

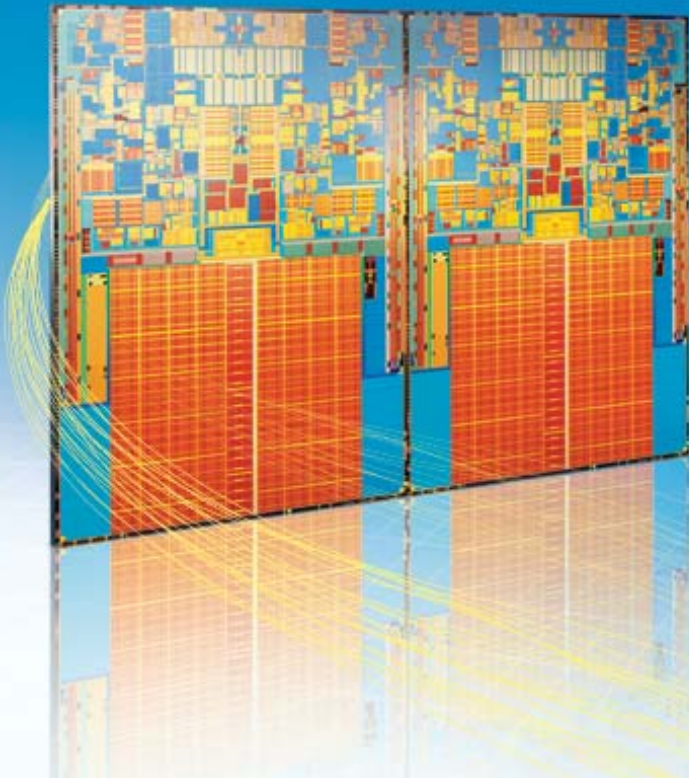


# OpenCL\*, Heterogeneous Computing, and the CPU

Dr. Tim Mattson

Visual Applications Research lab

Intel Corporation



\*3<sup>rd</sup> party names are the property of their owners

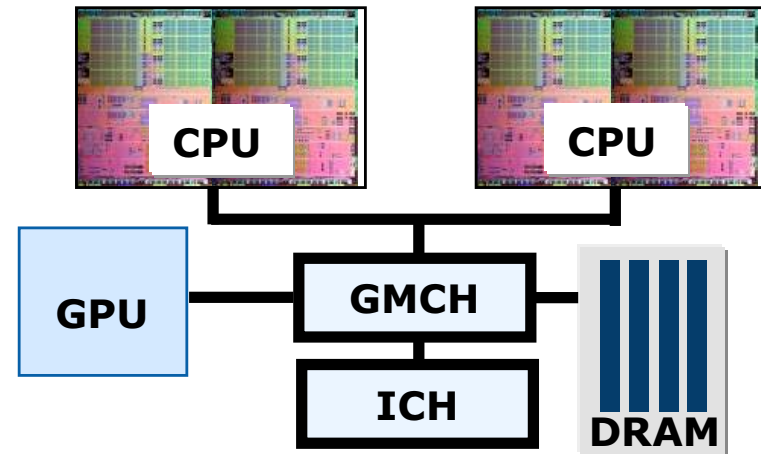
© Intel Corporation, 2009

# Agenda

- **Heterogeneous computing in OpenCL**
- Data parallelism in OpenCL
- Task parallelism in OpenCL
- Future direction for parallel computing

# Heterogeneous computing

- A modern platform has:
  - Multi-core CPU(s)
  - A GPU
  - DSP processors
  - ... other?



- The goal should NOT be to “off-load” the CPU. We need to make the best use of all the available resources from within a single program:
  - One program that runs well (i.e. reasonably close to “hand-tuned” performance) on a heterogeneous mixture of processors.

# OpenCL: it's not just a GPGPU Language

- OpenCL defines a platform API to coordinate heterogeneous parallel computations
  - Literature rich with parallel coordination languages/API
  - OpenCL unique in its ability to coordinate CPUs, GPUs, etc
- Key coordination concepts
  - Each device has its own asynchronous workqueue
  - Synchronize between OCL computations w/event handles from different (or same) devices
  - Enables algorithms and systems that use all available computational resources
  - Enqueue “native functions” for integration with C/C++ code

# Agenda

- Heterogeneous computing in OpenCL
- **Data parallelism in OpenCL**
- Task parallelism in OpenCL
- Future direction for parallel computing

# OpenCL's Two Styles of Data-Parallelism

- Explicit SIMD data parallelism:
  - The kernel defines one stream of instructions
  - Parallelism from using wide vector types
  - Size vector types to match native HW width
  - Combine with task parallelism to exploit multiple cores
- Implicit SIMD data parallelism (i.e. shader-style):
  - Write the kernel as a “scalar program”
  - Use vector data types sized naturally to the algorithm
  - Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware.

**Both approaches are viable CPU options**

# Data-Parallelism: options on IA processors

- Explicit SIMD data parallelism
  - Programmer chooses vector data type (width)
  - Compiler hints using attributes
    - `vec_type_hint(typen)`
- Implicit SIMD Data parallel
  - Map onto CPUs, GPUs, Larrabee, ...
    - SSE/AVX/LRBni: 4/8/16 workitems in parallel
- Hybrid use of the two methods
  - AVX: can run two 4-wide workitems in parallel
  - LRBni: can run four 4-wide workitems in parallel

# Explicit SIMD data parallelism

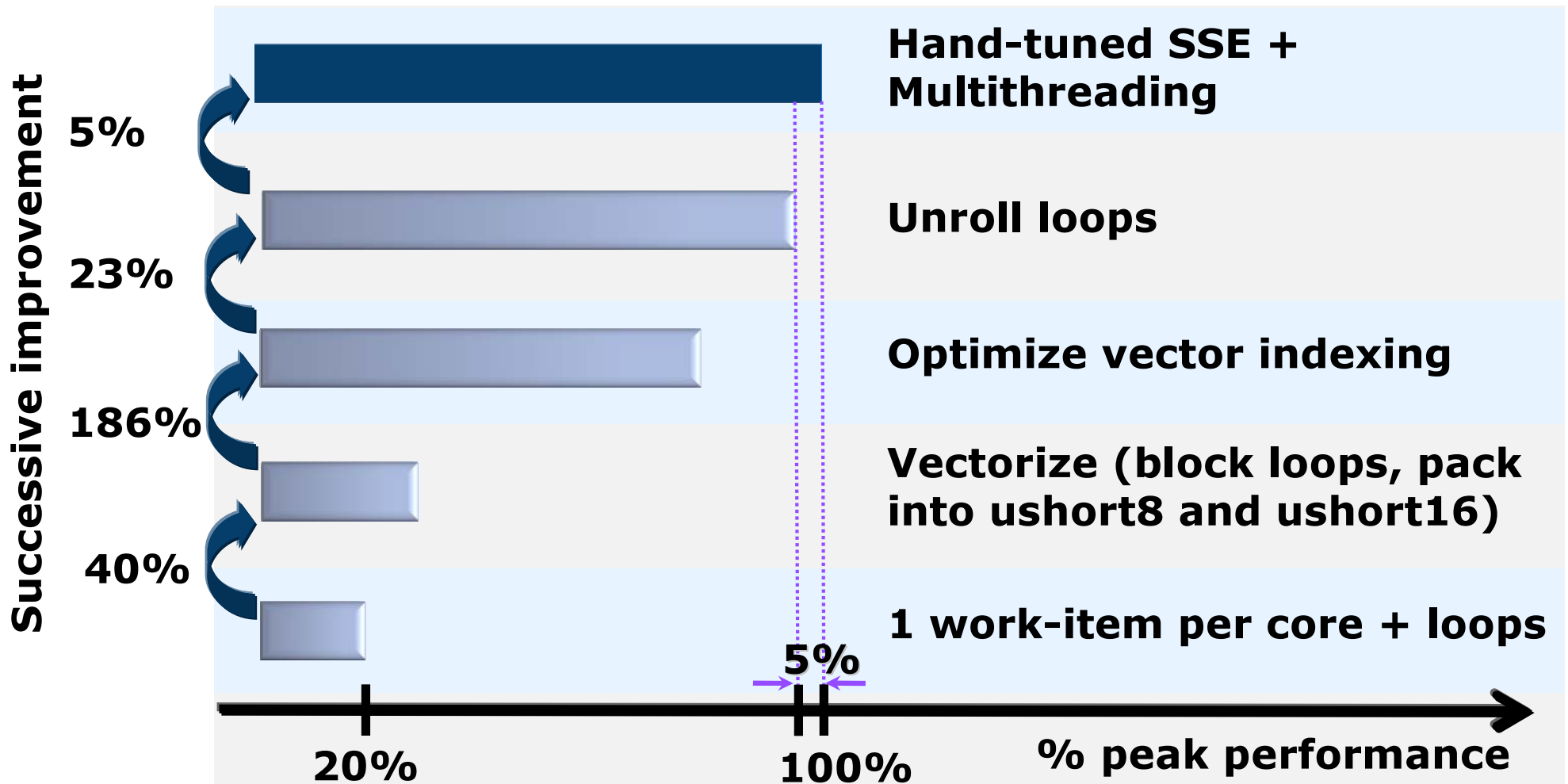
- OpenCL as a portable interface to vector instruction sets
  - Block loops and pack data into vector types (float4, ushort16, etc).
  - Replace scalar ops in loops with blocked loops and vector ops.
  - Unroll loops, optimize indexing to match machine vector width

```
float a[N], b[N], c[N];  
for (i=0; i<N; i++)  
    c[i] = a[i]*b[i];  
  
<<< the above becomes >>>>  
  
float4 a[N/4], b[N/4], c[N/4];  
for (i=0; i<N/4; i++)  
    c[i] = a[i]*b[i];
```

**Explicit SIMD data parallelism means you tune your code to the vector width and other properties of the compute device**

# Explicit SIMD data parallelism: Case Study

- Video contrast/color optimization kernel on a dual core CPU

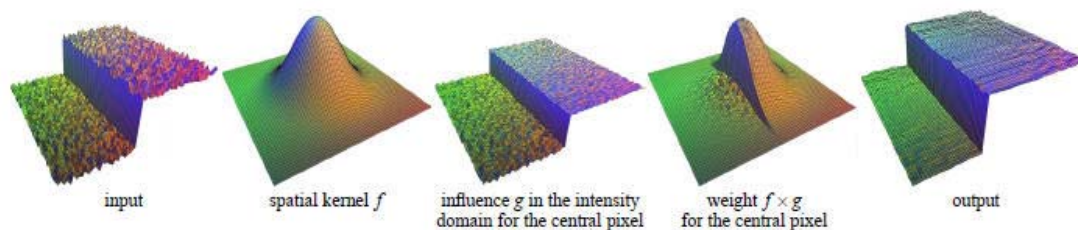


**Good news: OpenCL code 95% of hand-tuned SSE/MT perf.**

**Bad news: New platform, redo all those optimizations.**

# Towards “Portable” Performance

- The following C code is an example of a Bilateral 1D filter:
- Reminder: Bilateral filter is an edge preserving image processing algorithm.
- See more information here:  
<http://scien.stanford.edu/class/psych221/projects/06/imagescaling/bilati.html>



```
void P4_Bilateral9 (int start, int end, float v)
{
    int i, j, k;
    float w[4], a[4], p[4];
    float inv_of_2v = -0.5 / v;
    for (i = start; i < end; i++) {
        float wt[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
        for (k = 0; k < 4; k++)
            a[k] = image[i][k];
        for (j = 1; j <= 4; j++) {
            for (k = 0; k < 4; k++)
                p[k] = image[i - j*SIZE][k] - image[i][k];
            for (k = 0; k < 4; k++)
                w[k] = exp (p[k] * p[k] * inv_of_2v);
            for (k = 0; k < 4; k++) {
                wt[k] += w[k];
                a[k] += w[k] * image[i - j*SIZE][k];
            }
        }
        for (j = 1; j <= 4; j++) {
            for (k = 0; k < 4; k++)
                p[k] = image[i + j*SIZE][k] - image[i][k];
            for (k = 0; k < 4; k++)
                w[k] = exp (p[k] * p[k] * inv_of_2v);
            for (k = 0; k < 4; k++) {
                wt[k] += w[k];
                a[k] += w[k] * image[i + j*SIZE][k];
            }
        }
        for (k = 0; k < 4; k++) {
            image2[i][k] = a[k] / wt[k];
        }
    }
}
```

# Towards “Portable” Performance

```
void P4_Bilateral9 (int start, int end, float v)
```

```
{
```

- The following example

```
void P4_Bilateral9 (int start, int end, float v)  
{
```

```
    <<< Declarations >>>
```

- Reminder edge processing

```
    for (i = start; i < end; i++) {
```

```
        for (j = 1; j <= 4; j++) {
```

- See more <http://scien.projects/06/>

```
            <<< a series of short loops >>>>
```

```
        }
```

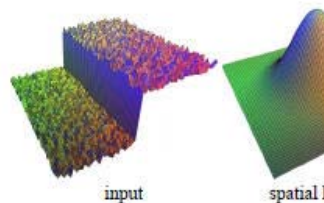
```
    for (j = 1; j <= 4; j++) {
```

```
        <<< a 2nd series of short loops >>>
```

```
    }
```

```
}
```

```
}
```



# “Implicit SIMD” data parallel code

- “outer” loop replaced by work-items running over an NDRange index set
- NDRange 4\*image size ... since each workitem does a color for each pixel
- Leave it to the compiler to map work-items onto lanes of the vector units ...

```
__kernel void P4_Bilateral9 (__global float* inImage, __global float* outImage, float v)
{
    const size_t myID    = get_global_id(0);
    const float inv_of_2v = -0.5f / v;
    const size_t myRow    = myID / IMAGE_WIDTH;
        size_t maxDistance = min(DISTANCE, myRow);
        maxDistance = min(maxDistance, IMAGE_HEIGHT - myRow);
    float currentPixel, neighborPixel, newPixel;
    float diff;
    float accumulatedWeights, currentWeights;
    newPixel = currentPixel = inImage[myID];
    accumulatedWeights = 1.0f;
    for (size_t dist = 1; dist <= maxDistance; ++dist)
    {
        neighborPixel    = inImage[myID + dist*IMAGE_WIDTH];
        diff              = neighborPixel - currentPixel;
        currentWeights    = exp(diff * diff * inv_of_2v);
        accumulatedWeights += currentWeights;
        newPixel          += neighborPixel * currentWeights;
        neighborPixel    = inImage[myID - dist*IMAGE_WIDTH];
        diff              = neighborPixel - currentPixel;
        currentWeights    = exp(diff * diff * inv_of_2v);
        accumulatedWeights += currentWeights;
        newPixel          += neighborPixel * currentWeights;
    }
    outImage[myID] = newPixel / accumulatedWeights;
}
```

# “Implicit SIMD” data parallel code

```
kernel void P4_Bilateral9 ( __global float* inImage, __global float* outImage, float v)
```

- “O by run ND
- ND siz WC CO
- Le co WC lar un

```
__kernel void p4_bilateral9(__global float* inImage,  
                            __global float* outImage, float v)  
{  
    const size_t myID    = get_global_id(0);  
    <<< declarations >>>  
    for (size_t dist = 1; dist <= maxDistance; ++dist){  
        neighborPixel    = inImage[myID +  
                               dist*IMAGE_WIDTH];  
        diff              = neighborPixel - currentPixel;  
        currentWeights    = exp(diff * diff * inv_of_2v);  
        << plus others to compute pixels, weights, etc >>  
        accumulatedWeights += currentWeights;  
    }  
    outImage[myID] = newPixel / accumulatedWeights;  
}
```

```
}
```

# Portable Performance in OpenCL

- Implicit SIMD code ... where the framework maps work-items onto the “lanes of the vector unit” ... creates the opportunity for portable code that performs well on full range of OpenCL compute devices
- Requires mature OpenCL technology that “knows” how to do this:
  - ... But it is important to note .... we know this approach works since its based on the way shader compilers work today

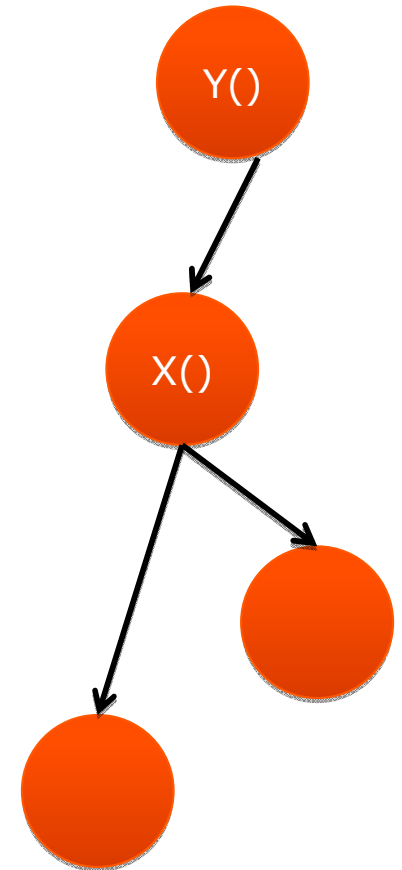
# Agenda

- Heterogeneous computing in OpenCL
- Data parallelism in OpenCL
- **Task parallelism in OpenCL**
- Future direction for parallel computing

# Task Parallelism Overview

- Think of a task as an asynchronous function call
  - “Do X at some point in the future”
  - Optionally “... after Y is done”
  - Light weight, often in user space
- Strengths
  - Copes well with heterogeneous workloads
  - Doesn't require 1000's of strands
  - Scales well with core count
- Limitations
  - No automatic support for latency hiding
  - Must explicitly write SIMD code

**A natural fit to multi-core CPUs**



# Task Parallelism in OpenCL

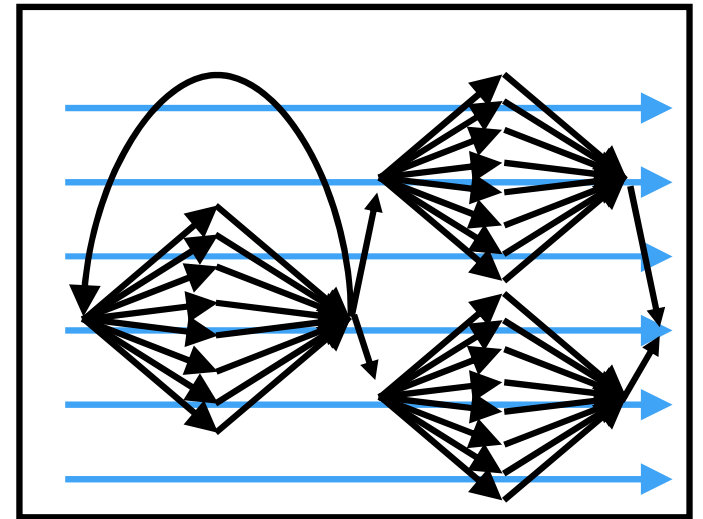
- `clEnqueueTask`
  - Imagine “sea of different tasks” executing concurrently
  - A task “owns the core” (i.e., a workgroup size of 1)
- Use tasks when algorithm...
  - Benefits from large amount of local/private memory
  - Has predictable global memory accesses
  - Can be programmed using explicit vector style
  - “Just doesn’t have 1000’s of identical things to do”
- Use data-parallel kernels when algorithm...
  - Does not benefit from large amounts of local/private memory
  - Has unpredictable global memory accesses
  - Needs to apply same operation across large number of data elements

# Agenda

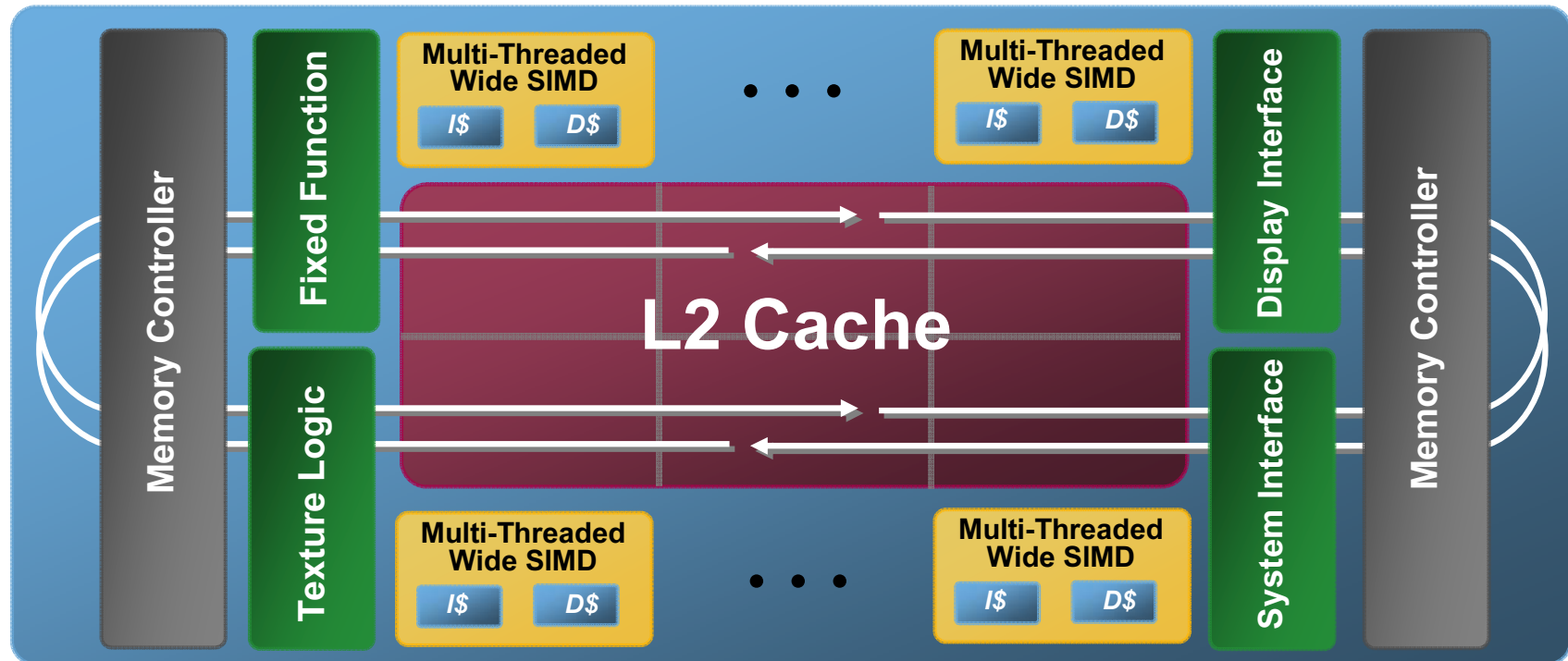
- Heterogeneous computing in OpenCL
- Data parallelism in OpenCL
- Task parallelism in OpenCL
- **Future direction for parallel computing**

# Future Parallel Programming

- Real world applications contain data parallel parts as well as serial/sequential parts
- OpenCL addresses these Apps need by supporting Data Parallel & Task Parallel
- Braided Parallelism – composing Data Parallel & Task Parallel constructs in a single algorithm
- CPUs are ideal for Braided Parallelism

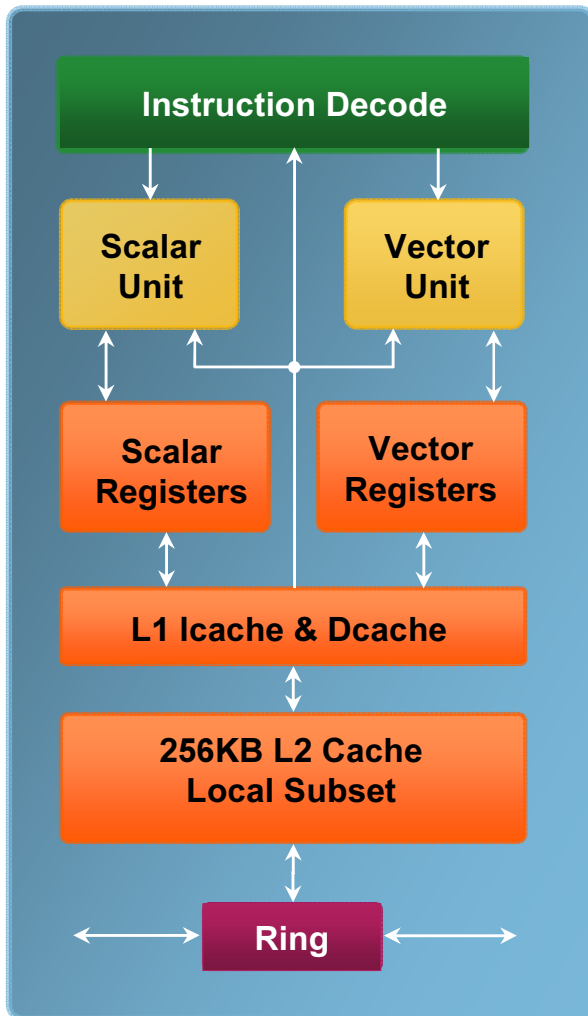


# Future parallel programming: Larrabee



- Cores communicate on a wide ring bus
  - Fast access to memory and fixed function blocks
  - Fast access for cache coherency
- L2 cache is partitioned among the cores
  - Provides high aggregate bandwidth
  - Allows data replication & sharing

# Processor Core Block Diagram



- Separate scalar and vector units with separate registers
- Vector unit: 16 32-bit ops/clock
- In-order instruction execution
- Short execution pipelines
- Fast access from L1 cache
- Direct connection to each core's subset of the L2 cache
- Prefetch instructions load L1 and L2 caches

# Key Differences from Typical GPUs

- Each Larrabee core is a complete Intel processor
  - Context switching & pre-emptive multi-tasking
  - Virtual memory and page swapping, even in texture logic
  - Fully coherent caches at all levels of the hierarchy
- Efficient inter-block communication
  - Ring bus for full inter-processor communication
  - Low latency high bandwidth L1 and L2 caches
  - Fast synchronization between cores and caches

***Larrabee is perfect for the braided parallelism  
in future applications***

# Conclusion

- OpenCL defines a platform-API/framework for heterogeneous computing ... not just GPGPU or CPU-offload programming
- OpenCL has the potential to deliver portably performant code; but only if its used correctly:
  - Implicit SIMD data parallel code has the best chance of mapping onto a diverse range of hardware ... once OpenCL implementation quality catches up with mature shader languages
- The future is clear:
  - Braided parallelism mixing task parallel and data parallel code in a single program ... balancing the load among ALL OF the platform's resources
  - OpenCL can handle this ... and emerging platforms such as Larrabee are well suited to support this model

# References

- **s09.idav.ucdavis.edu** for slides from a Siggraph2009 course titled “Beyond Programmable Shading”
- Seiler, L., Carmean, D., et al. 2008. *Larrabee: A many-core x86 architecture for visual computing*. SIGGRAPH '08: ACM SIGGRAPH 2008 Papers, ACM Press, New York, NY
- Fatahalian, K., Houston, M., “GPUs: a closer look”, Communications of the ACM October 2008, vol 51 #10.  
[graphics.stanford.edu/~kayvonf/papers/fatahalianCACM.pdf](http://graphics.stanford.edu/~kayvonf/papers/fatahalianCACM.pdf)

# Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.
- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Larrabee and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user
- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
- Intel, Intel Inside and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- \*Other names and brands may be claimed as the property of others.
- Copyright © 2009 Intel Corporation.

# Risk Factors

This presentation contains forward-looking statements that involve a number of risks and uncertainties. These statements do not reflect the potential impact of any mergers, acquisitions, divestitures, investments or other similar transactions that may be completed in the future. The information presented is accurate only as of today's date and will not be updated. In addition to any factors discussed in the presentation, the important factors that could cause actual results to differ materially include the following: Demand could be different from Intel's expectations due to factors including changes in business and economic conditions, including conditions in the credit market that could affect consumer confidence; customer acceptance of Intel's and competitors' products; changes in customer order patterns, including order cancellations; and changes in the level of inventory at customers. Intel's results could be affected by the timing of closing of acquisitions and divestitures. Intel operates in intensely competitive industries that are characterized by a high percentage of costs that are fixed or difficult to reduce in the short term and product demand that is highly variable and difficult to forecast. Revenue and the gross margin percentage are affected by the timing of new Intel product introductions and the demand for and market acceptance of Intel's products; actions taken by Intel's competitors, including product offerings and introductions, marketing programs and pricing pressures and Intel's response to such actions; Intel's ability to respond quickly to technological developments and to incorporate new features into its products; and the availability of sufficient supply of components from suppliers to meet demand. The gross margin percentage could vary significantly from expectations based on changes in revenue levels; product mix and pricing; capacity utilization; variations in inventory valuation, including variations related to the timing of qualifying products for sale; excess or obsolete inventory; manufacturing yields; changes in unit costs; impairments of long-lived assets, including manufacturing, assembly/test and intangible assets; and the timing and execution of the manufacturing ramp and associated costs, including start-up costs. Expenses, particularly certain marketing and compensation expenses, vary depending on the level of demand for Intel's products, the level of revenue and profits, and impairments of long-lived assets. Intel is in the midst of a structure and efficiency program that is resulting in several actions that could have an impact on expected expense levels and gross margin. Intel's results could be impacted by adverse economic, social, political and physical/infrastructure conditions in the countries in which Intel, its customers or its suppliers operate, including military conflict and other security risks, natural disasters, infrastructure disruptions, health concerns and fluctuations in currency exchange rates. Intel's results could be affected by adverse effects associated with product defects and errata (deviations from published specifications), and by litigation or regulatory matters involving intellectual property, stockholder, consumer, antitrust and other issues, such as the litigation and regulatory matters described in Intel's SEC reports. A detailed discussion of these and other factors that could affect Intel's results is included in Intel's SEC filings, including the report on Form 10-Q for the quarter ended June 28, 2008.