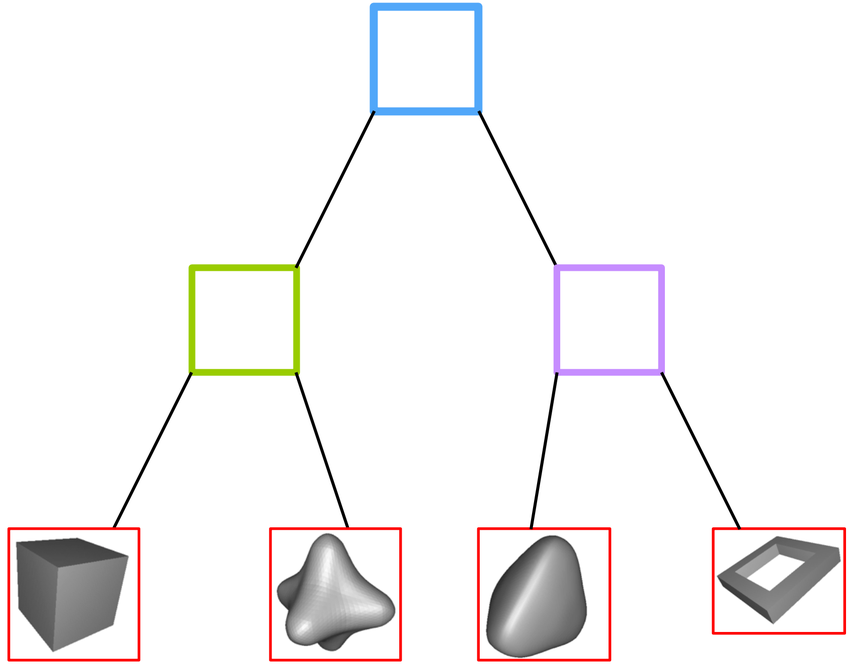
- Not gonna work…
- Turn instead to some kind of data structure
  - Analogue: you can use a heap to search for keys
- For graphics/geometry, we <u>always</u> exploit **hierarchical** data structures
- There are many of those:
  - Hierarchical grids
  - Octrees
  - kD-trees
  - Bounding volume hierarchies (BVHs)

# The bounding volume hierarchy

**The real 3D world:**

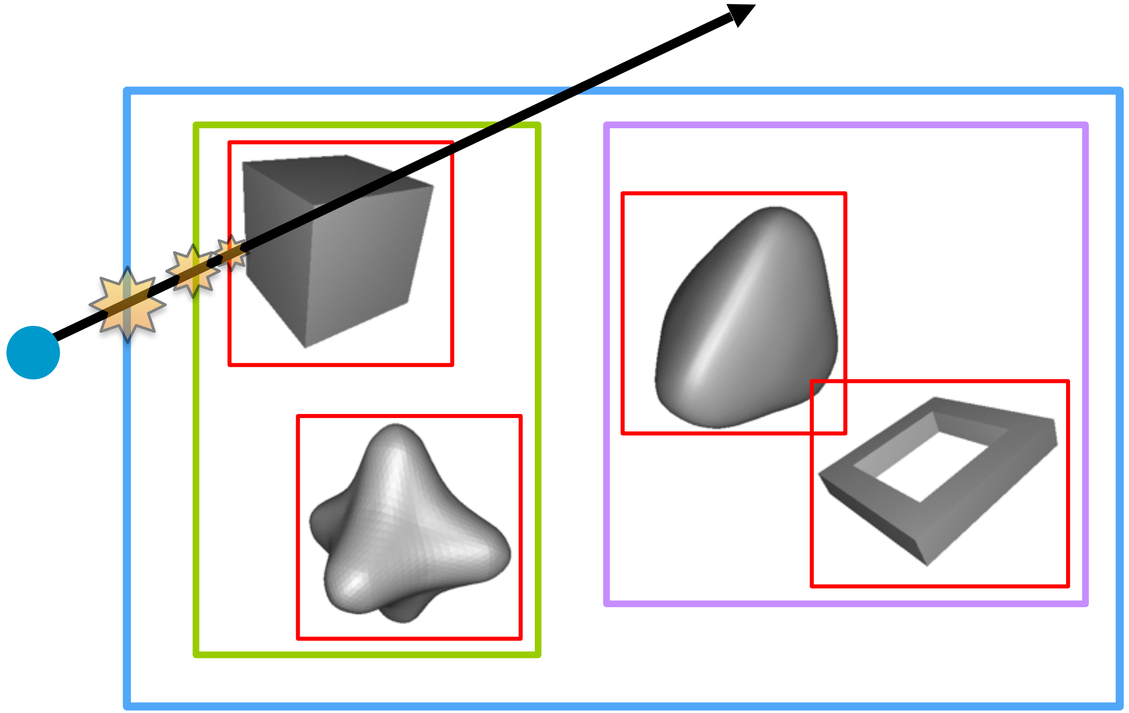


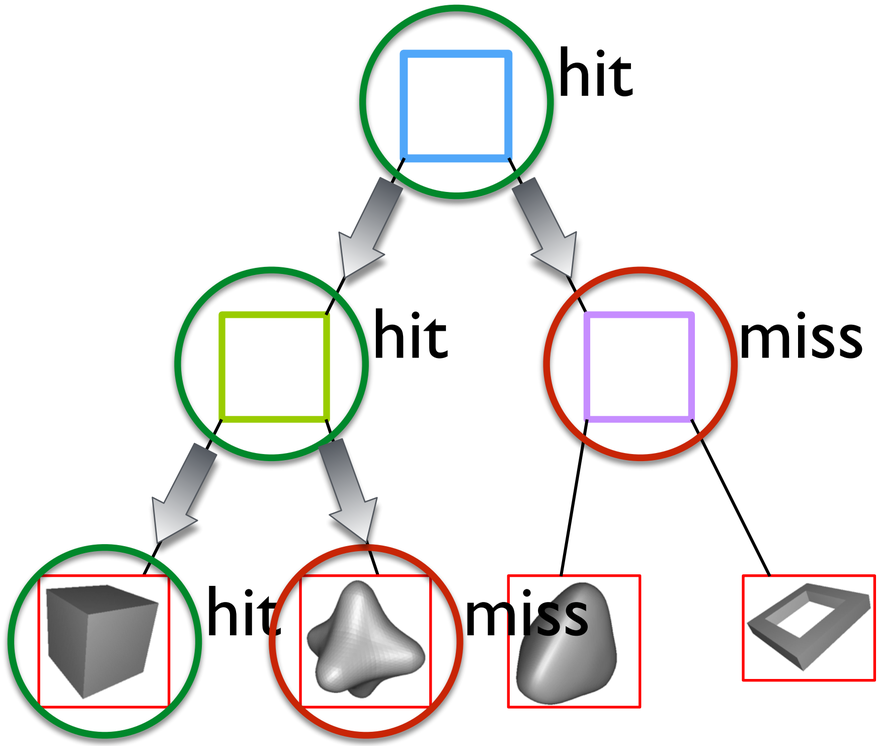
**The data structure:**



# How to use the BVH?

- Start from the root: test ray against box, if hit traverse to children






- Leaves (usually) contain 1-8 triangles
- Hierarchical spatial data structures typically gives $\mathbf{O(\log n)}$ instead of $\mathbf{O(n)}$, where $n$ is the number of triangles in the entire scene (*per ray*)

# Some code

## tree traversal

```
void traversal(float8* rays, nodePtr nodes, float4* triangles, float4* hits, int count) {
   int rayIndex = get_global_id(0);
   if (rayIndex >= count) return;
   float8 ray = rays[rayIndex];
   const float epsilon = 1e-10f;
   if (fabs(rayDirection(ray).x) < epsilon) rayDirection(ray).x = copysign(epsilon, rayDirection(ray).x);
   if (fabs(rayDirection(ray).y) < epsilon) rayDirection(ray).y = copysign(epsilon, rayDirection(ray).y);
   if (fabs(rayDirection(ray).z) < epsilon) rayDirection(ray).z = copysign(epsilon, rayDirection(ray).z);
   float3 invDir = make_float3(1.0f/rayDirection(ray).x, 1.0f/rayDirection(ray).y, 1.0f/rayDirection(ray).z);
   float3 OoD = -rayOrigin(ray) * invDir;
   float4 hit = make_float4(as_float(-1), rayTFar(ray), 0.0f, 0.0f);
   int node = 0x80000000, stack[64], *stackHead = stack, cnt = 0;

   for (;;) {
    ++cnt;
    if (node & 0x80000000) {
         node &= ~0x80000000;
         float4 d0 = readNode4(node, 0, 4);
         int first = as_int(d0.z);
         int last = as_int(d0.w);
         float4 d1 = readNode4(node, 1, 4);
         float4 d2 = readNode4(node, 2, 4);
         float3 firstMin = d1.xyz;
         float3 firstMax = make_float3(d1.w, d2.x, d2.y);
         float tRay = rayTFar(ray);
         float tFirst = aabbIntersect(firstMin, firstMax, ray, invDir, OoD);
         float4 d3 = readNode4(node, 3, 4);
         float3 lastMin = make_float3(d2.z, d2.w, d3.x);
         float3 lastMax = make_float3(d3.y, d3.z, d3.w);
         float tLast = aabbIntersect(lastMin, lastMax, ray, invDir, OoD);
         if (tFirst < tLast) {
              if (tLast != tRay) *(stackHead++) = last;
              node = first;
              continue;
         }
         else if (tLast != tRay) {
              if (tFirst != tRay) *(stackHead++) = first;
              node = last;
              continue;
         }
    }
    else {
         int first = node & 0xffffff;
         int last = first + (node >> 24);
         for (int i = first; i < last; ++i) rayTFar(ray) = triangleIntersect(triangles, i, ray, &hit);
    }
    if (stackHead == stack) break;
    node = *(--stackHead);
   }
   hits[rayIndex] = hit;
}
```

ray setup

internal node?

read right child's box

test right box against ray

read left child's box

test left box against ray

traverse to closest node, push other to stack

leaf, test triangles

MAD=fused mul+add

## ray/box intersection

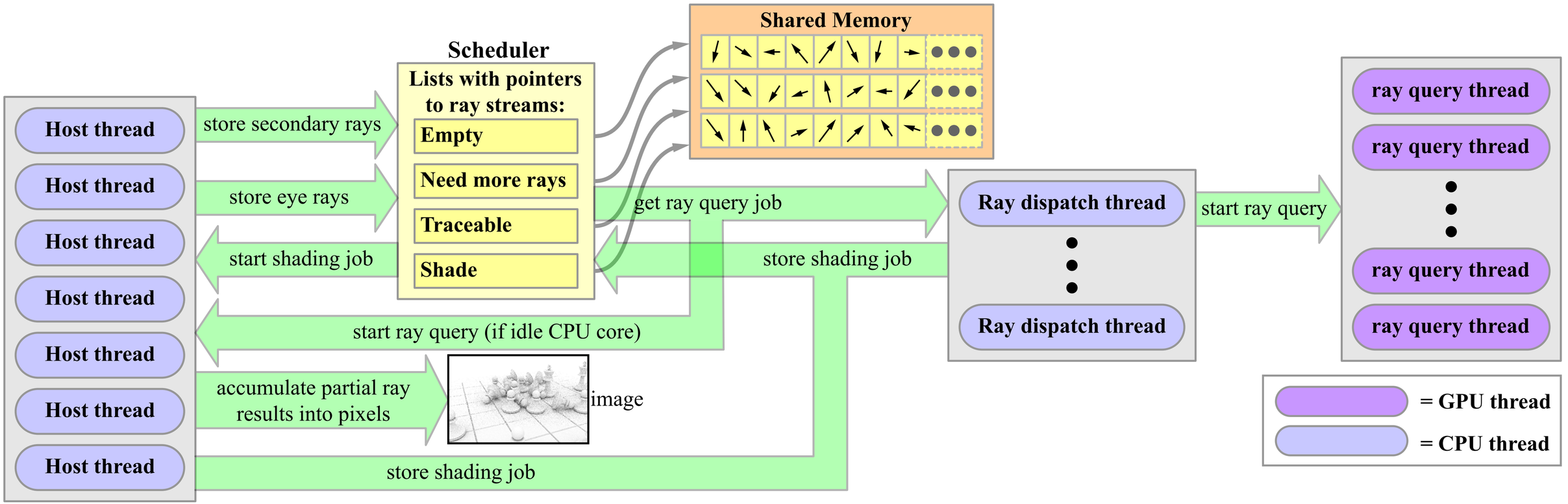
```
float aabbIntersect(float3 min, float3 max, float8 ray,
                    float3 invDir, float3 OoD) {
    float t0 = rayTNear(ray); float t1 = rayTFar(ray);
    float tNear = mad(min.x, invDir.x, OoD.x);
    float tFar  = mad(max.x, invDir.x, OoD.x);
    t0 = fmax( fmin(tNear, tFar), t0);
    t1 = fmin( fmax(tNear, tFar), t1);
    tNear = mad(min.y, invDir.y, OoD.y);
    tFar  = mad(max.y, invDir.y, OoD.y);
    t0 = fmax( fmin(tNear, tFar), t0);
    t1 = fmin( fmax(tNear, tFar), t1);
    tNear = mad(min.z, invDir.z, OoD.z);
    tFar  = mad(max.z, invDir.z, OoD.z);
    t0 = fmax( fmin(tNear, tFar), t0);
    t1 = fmin( fmax(tNear, tFar), t1);
    if (t0 > t1) return rayTFar(ray);
    return t0;
}
```

## ray/triangle intersection

```
inline float triangleIntersect(float4* triangles, int index,
                               float8 ray, float4* hit) {
    float tNear = rayTNear(ray);
    float tMax = rayTFar(ray);
    float3 ro = rayOrigin(ray);
    float3 rd = rayDirection(ray);
    float3 e1 = triangles[index*3+0].xyz;
    float3 e2 = triangles[index*3+1].xyz;
    float3 n = cross(e1, e2);
    float3 v0 = triangles[index*3+2].xyz;
    float3 C = v0 - ro;
    float3 R = cross(rd, C);
    float det = dot(n, rd);
    int sgnDet = as_int(det) & 0x80000000;
    int iU = as_int(dot(R, e2)) ^ sgnDet;
    int iV = as_int(dot(R, e1)) ^ sgnDet;
    if ((iU | iV) < 0) return tMax;
    float U = as_float(iU);
    float V = as_float(iV);
    float absDet = fabs(det);
    float W = absDet-U-V;
    float T = as_float(as_int(dot(n, C)) ^ sgnDet);
    if (W < 0.0f || T <= absDet*tNear || T > absDet*tMax) return tMax;
    float rcpAbsDet = native_recip(absDet);
    float t = T * rcpAbsDet;
    float u = U * rcpAbsDet;
    float v = V * rcpAbsDet;
    *hit = make_float4(as_float(index), t, u, v);
    return t;
}
```

# Our heterogenous ray tracing system:

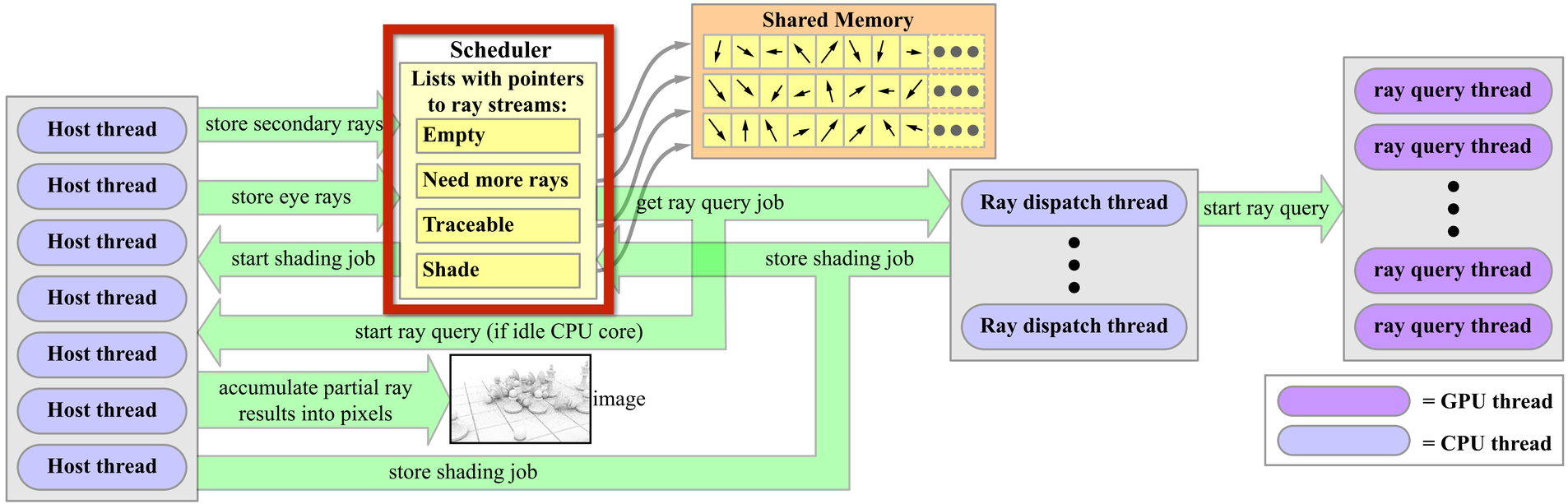
# RayAccelerator




- **CPU cores:** eye rays, scheduling ray queries, evaluating BRDFs, extending light paths
- **GPU cores:** ray query acceleration, i.e., traversal and intersection
- Ray streams: a list of rays, large enough to saturate CPU/GPU resources
- Communication of ray streams happen through shared memory

Developed by Rasmus Barringer & Tomas Akenine-Möller 2014

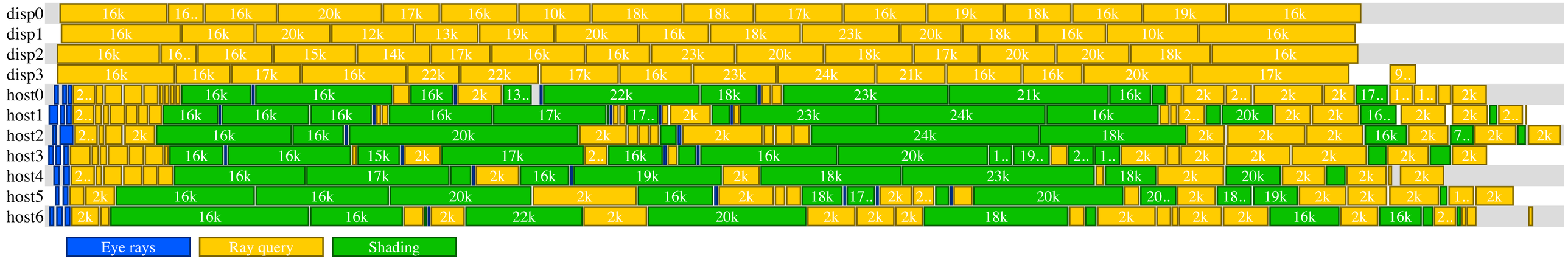
# Central scheduler




- **Host threads:** eye rays, shading, secondary rays, etc.
- **Ray dispatch threads:** feeds the GPU with ray streams to trace
- Host and ray dispatch threads steal jobs (ray streams) from scheduler
- If host threads are starved: perform ray queries on CPU

# RayAccelerator

- Threaded C++ and AVX2
- OpenCL 1.2
- Ray stream: up to 8k rays
- For battlefield, our vanilla ray tracer started at 23 Mrays/s
- Three months of improvements: 50 Mrays/s



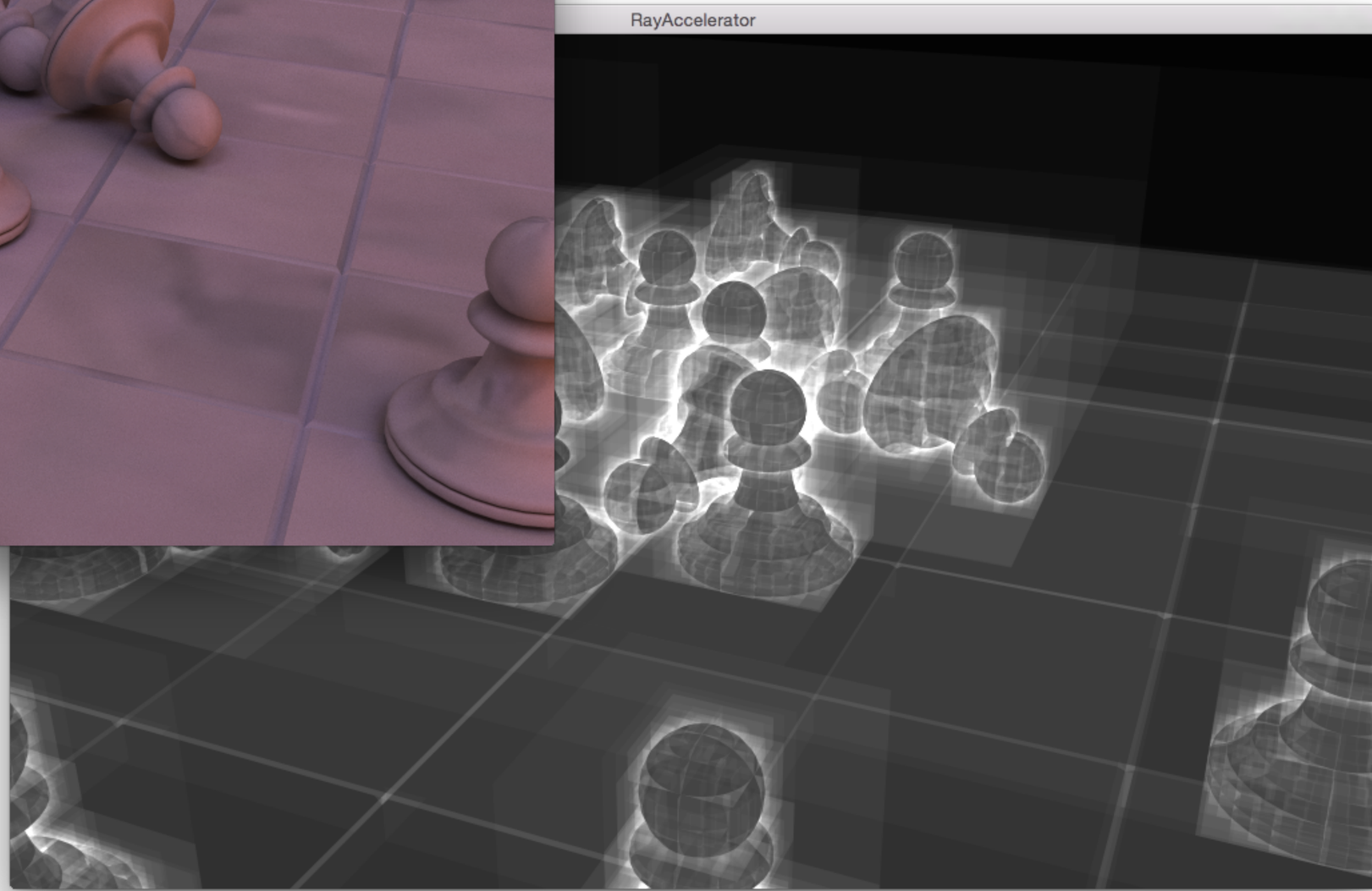

- Improvements possible towards end of frame: could start on next frame

# Some results

- CPU only: 31 Mrays/s
- CPU/GPU: 44 Mrays/s
- CPU/GPU++: 50 Mrays/s
  - ++ means CPU is tracing rays when starved

- Recall that big parts of the memory system is shared (LLC, L4 EDRAM, DRAM), so cannot expect a 3x performance increase
- Also, can in general not “fire up” the entire chip at the time
  - “Dark silicon… ,” Esmaeilzadeh et al., ISCA 2011
- Finally, our* CPU tracer is optimized (years of work)
  - In contrast to many other comparisons between CPU and GPU, where only the GPU implementation is optimized
  - * we use Intel’s Embree traversal/intersection kernels

# Demo

# Current and Future Work

- Wanted to run BVH building on GPU as well
  - Recent result: we can build BVHs on laptop CPU using only 1.6x more time compared to an NVIDIA GTX Titan, which has 10x more FLOPS
  - *BVH building **not** a perfect fit for GPUs* (despite tons of research, lately)
- Improve system using SVM + OpenCL2.0 (nested parallelism)
- Always improve performance
  - Test new tools and discover new ways to get better performance
- Propose new HW features?
  - Instructions for EUs, for example.
- Scaling **up** and **down** is always a challenge
- Dark silicon…

# The End

- **Questions?**
  - Just know that **“I do not know”** can mean either:
    - “I do not know,” or
    - “I know, but cannot tell you.” ;-)

## Sources and Acknowledgements:


- Thanks to the Advanced Rendering Technology team
  - For feedback: Jim Nilsson, Robert Toth, Magnus Andersson, Jon Hasselgren, Jacob Munkberg
- Thanks to Murali Sundaresan at Intel’s GPGPU group + Julia Federova
- Thanks to Rasmus Barringer for collaboration on RayAccelerator

- Sources:

  - “The Compute Architecure of Intel Processor Graphics Gen8,” Stephen Junkins, Intel Developer Forum (IDF), 2014
  - “Optimizing Your Applications Using OpenCL on Intel Iris Graphics,” Jayant Rao and Raun Krisch, Intel Developer Forum (IDF), 2013
  - “The Compute Architecture of Intel® Processor Graphics Gen8,” v1.0, 2014.
  - “OpenCL 2.0 Shared Virtual Memory Overview,” September 2014.
    - https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview
  - “The OpenCL Specification 2.0,” Aaftab Munshi (ed.), rev. 22, 2014-03-18
  - “Real-Time Rendering,” Akenine-Möller et al., 3rd ed., AK Peters Ltd, 2008. http://www.realtimerendering.com
  - “Embree: a kernel framework for efficient CPU ray tracing,” SIGGRAPH 2014, by Wald et al.