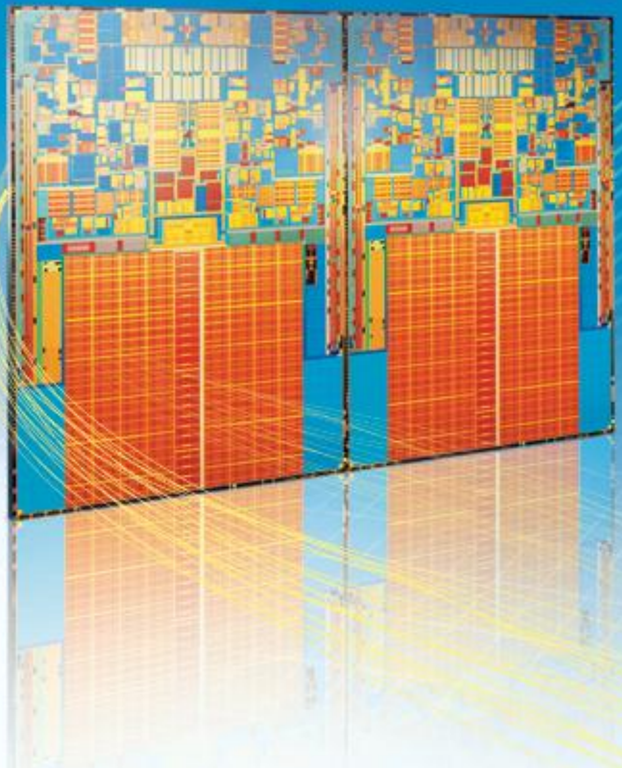
On the value of vector level parallelism



Robert Geva

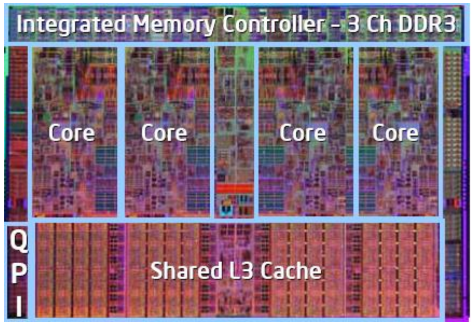
Disclaimer

- Performance data shown here is preliminary
- It is work in progress
- Peer review still ongoing, conclusions may change

HW Resources for Parallel Execution

Multiple cores

Hardware threads



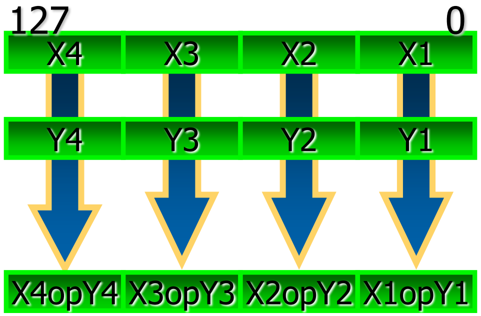
Tasks

Cilk, TBB, PPL

OpenMP

Auto Par??

SIMD instructions



Vectors

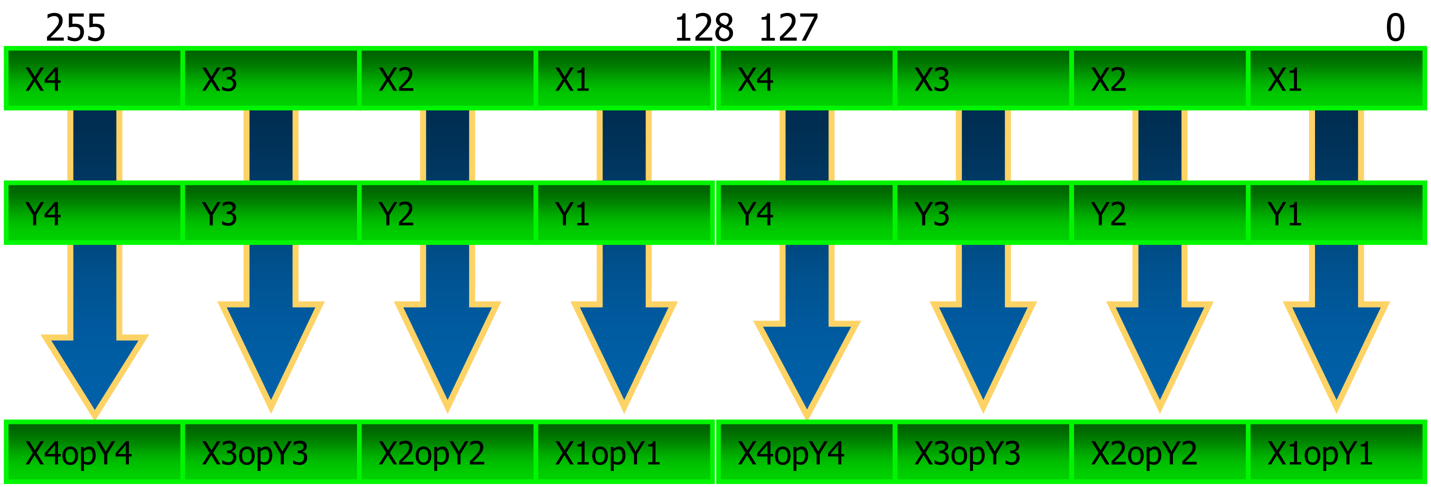
Array Notations

SIMD loops

Auto Vec??

Parallel tasks with SIMD kernels

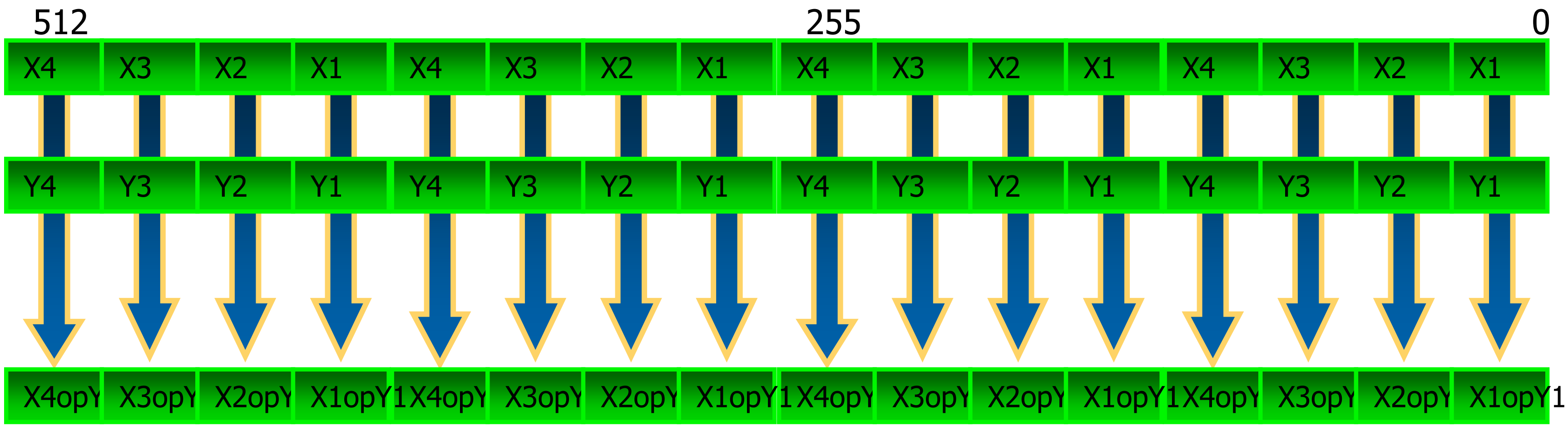
SIMD Instructions Compute Multiple Operations per Instruction



256b Intel® Advanced Vector Extensions (Intel® AVX)

Intel® Next Generation microarchitecture codename Sandy Bridge
256-bit Multiply + 256-bit ADD + 256-bit Load per clock…
Double your FLOPs with great energy-efficiency

SIMD Instructions Compute Multiple Operations per Instruction

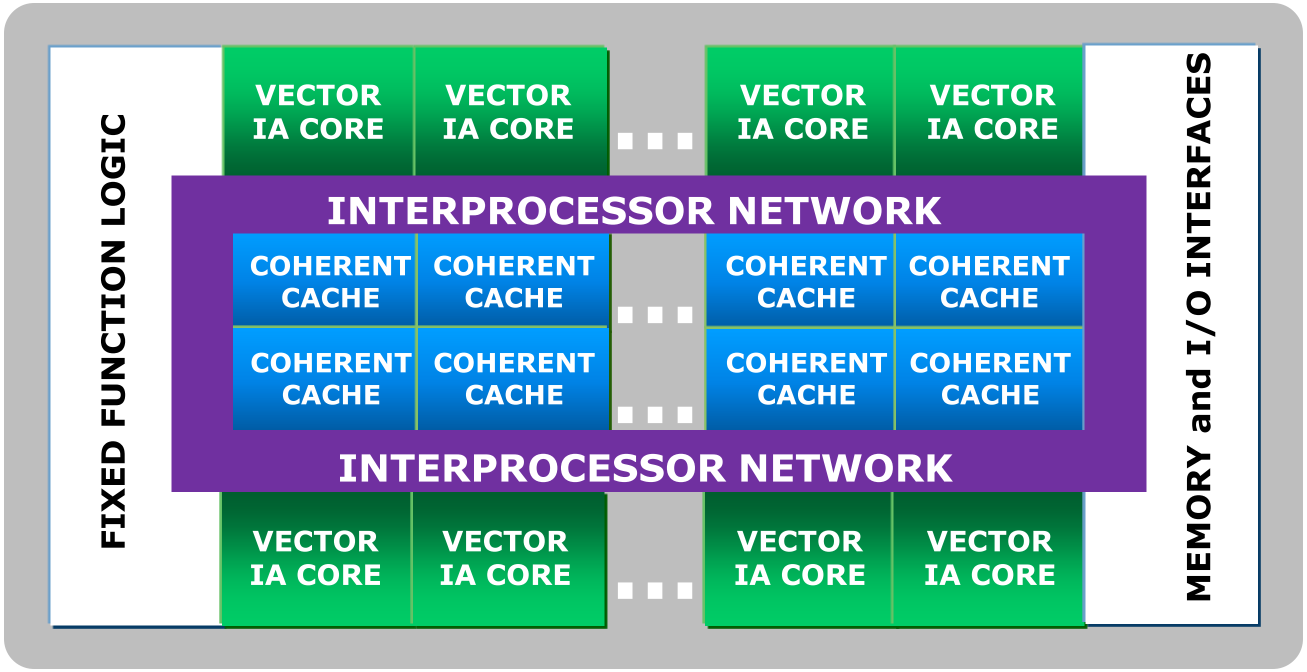


Intel® Many Integrated Core Architecture

Wide SIMD to support data parallel programming

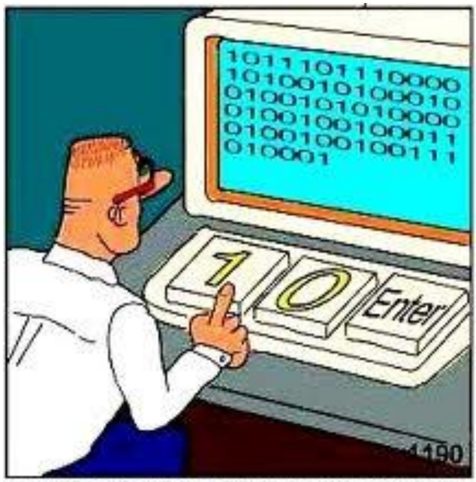
Intel® Many Integrated Core Architecture

An Intel Co-Processor Architecture



Many Core and many, many core threads, wider vectors
Standard Intel® Architecture programming and memory model

Programmer Personalities



REAL Programmers code in BINARY.

Different programmers want different levels of control over how their program executes

Programmer perspective vs. HW perspective

- Ability to program tasks vs. vectors explicitly
  - Requirement to be able to program the vectors with a single thread semantic guarantee
  - Tasking is expressed at an outer level
  - Have been burned by over subscription, don't rely on composability guarantees
- Ability to express intent for parallel execution and let the compiler map to HW resources
  - Parallel loops, utilize all HW resources
  - Elemental functions, or SPMD execution model
  - Implementation of these constructs with only cores is non competitive
    - Lower performance than other languages, e.g. OpenCL kernels

Data parallelism uses both cores and vectors

Therefore needs to compose with tasking

Currently Available

- Auto vectorizers
- Intrinsics
- Fortran
- OpenMP – coming soon
- OpenCL

Components of Intel® Cilk™ Plus

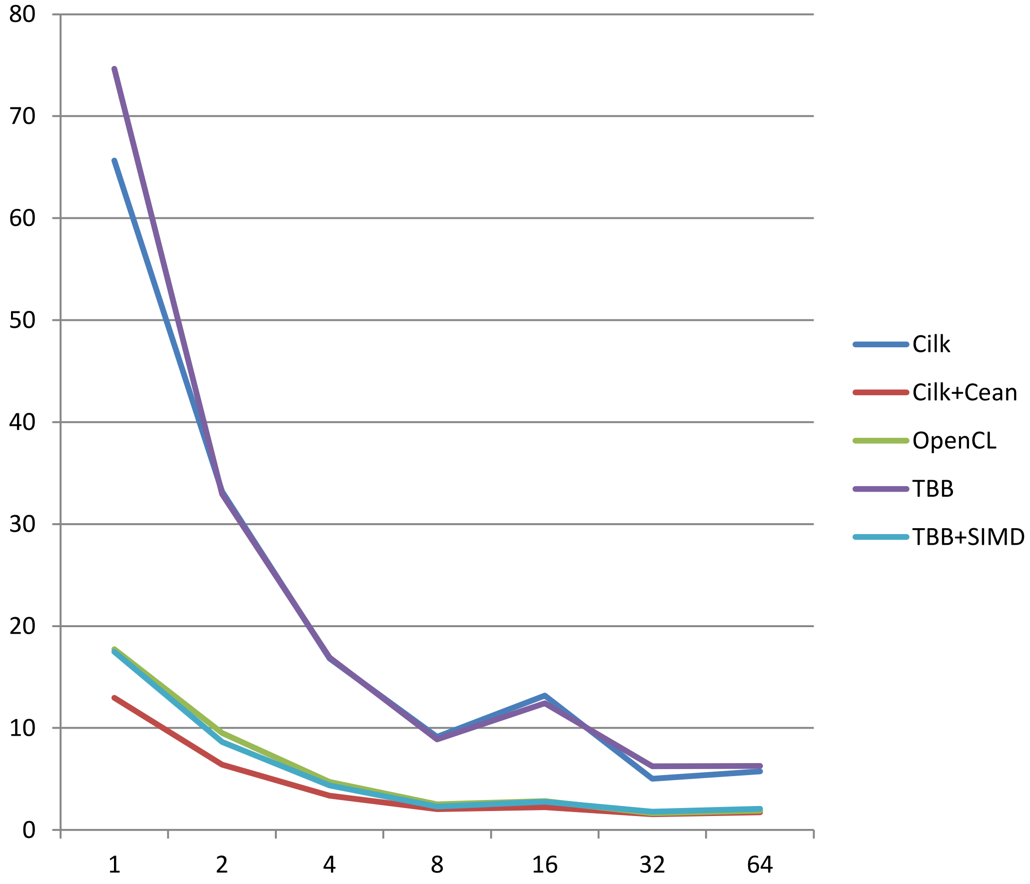
3 keywords for tasking	•Easy to learn, use and maintain, no programming overhead •Execution of parallel code is equivalent to execution on a single thread •Very low run time overhead
Hyper Objects	•Provide local views of global data to allow reduction operation w/o data races •No use of locks •Can be used independent of program control flow
Array notations	•Mathematical operations on arrays w/o constrained serial ordering •Implementation utilizes the vector ISA
Elemental Functions	•Write standard C/C++ scalar •Compiler generates a version to operate on a short vector of arguments •The implementation can also spawn instances onto multiple cores
Pragma SIMD	•Write standard C/C++/FTN loops •Guaranteed vector implementation by the compiler

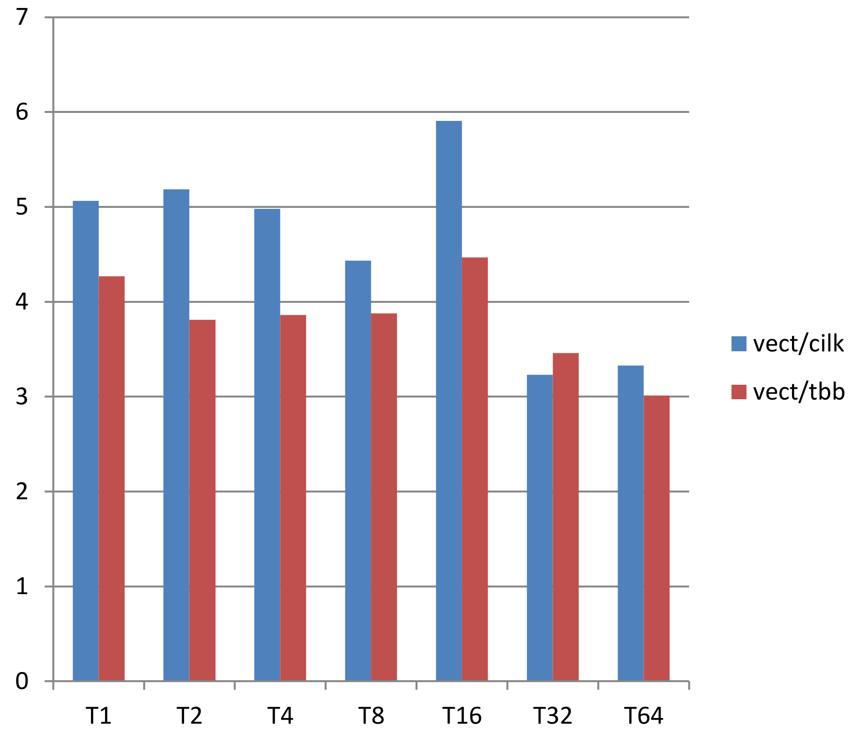
Significance of vectorization - RTM stencil

	1	2	4	8	16	32	64
Cilk	65.64	33.18	16.83	9.13	13.17	5.04	5.76
Cilk+vec	12.96	6.4	3.38	2.06	2.23	1.56	1.73
OpenCL	17.72	9.5	4.73	2.51	2.84	1.65	1.89
TBB	74.66	32.93	16.91	8.88	12.42	6.26	6.29
TBB+vec	17.49	8.64	4.38	2.29	2.78	1.81	2.09

- In both Cilk+vec and TBB+vec, significant speed up over tasking alone, at all thread counts
- Without vectorizaiton, OpenCL (SPMD model) wins over C++

And now with pictures





Significance of vectorization - AObench

Nthreads	Cilk	Cilk + simd	improvement	TBB	TBB + simd	improvement
1	18.93	13.9	0.734284	16.95	13.81	0.814749
2	9.07	6.83	0.753032	8.49	6.84	0.805654
4	4.66	3.37	0.723176	4.34	3.41	0.785714
8	2.12	1.81	0.853774	2.14	1.91	0.892523
16	1.71	1.35	0.789474	1.63	1.44	0.883436
32	0.81	0.7	0.864198	0.83	0.72	0.86747
64	0.6	0.52	0.866667	0.62	0.53	0.854839

Vector level parallelism provides significant improvement over thread level parallelism

Significance of vectorization – Binomial Lattice

nthreads	cilk	cilk + cean	improvement
1	18.39	17.62	0.95812942
2	9.45	9.06	0.95873016
4	4.84	4.64	0.95867769
8	2.57	2.45	0.95330739
16	2.81	2.17	0.77224199
32	1.15	1.02	0.88695652
64	0.98	0.76	0.7755102

Vector level parallelism provides significant improvement over thread level parallelism

Significance of vectorization – Track Fitting

nthreads	cilk	cilk_simd	opencl	tbb	tbb_simd
1	47.27	24.94	16.96	43.04	22.43
2	24.02	12.79	8.74	20.9	11.49
4	12.38	6.63	4.8	10.7	5.77
8	6.85	3.47	2.85	5.45	2.94
16	6.17	3.21	2.61	5.2	2.71
32	2.48	1.41	1.66	2.02	1.16
64	2.08	1.19	1.56	1.55	0.93

Vector level parallelism provides significant improvement over thread level parallelism

Array notations Example: Dot product

Serial version

```
float dot_product(unsigned int sz, float A[sz], float B[sz])
{
    int i;
    float dp=0.0f;
    for (i=0; i<size; i++) {
         dp += A[i] * B[i];
    }
    return dp;
}
```

Array Notation version

```
float dot_product(float A[], float B[])
{
    return __sec_reduce_add(A[:] * B[:]);
}
```

Elemental functions example: Black Scholes

```
__declspec(vector)
double option_price_call_black_scholes(double S, double K,double r,double sigma,  double time)
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
```

```
cilk_for (int i=0; i < NUM_OPTIONS; i++) {
        call_serial[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
```

Optimal utilization of cores and vectors

Invoking Elemental Functions

Constrcut	Example	Semantics

Standard for loop	`for (j = 0; j < N; j++) {`
	`    a[j] = my_ef(b[j]);`
	`}`
		Single thread,
		auto vectorization

#pragma simd	`#pragma simd`
	`for (j = 0; j < N; j++) {`
	`    a[j] = my_ef(b[j]);`
	`}`
		Single thread,
		Guaranteed to use the vector version

cilk for loop	`cilk_for (j = 0; j < N; j++) {`
	`    a[j] = my_ef(b[j]);`
	`}`
		Both vectorization and concurrent execution

Array notation	`a[:] = my_ef(b[:]);`	Vectorization. Concurrency allowed by not yet implemented

The execution of the elemental functions is serial with respect to the code that follows the invocation.

SIMD loop example: Mandelbrot

```
// vectorizable outer loop
#pragma simd
for (i=0; i<n; i++) {
    complex<float> c = a[i];
    complex<float> z = c;
    int j = 0;
    while ((j < 255)
          && (abs(z)< limit)) {
          z = z*z + c;
          j++;
    };
    color[i] = j;
}
```

- This program results in good utilization of vector level parallelism and provides measureable speedups.
- Arguably out of reach of auto vectorizers
- Outlining the loop body can be written as an elemental function. However, in line code is normally more efficient.

A simd loop

- Loops count number of elements that are inside/outside Mandelbrot set

```
 for (int32_t y = 0; y < ImageHeight; ++y) {
#pragma simd reduction(+:num_out) reduction(+:num_in)
       for(int32_t x = 0; x < ImageWidth; ++x) {
           if (count[y][x] < max_iter) {
               num_out += 1;
           }
           else {
               num_in += 1;
           }
       }
    }
```

Capabilities

- Uniform values – same across all vector lanes
- Linearly increasing values, inductive
- Reductions

- x += something → an induction, a reduction, other?

Sponsors of Tomorrow.™

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.
- Nehalem, Westmere, Sandy Bridge 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, Core, Itanium 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 © 2010 Intel Corporation.

Optimization Notice

Intel® Composer XE 2011 includes compiler options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel® Composer XE 2011 are reserved for Intel microprocessors. For a detailed description of these compiler options, including the instruction sets they implicate, please refer to "Intel® Composer XE 2011 Documentation > Intel® C++ Compiler 12.0 User and Reference Guides > Compiler Options." Many library routines that are part of Intel® Composer XE 2011 are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® Composer XE 2011 offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® Composer XE 2011, with respect to Intel's compilers and associated libraries as a whole, Intel® Composer XE 2011 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

Intel recommends that you evaluate other compilers to determine which best meet your requirements.

A Linear Argument

- An argument whose value increments linearly across the projection
- When used as an index in loads and stores, linear makes the difference between ld/st and gather / scatter

```
__declspec (vector (linear(i:1)))
  void add_vec (int i)
  {
              a[i] = b[i] + c[i];
  }

----------------------------------------
for (int i = 0; i < N; ++i) {
    add_vec(i);
}
```

A Linear Argument

```
push      edi
movd      eax, xmm0
pshuflw   xmm1, xmm0, 238
punpckhqdq xmm0, xmm0
movd      edx, xmm1
pshuflw   xmm1, xmm0, 238
movd      xmm3, DWORD PTR [_b+eax*4]
movd      ecx, xmm0
movd      edi, xmm1
movd      xmm2, DWORD PTR [_b+edx*4]
punpcklqdq xmm3, xmm2
movd      xmm2, DWORD PTR [_b+ecx*4]
movd      xmm0, DWORD PTR [_b+edi*4]
punpcklqdq xmm2, xmm0
shufps    xmm3, xmm2, 136
movd      xmm1, DWORD PTR [_c+eax*4]
movd      xmm2, DWORD PTR [_c+edx*4]
punpcklqdq xmm1, xmm2
movd      xmm2, DWORD PTR [_c+ecx*4]
movd      xmm0, DWORD PTR [_c+edi*4]
punpcklqdq xmm2, xmm0
shufps    xmm1, xmm2, 136
paddd     xmm3, xmm1
pshuflw   xmm1, xmm3, 238
movd      DWORD PTR [_a+eax*4], xmm3
punpckhqdq xmm3, xmm3
movd      DWORD PTR [_a+edx*4], xmm1
movd      DWORD PTR [_a+ecx*4], xmm3
pshuflw   xmm3, xmm3, 238
movd      DWORD PTR [_a+edi*4], xmm3
pop       edi
ret
```

```
.B5.1:                          ; Preds .B5.0
        movdqu    xmm1, XMMWORD PTR [_b+eax*4]
        movdqu    xmm0, XMMWORD PTR [_c+eax*4]
        paddd     xmm1, xmm0
        movdqu    XMMWORD PTR [_a+eax*4], xmm1
        ret
```

A Scalar Argument

- An argument whose value is the same for all lanes
- When used as a indexbase in loads and stores, scalar makes the difference between ld/st and gather / scatter

```
__declspec (vector (linear(i:1)),scalar(a,b,c))

  void add_vec (float *a, float *b, float *c, int i)
  {
          a[i] = b[i] + c[i];
  }

--------------------------------------------
for (int i = 0; i < N; ++i) {
     add_vec(i);
}
```

A Scalar Argument

```
.B5.1:                          ; Preds .B5.0
        movups    xmm1, XMMWORD PTR [ecx+edi*4]
        movups    xmm0, XMMWORD PTR [edx+edi*4]
        addps     xmm1, xmm0
        movups    XMMWORD PTR [eax+edi*4], xmm1
        ret
```

```
.B2.1:                          ; Preds .B2.0
        push      ebp
        mov       ebp, esp
        and       esp, -16
        sub       esp, 32
        lea       edx, DWORD PTR [1+eax]
        movaps    XMMWORD PTR [16+esp], xmm6
        lea       ecx, DWORD PTR [2+eax]
        movd      xmm6, eax
        add       eax, 3
        movaps    XMMWORD PTR [esp], xmm7
        movd      xmm3, edx
        punpckldq xmm6, xmm3
        movd      xmm7, ecx
        movd      xmm3, eax
        punpckldq xmm7, xmm3
        shufps    xmm6, xmm7, 136
        pslld     xmm6, 2
        paddd     xmm1, xmm6
        paddd     xmm2, xmm6
        movd      edx, xmm1
        paddd     xmm0, xmm6
        pshuflw   xmm7, xmm1, 238
        punpckhqdq xmm1, xmm1
        movd      ecx, xmm7
        movd      eax, xmm1
        pshuflw   xmm1, xmm1, 238
        movd      xmm3, DWORD PTR [edx]
        movd      edx, xmm1
        movd      xmm7, DWORD PTR [ecx]
        punpckldq xmm3, xmm7
        movd      xmm7, DWORD PTR [eax]
        movd      xmm1, DWORD PTR [edx]
        movd      ecx, xmm2
        punpckldq xmm7, xmm1
        shufps    xmm3, xmm7, 136
        pshuflw   xmm7, xmm2, 238
        punpckhqdq xmm2, xmm2
        movd      eax, xmm7
        movd      edx, xmm2
        pshuflw   xmm2, xmm2, 238
        movd      xmm1, DWORD PTR [ecx]
        movd      ecx, xmm2
        movd      xmm7, DWORD PTR [eax]
        punpckldq xmm1, xmm7
        movd      xmm7, DWORD PTR [edx]
        movd      xmm2, DWORD PTR [ecx]
        punpckldq xmm7, xmm2
        shufps    xmm1, xmm7, 136
        movd      eax, xmm0
        addps     xmm3, xmm1
        pshuflw   xmm6, xmm0, 238
        punpckhqdq xmm0, xmm0
        movd      ecx, xmm0
        pshuflw   xmm0, xmm0, 238
        movd      DWORD PTR [eax], xmm3
        movd      edx, xmm6
        movd      eax, xmm0
        pshuflw   xmm1, xmm3, 238
        punpckhqdq xmm3, xmm3
        pshuflw   xmm0, xmm3, 238
        movaps    xmm6, XMMWORD PTR [16+esp]
        movaps    xmm7, XMMWORD PTR [esp]
        movd      DWORD PTR [edx], xmm1
        movd      DWORD PTR [ecx], xmm3
        movd      DWORD PTR [eax], xmm0
        mov       esp, ebp
        pop       ebp
```

Vector Length

- How many elements should be processed in each invocation
  - The "vector length"
- The VL needs to be determined independently and consistently at the call sites and at the definition.
- Default: size of HW register / size of return type

- What if: size of return type ≠ size of prevalently used type inside the function

```
__declspec (vector)
float add_vec (float x, float y)
{
    return x+y;
}

--------------------------------
__declspec(vector)
double add_vec(double x, double y)
{
    return x+y;
}
```

```
__declspec (vector)
double add_vec (double x, double y)
{
    return sinf((float) x)
          +sinf((float) y);
}
```

Track Fitting

- Track finding involves associating a set of readings with the likely trajectory of a specific particle. Track fitting then takes those sets and determines a particle's position, direction and the magnitude of its momenta at any time by fitting the readings to a mathematical description of the trajectory.

- A charged particle moving in a homogeneous magnetic field experiences a sideways force (the Lorentz force) proportional to the strength of the magnetic field, the component of the velocity that is perpendicular to the magnetic field and the charge of the particle. In this way, the trajectory
- such a particle follows is helical along an axis parallel to the direction of the magnetic field.

- This perfectly helical behaviour is a simplification, as the magnetic field is rarely homogeneous, which deforms the helix. Also, as the particle moves, it is subject to multiple Coulomb scattering, which introduces variances in the momentum and makes the helical trajectory less crisp.
- Finally, the particle loses energy as it moves, and correspondingly the radius of the helix it describes contracts.

RTM Stencil

Description

Stencil computation is the basis for the reverse time migration algorithm in seismic computing. The underlying mathematical problem is to solve the wave equation using finite difference method. This benchmark computes a 25-point 3D stencil.

Mathematical Details

It's essentially a 3D convolution with a small compact operator. It's quite stable numerically.

Pseudocode

```
void loop_stencil(int t0, int t1, int x0, int x1, int y0, int y1, int z0, int z1)
{
    // March forward in time
    for(int t = t0; t < t1; ++t) {
      // March over 3D Cartesian grid
      for xyz in [x0,x1)×[y0,y1)×[z0,z1] do {
         A'[xyz] = 25-point stencil applied to A, centered at point xyz.
      }
   }
}
```

AOBench

AOBench is a popular visual compute benchmark that has been ported to dozens of programming models across dozens of platforms. Although not truly representative as a contemporary real time rendering approach, AOBench's per pixel computations, ray casting, and object intersection tests, are quite similar to the computations often performed in advanced pixel shaders of high performance real time rendering engines.

Computationally, AOBench can be described as a simple ray trace kernel applied to a fixed test scene of 3 spheres and 1 plane. For each primary ray that intersects an object, a simple ambient occlusion approximation is computed by random ray casting back into the scene.

Binomial Lattice

Option pricing is the problem of computing the expected present value of a financial instrument (most usually stocks, but also interest rates, foreign exchange rates, bonds, etc). This is based on a forecast of cashflows over a specific time horizon. The expected present value is used to determine the fair premium of an option on that instrument.

The binomial (tree or lattice) option pricing model is a confluent discretization of a Brownian motion process. Note that for a large number of steps the binomial distribution approximates a Gaussian distribution. The discretization takes the form of a recombinant tree, where each level of the tree represents the set of values the underlying can take at specific points in time in the lifetime of the option.

The dataflow of this algorithm is shown in the figure to the left, and pseudocode is given in the next section. Values at the base of the lattice are computed first, and then propagated up the lattice. The expected present value is computed in the topmost node.

Although this algorithm uses the (simplistic) Brownian motion assumption, and even then approximates it with a binomial distribution, the binomial lattice option pricing model provides a relatively close approximation to the expected present value for a variety of derivatives and underlying assets, for example early-exercise, path-dependent and log-normally distributed underlying derivatives. It also serves as the basis of more elaborate option pricing algorithms, such as the trinomial tree. In addition, this algorithm is an example of a 2D recurrence which also appears in many other algorithms, such as infinite impulse response filters and matrix factorization.

Measurement System Configuration

Operating System	**Windows Server 2008 R2 Enterprise – Service Pack 1**
Processor	Intel® Xeon® CPU X7560 @ 2.27Ghz (4 processors)
Installed Memory (RAM)	64.0 GB
System Type	64-bit Operating System
Computer name	Fxe32win02.amr.corp.intel.com
Intel Compiler	Intel(R) C++ Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 12.10.233 Build 20110811
Microsoft Compiler	Microsoft Visual Studio 2010, Version 10.0.31118.1.SP1Rel