Robert Geva

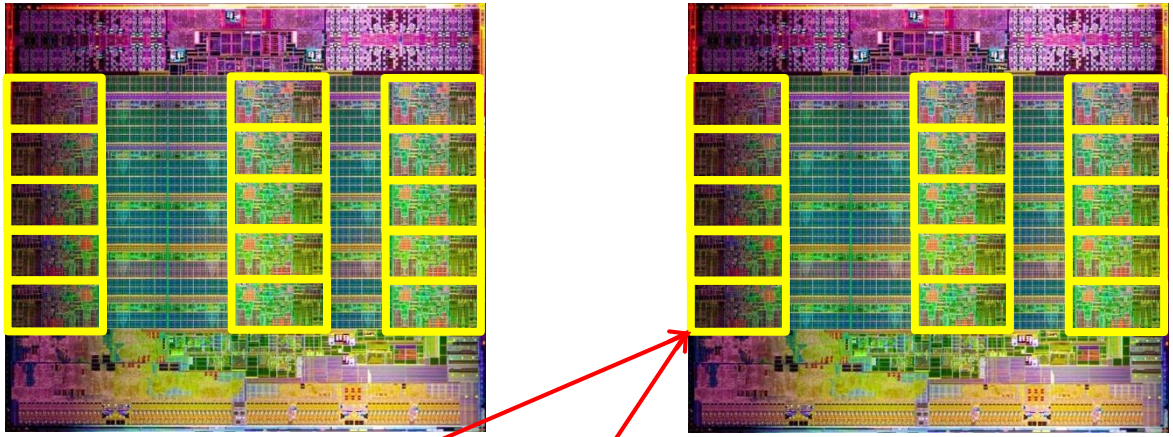
Parallel Computing In C++: A view from Intel

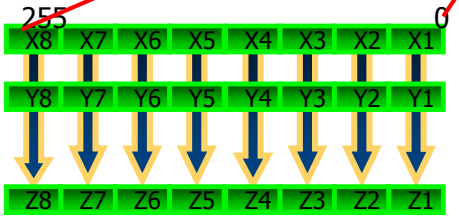
Topics

- Hardware
  - Multi-core
  - SIMD
  - Heterogeneous processing
  - Heterogeneous memory
- programmability
  - TBB for shared memory parallelism
  - SIMD in OpenMP, what is needed in C++
  - TBB heterogeneous and distributed flow graph
  - Data layout template

Objective: to be able to do everything in C++

A multi core processor



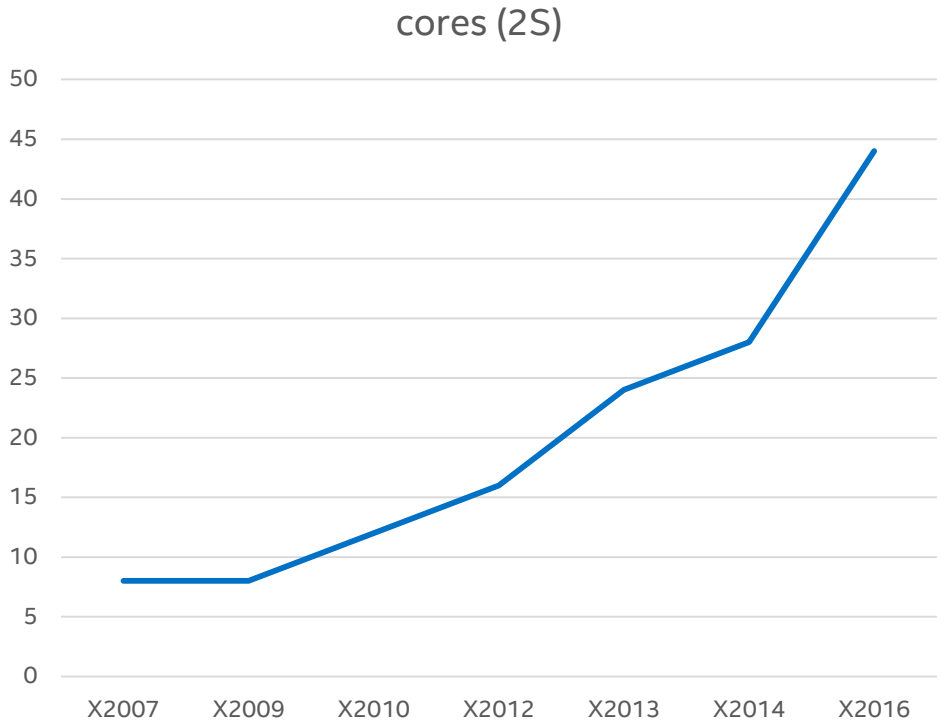


Most CPUs have two levels of parallelism: cores and vectors.

Modernization needs to enable both and also the memory hierarchy

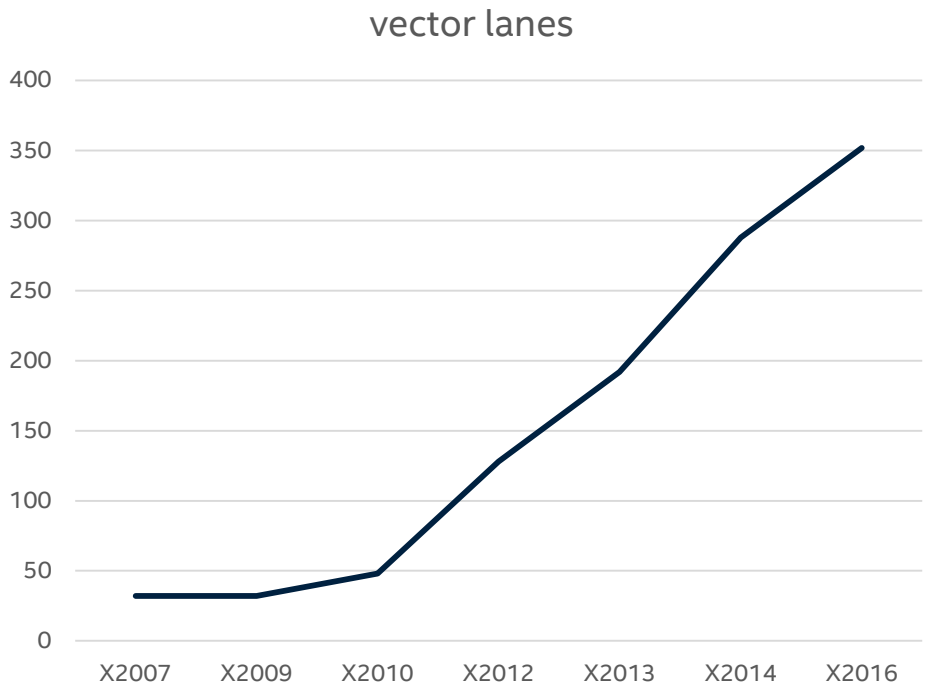
Growth of core count over time

Year	cores (2S)
X2007	8
X2009	8
X2010	12
X2012	16
X2013	24
X2014	28
X2016	44

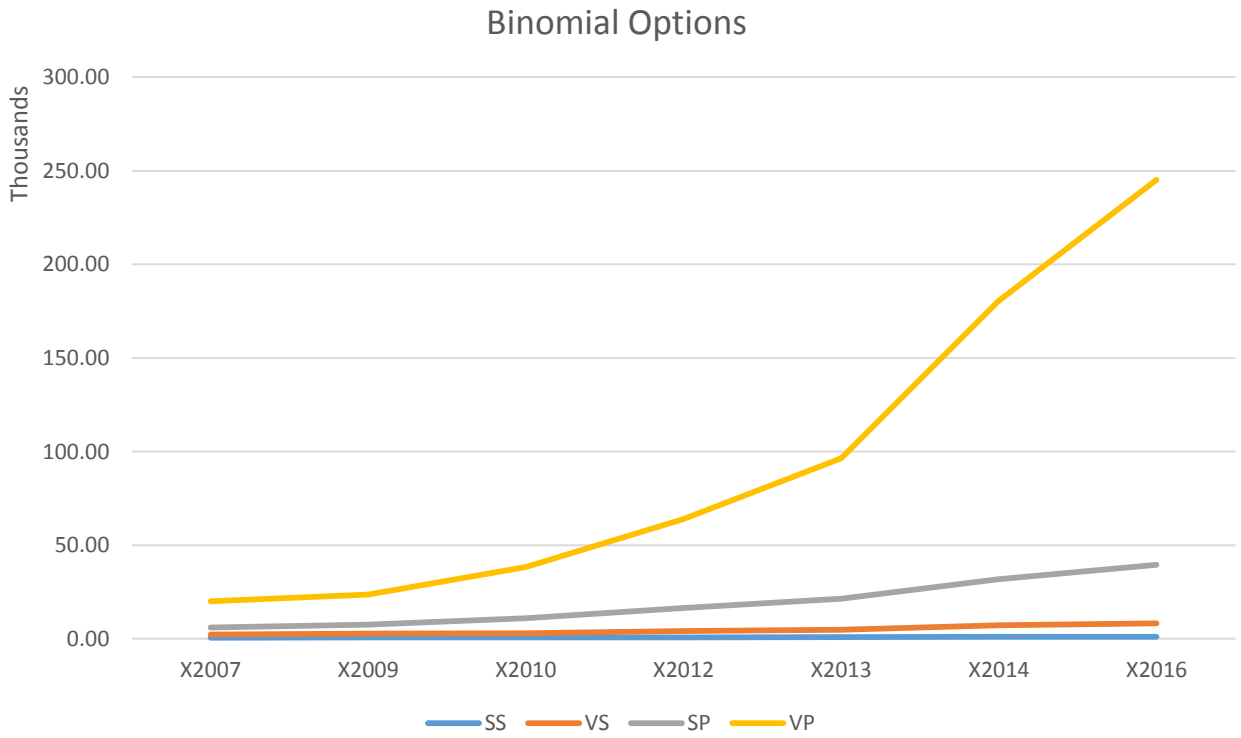


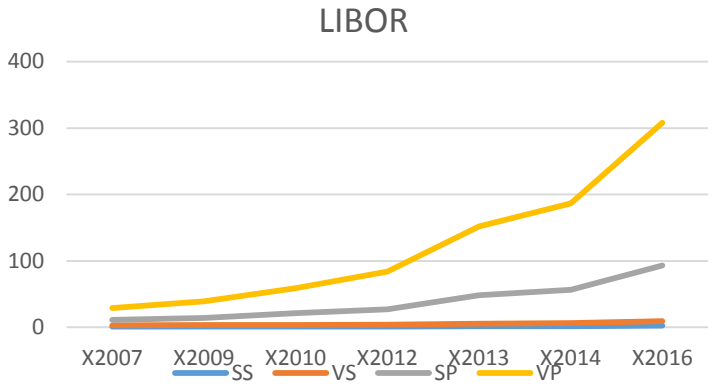
Growth of Vector Lanes over Time

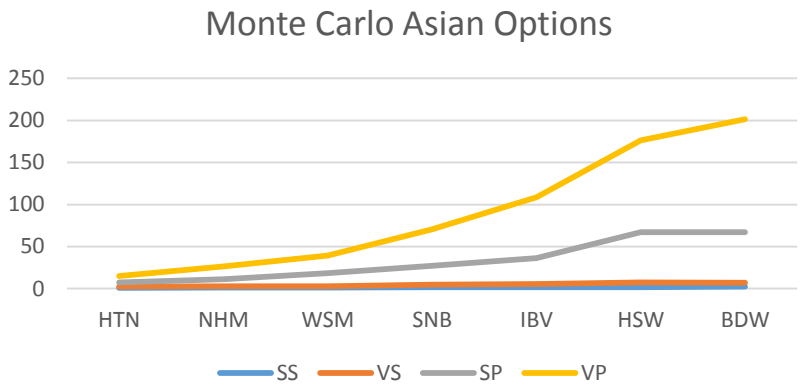
	Cores	SIMD	LANES
X2007	8	128	32
X2009	8	128	32
X2010	12	128	48
X2012	16	256	128
X2013	24	256	192
X2014	36	256	288
X2016	44	256	352

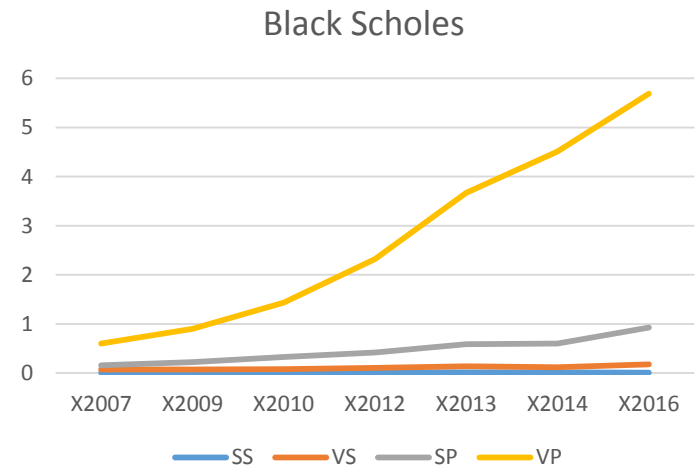


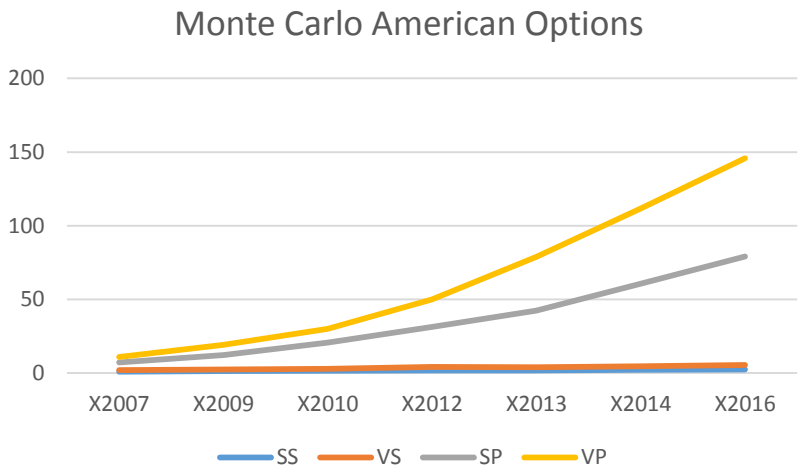
Impact of parallelism

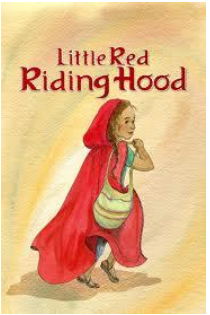












Myth: vector code executes in vector lanes

Vector Instructions are Sometimes Smarter (not just wider)

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
#define SAT2SI16(x) \
MAX(MIN((x),32767),-32768)
short A[N];

for (i=0; i<n; i++) {
    A[i] = SAT2SI16(A[i]+B[i]);
}
```

Saturating Add

```
movsx     r11d, [rdx+r9*2]
movsx     ebx, [r8+r9*2]
add       r11d, ebx
cmp       r11d, 32767
cmovge    r11d, eax
cmp       r11d, -32768
cmovl     r11d, ecx
mov       [rdx+r9*2], r11w
inc       r9
cmp       r9, r10
jb        .B1.8
```

11 insts / 1 elem

```
movdqa    xmm0, [rdx+rax*2]
paddsw    xmm0,  [r8+rax*2]
movdqa    [rdx+rax*2], xmm0
add       rax, 8
cmp       rax, r9
jb        .B1.4
```

6 insts / 8 elems

Write Vector Code Only Once and Recompile for New Targets

```
#define ABS(X) \
   ((X) >= 0? (X) : -(X))
int A[1000]; double B[1000];
void foo(int n){
  int i;
  for (i=0; i<n; i++){
    B[i] += ABS(A[i]);
  }
}
```

-O2 →

```
movq      xmm1, [A+r9+rax*4]
pxor      xmm0, xmm0
pcmpgtd   xmm0, xmm1
pxor      xmm1, xmm0
psubd     xmm1, xmm0
cvtdq2pd  xmm2, xmm1
addpd     xmm2, [B+r9+rax*8]
movaps    [B+r9+rax*8], xmm2
add       rax, 2
cmp       rax, rcx
jb        .B1.4
```

ABS sequence

2 elements

-O2 -QxAVX ↓

```
vpabsd    xmm0, [A+r9+rax*4]
vcvtdq2pd ymm1, xmm0
vaddpd    ymm2, ymm1,
    [B+r9+rax*8]
vmovupd   [B+r9+rax*8], ymm2
add       rax, 4
cmp       rax, rcx
jb        .B1.4
```

4 elements

-QxSSSE3 ↘

```
movq      xmm0, [A+r9+rax*4]
pabsd     xmm1, xmm0
cvtdq2pd  xmm2, xmm1
addpd     xmm2, [B+r9+rax*8]
movaps    [B+r9+rax*8], xmm2
add       rax, 2
cmp       rax, rcx
jb        .B1.4
```

ABS instruction

2 elements

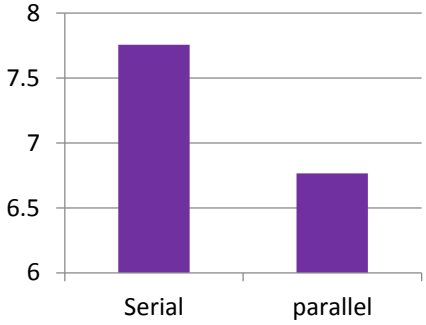
Super Linear speedup with vectors

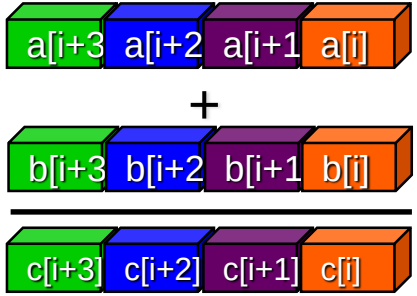
Black Scholes double precision:

>4X speed up with 4 doubles in an AVX register

Vector execution is not a naïve side by side execution of scalar code

- Just enough registers to eliminate spills / fills
- Save on (math) function calls
- Register calling conventions





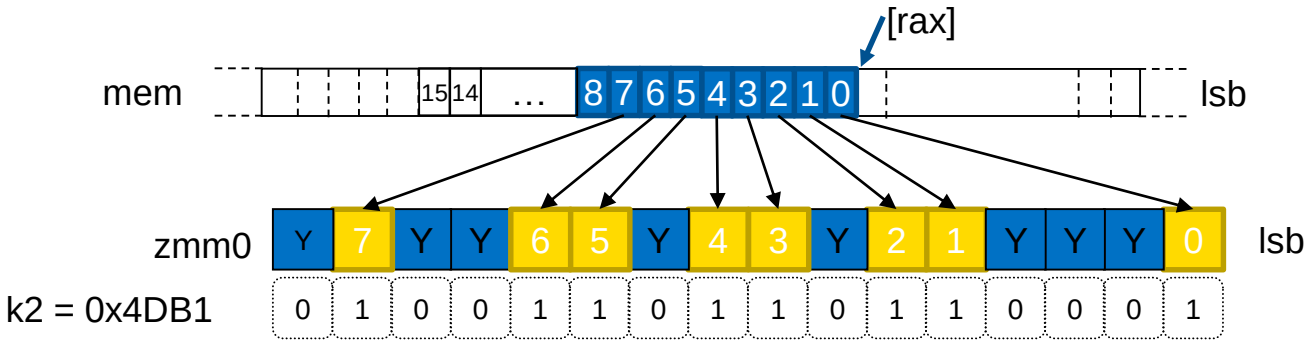
Expand & Compress

Allows vectorization of conditional loops
- Opposite operation (compress) in AVX512F
- Similar to FORTRAN pack/unpack intrinsics
- Provides mem fault suppression
- Faster than alternative gather/scatter

```
for(j=0, i=0; i<N; i++)
{
  if(X[i] != 0.0)
  {
     B[i] = A[i] * C[j++];
  }
}
```

VEXPANDPS zmm0 {k2}, [rax]

Moves compressed (consecutive) elements in register or memory to sparse elements in register (controlled by mask), with merging or zeroing



Conflict Detection

Sparse computations are common in HPC, hard to vectorize due to race conditions

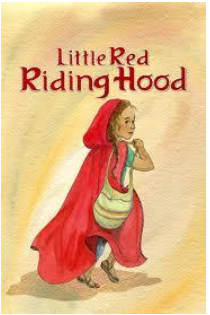
Consider the "histogram" problem:

```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

Wrong implementation

```
index = vload &B[i]                    // Load 16 B[i]
old_val = vgather A, index             // Grab A[B[i]]
new_val = vadd old_val, +1.0           // Compute new values
vscatter A, index, new_val             // Update A[B[i]]
```

```
index = vload &B[i]                                  // Load 16 B[i]
pending_elem = 0xFFFF;                               // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index            // Grab A[B[i]]
   new_val = vadd old_val, +1.0                      // Compute new values
   vscatter A {curr_elem}, index, new_val            // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem           // remove done idx
} while (pending_elem)
```

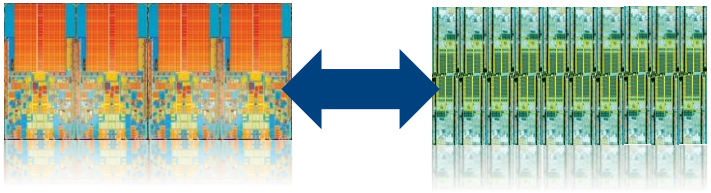


Reality: vector code executes single instruction multiple data, different calling conventions, support for “cross lane” operations.

Heterogeneous Systems

Distributed memory, both sides are CPUs, running OS.

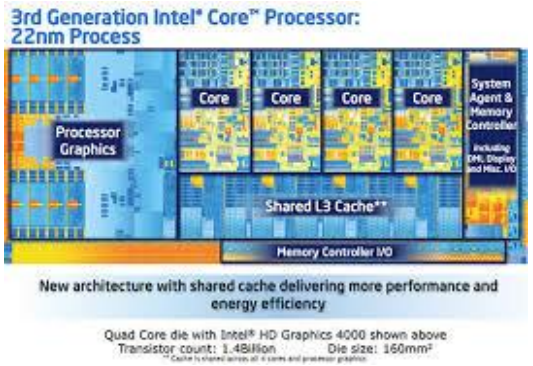
ISA exposed to SW, not the same



Shared memory. GPU.

Virtual ISA exposed to SW.

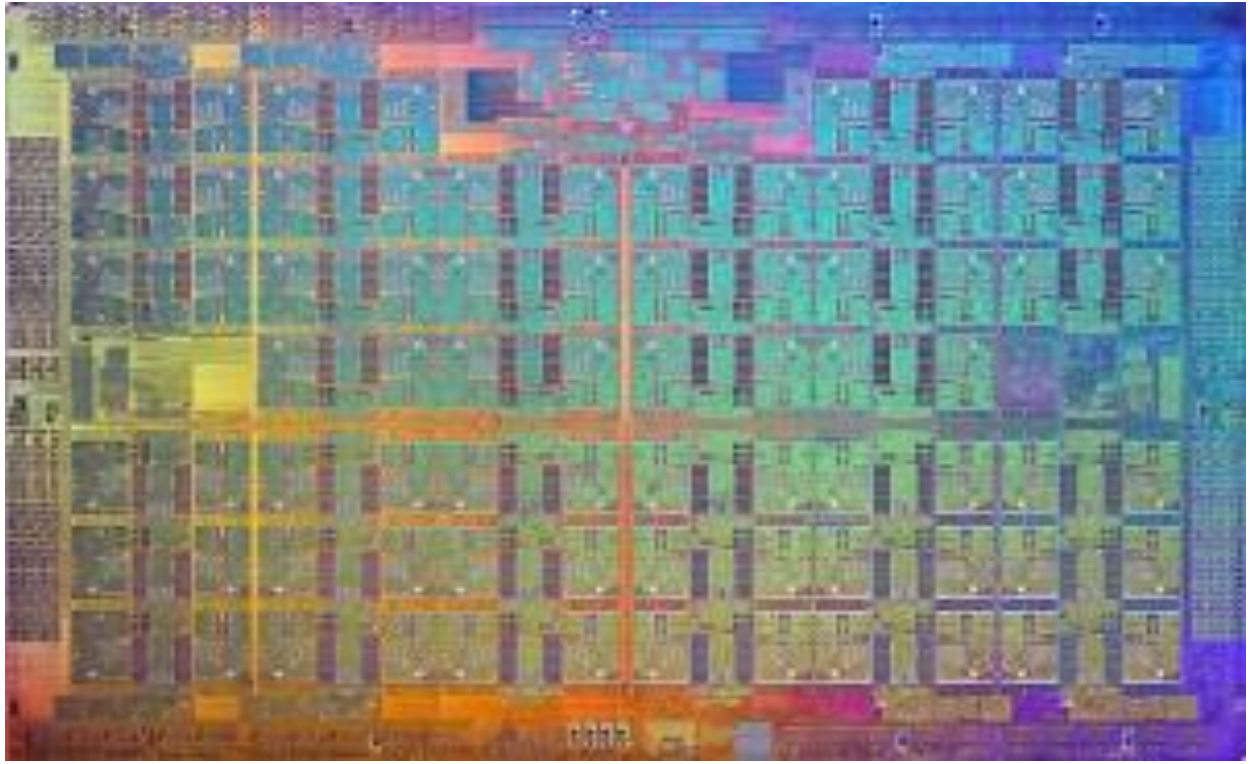
GPU is SIMD



Current support for heterogeneity: C++ for parallelism on either side, OpenMP for heterogeneity via #pragma omp offload.

By default, code is compiled for the CPU host.
Annotation required to compile code for the co-processor / GPU.

Knights Landing



KNL Architecture Overview

ISA

Intel® Xeon® Processor Binary-Compatible (w/Broadwell)

On-package memory

Up to 16GB, ~460 GB/s STREAM at launch

Platform Memory

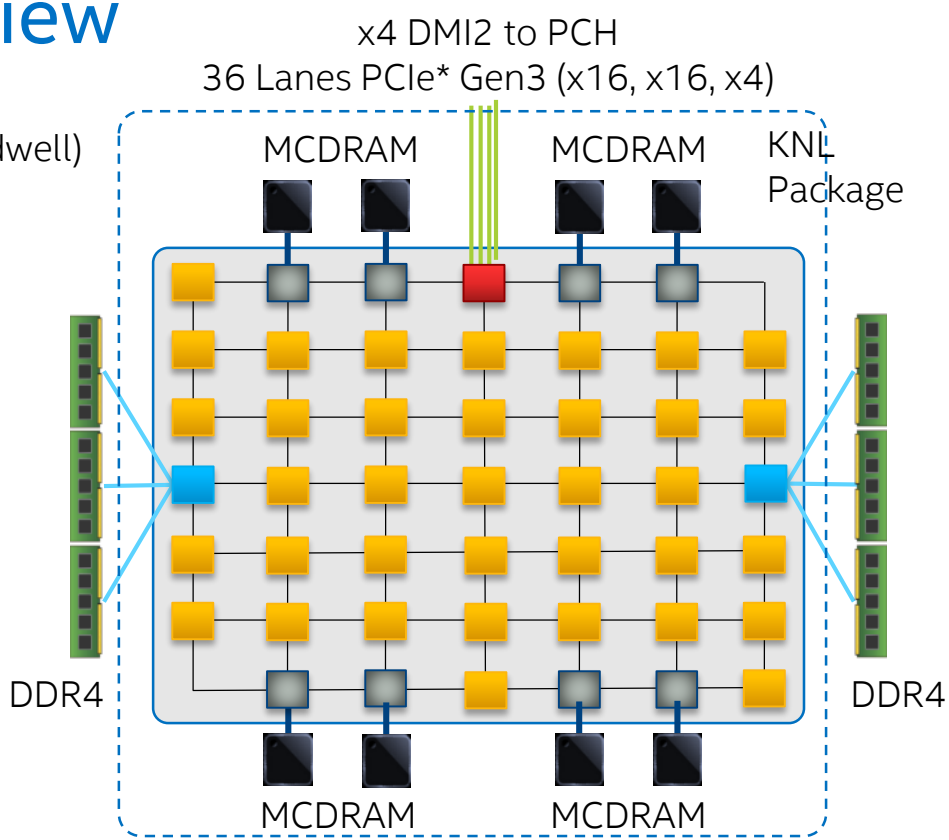
Up to 384GB (6ch DDR4-2400 MHz)

Fixed Bottlenecks

- ✓ 2D Mesh Architecture
- ✓ Out-of-Order Cores
- ✓ 3X single-thread vs. KNC

TILE:
(up to 36)

Enhanced Intel® Atom™ cores based on Silvermont Microarchitecture



Tile

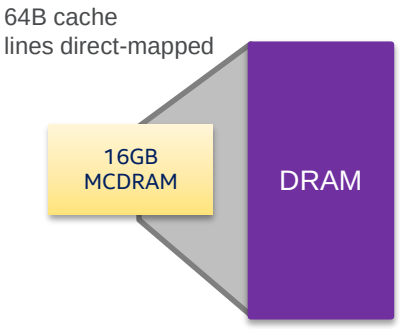
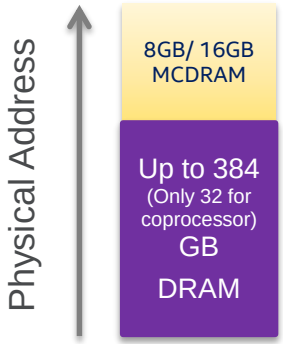
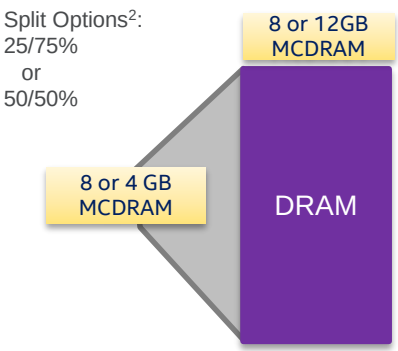
EDC (embedded DRAM controller)

IMC (integrated memory controller)

IIO (integrated I/O controller)

Integrated On-Package Memory Usage Models

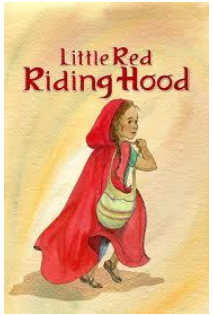
Model configurable at boot time and software exposed through NUMA[1]

	Cache Model	Flat Model	Hybrid Model
			
Description	Hardware automatically manages the MCDRAM as a "L3 cache" between CPU and ext DDR memory	Manually manage how the app uses the integrated on-package memory and external DDR for peak perf	Harness the benefits of both Cache and Flat models by segmenting the integrated on-package memory
Usage Model	▪ App and/or data set is very large and will not fit into MCDRAM ▪ Unknown or unstructured memory access behavior	▪ App or portion of an app or data set that can be, or is needed to be "locked" into MCDRAM so it doesn't get flushed out	▪ Need to "lock" in a relatively small portion of an app or data set via the Flat model ▪ Remaining MCDRAM can then be configured as Cache

1. NUMA = non-uniform memory access
2. As projected based on early product definition

Topics

- Hardware
  - Multi-core
  - SIMD
  - Heterogeneous processing
  - Heterogeneous memory
- programmability
  - TBB for shared memory parallelism
  - TBB heterogeneous and distributed flow graph
  - Data layout template
  - SIMD in OpenMP, what is needed in C++



Myth: To write a parallel program, write a sequential program and indicate where you give permission for parallelism

Example: A lopsided loop

Assume execution where expensive calc is called once per vector loop.

All lanes that execute inexpensive calc are held back, and execute as slow as the expensive calc.

Optimization: rewrite so that all expensive calcs are consecutive, and inexpensive calcs are consecutive.

The main loops speed-up for all HW targets.

The overhead is vectorizeable using compress / expand.

```
#pragma omp simd
  for (int x = 0; x < N; ++x) {
    double val = in[x];
    if (val == 0.0){
      results[x] = expensive_calc(val);
    }else {
     results[x] = inexpensive_calc(val);
    }
  }
```

Redesign the loop: Partition By Weight

```
for (int x = 0; x < N; ++x) {
  double val = in[x];
  int mask_local = val == 0.0;
  mask[x] = mask_local;
  if(mask_local){
    vecX[cnt] = val; //compress
    cnt++;
  }
}

#pragma omp simd
for (int y = 0; y < cnt; ++y) {
  vecX[y] = expensive_calc(vecX[y]);
}
cnt = 0;
```

```
for (int x = 0; x < N; ++x) {
  double val = in[x];
  if(__builtin_expect(mask[x],0))
    results[x] = vecX[cnt++]; //expand
  else
    results[x] = inexpensive_calc(val);
}
```

With vector length of 8, gains of 8.3X using AVX512

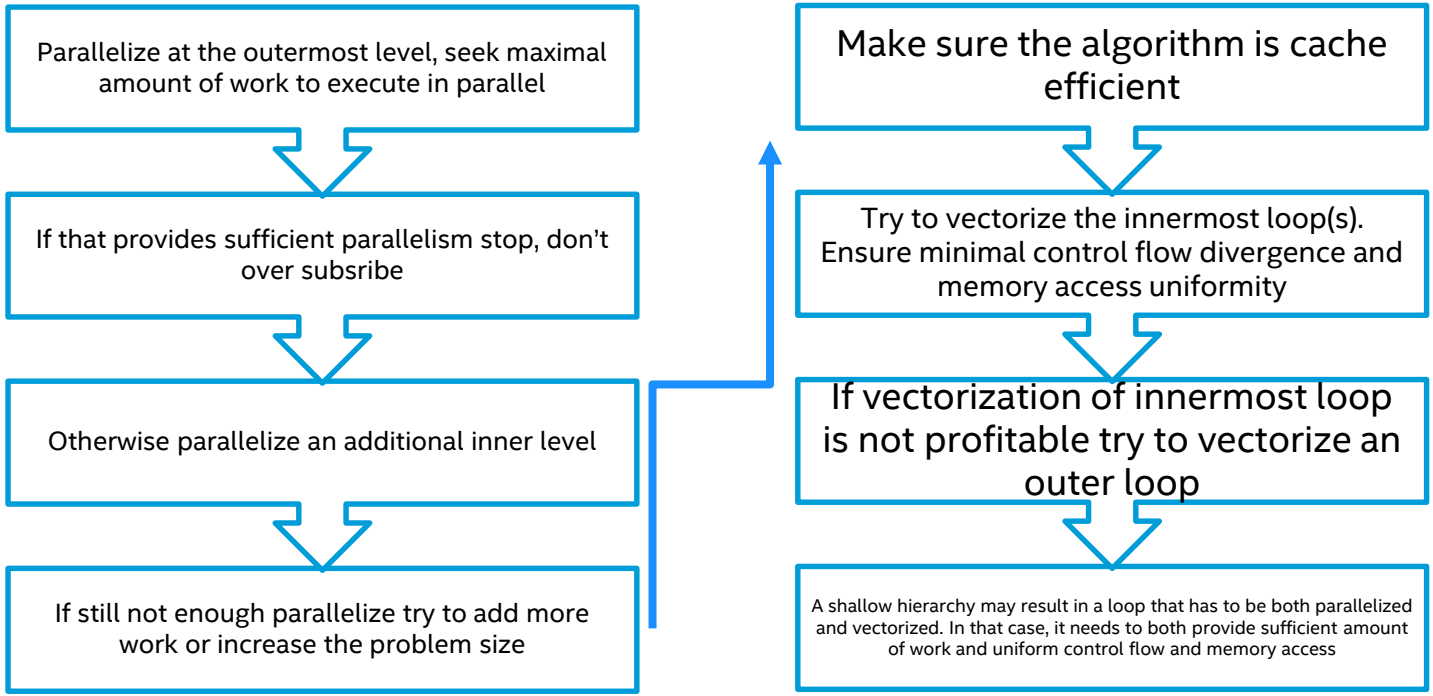
Reality: parallel code needs to be designed for efficient execution.

When parallelizing sequential code, the amount of refactoring is unbounded.

We start from design principles and look for programming solutions that support them.

The #1 best practice: parallelize coarse grain, vectorize fine grain

#1 Best Practice in Parallelizing a Loop Hierarchy



Vectorize Innermost, Parallelize Outermost (VIPO)

Threading Building Blocks: Rich Feature Set for Parallelism

- Parallel algorithms and data structures
- Threads and synchronization
- Memory allocation and task scheduling

Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch.

Flow Graph

A set of classes to express parallelism as a graph of compute dependencies and/or data flow

Concurrent Containers

Concurrent access, and a scalable alternative to serial containers with external locking

Synchronization Primitives

Atomic operations, a variety of mutexes with different properties, condition variables

Task Scheduler

Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph

Thread Local Storage

Unlimited number of thread-local variables

Threads

OS API wrappers

Miscellaneous

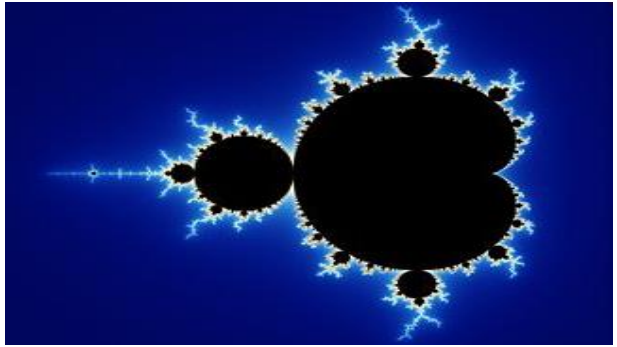
Thread-safe timers and exception classes

Memory Allocation

Scalable memory manager and false-sharing free allocators

Mandelbrot Example

Intel® Threading Building Blocks (Intel® TBB)



```
int mandel(Complex c, int max_count) {
  int count = 0; Complex z = 0;
  for (int i = 0; i < max_count; i++) {
    if (abs(z) >= 2.0) break;
    z = z*z + c; count++;
  }
  return count;
}
```

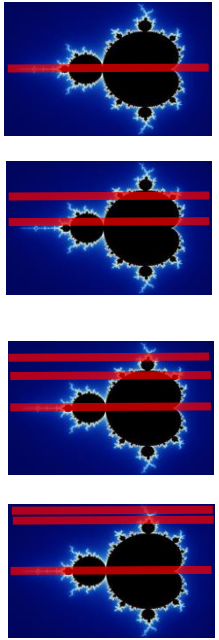
Task is a function object

```
parallel_for( 0, max_row,
  [&](int i) {
    for (int j = 0; j < max_col; j++)
      p[i][j]=mandel(Complex(scale(i),scale(j)),depth);
  }
);
```

Parallel algorithm

Use C++ lambda functions to define function object in-line

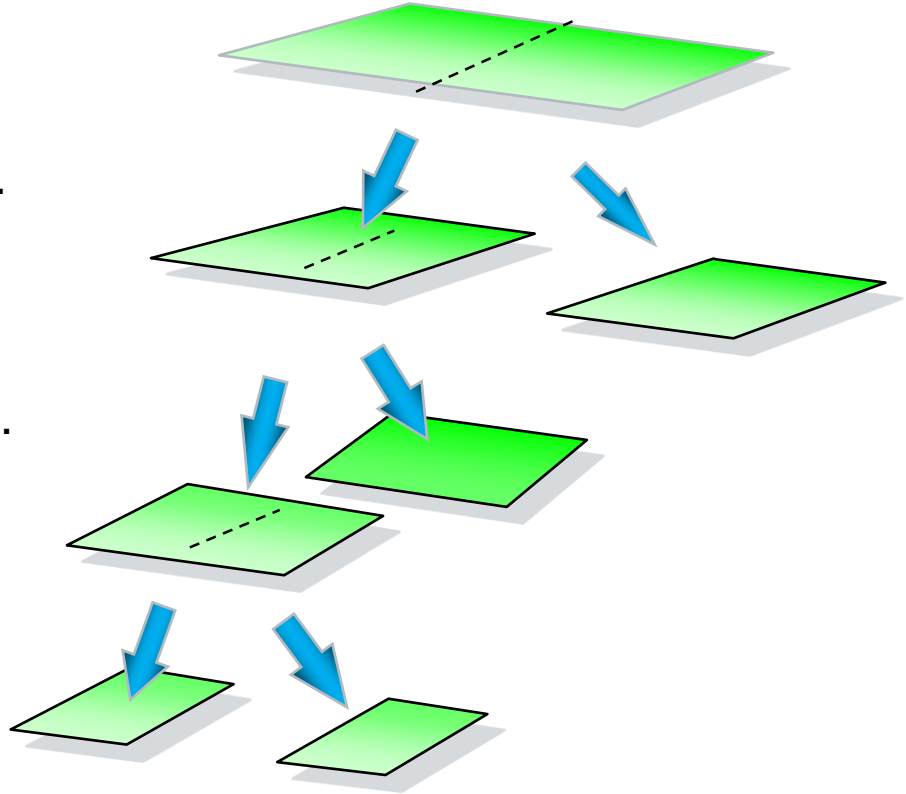
A parallel_for recursively divides the range into subranges that execute as tasks - Intel® Threading Building Blocks (Intel® TBB)



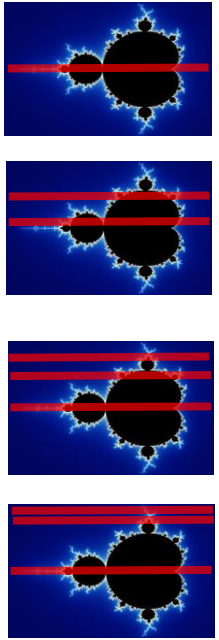
Split range...

.. recursively...

...until ≤ grainsize.



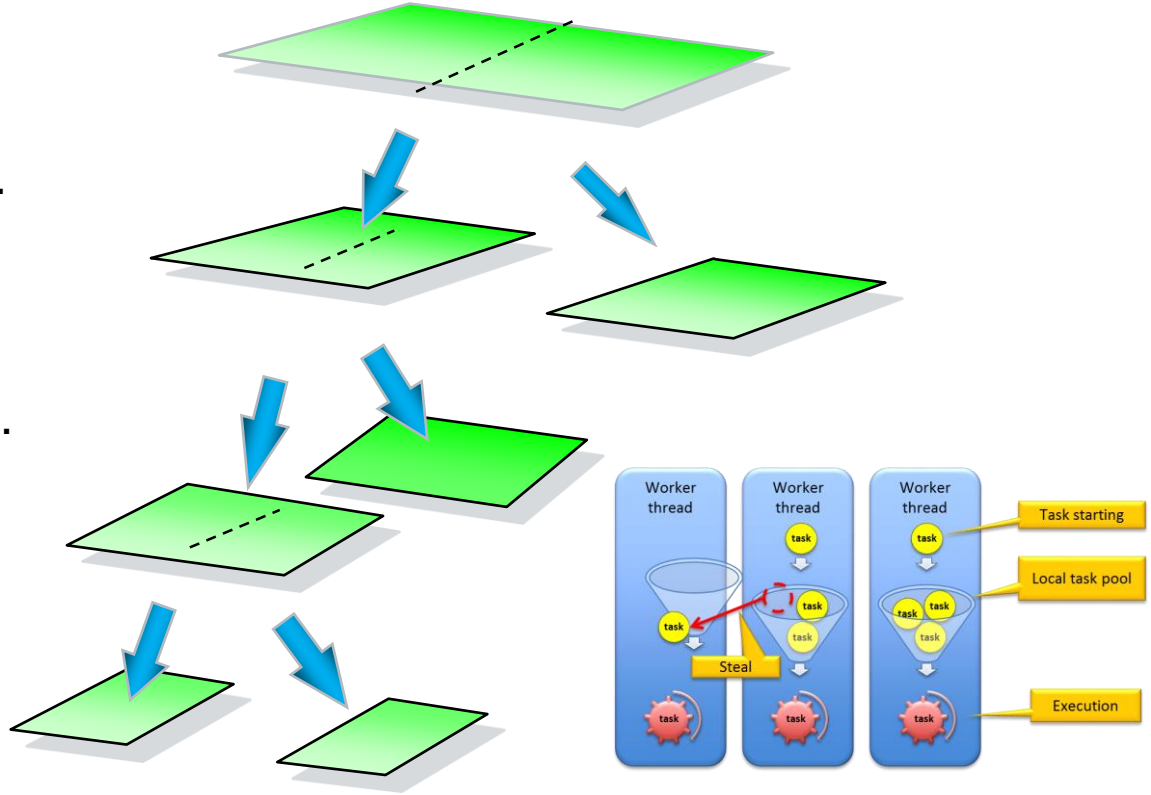
A parallel_for recursively divides the range into subranges that execute as tasks - Intel® Threading Building Blocks (Intel® TBB)



Split range...

.. recursively...

...until ≤ grainsize.



Generic Algorithms

Loop parallelization

parallel_for

parallel_reduce

- load balanced parallel execution
- fixed number of independent iterations

parallel_scan

- computes parallel prefix

  y[i] = y[i-1] op x[i]

Parallel sorting

parallel_sort

Parallel function invocation

parallel_invoke

- Parallel execution of a number of user-specified functions

Parallel Algorithms for Streams

parallel_do

- Use for unstructured stream or pile of work
- Can add additional work to pile while running

parallel_for_each

- parallel_do without an additional work feeder

pipeline / parallel_pipeline

- Linear pipeline of stages
- Each stage can be parallel or serial in-order or serial out-of-order.
- Uses cache efficiently

Computational graph

flow::graph

- Implements dependencies between nodes
- Pass messages between nodes

Scalable Memory Allocation

Problem

- Memory allocation is a bottle-neck in concurrent environment
  - Threads have to acquire a global lock to allocate / deallocate memory from the global heap

Solution

- Intel® TBB provides tested, tuned, and scalable memory allocator based on per-thread memory management
- Convenient interfaces:
  - As an *allocator* argument to STL template classes
  - As a replacement for malloc/realloc/free calls (C programs)
  - As a replacement for global *new* and *delete* operators (C++ programs)

Concurrent containers provide thread-safe, scalable alternatives to STL*

STL is not concurrency-friendly. For example, suppose two threads each execute:

```
extern
std::queue q;

if(!q.empty())
{


item=q.front()
;
    q.pop();
}
```

At this instant, another thread might pop last element.

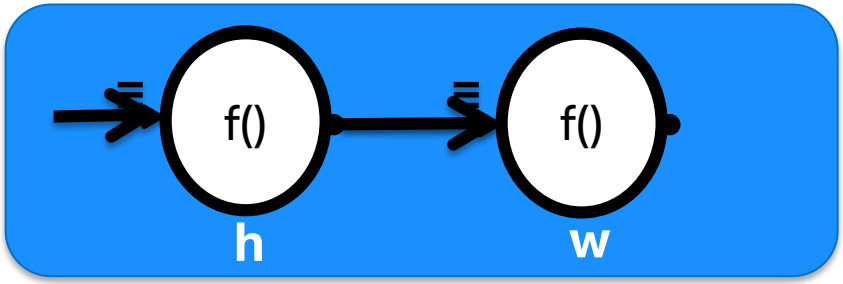
Solution: concurrent_queue has pop_if_present()

Other concurrent containers are concurrent_vector, concurrent_hash_map, and concurrent_unordered_map.

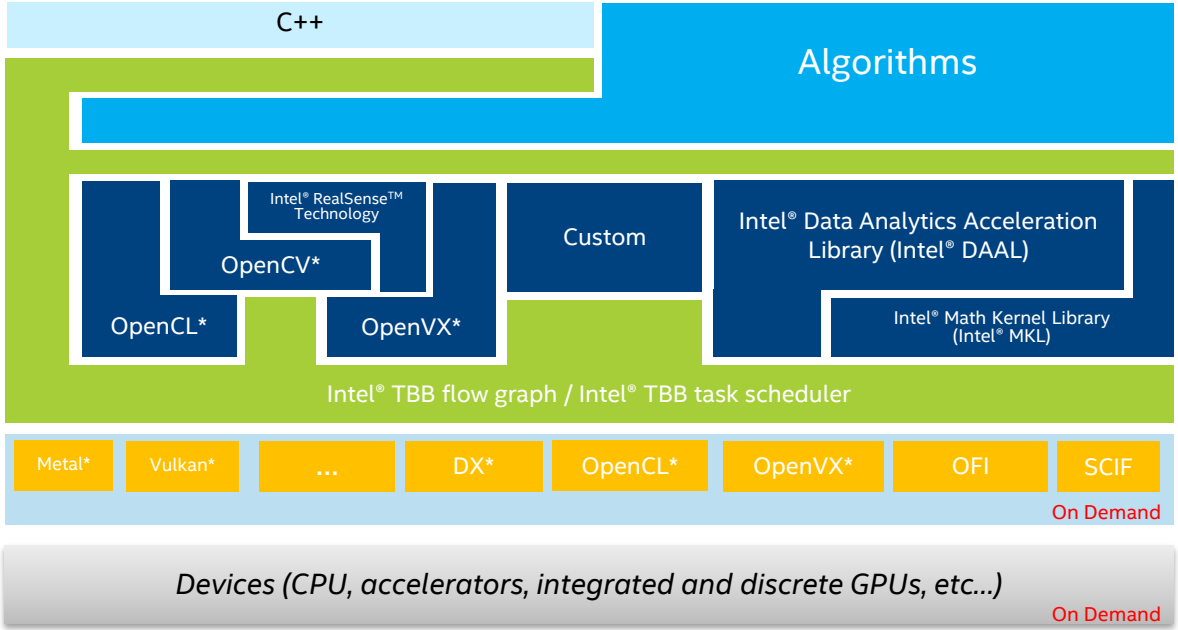
TBB Flow Graph Hello World Example

Users create nodes and edges, interact with the graph and wait for it to complete

```
tbb::flow::graph g;
tbb::flow::continue_node< tbb::flow::continue_msg >
  h( g, []( const continue_msg & ) { std::cout << "Hello "; } );
tbb::flow::continue_node< tbb::flow::continue_msg >
  w( g, []( const continue_msg & ) { std::cout << "World\n"; } );
tbb::flow::make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```

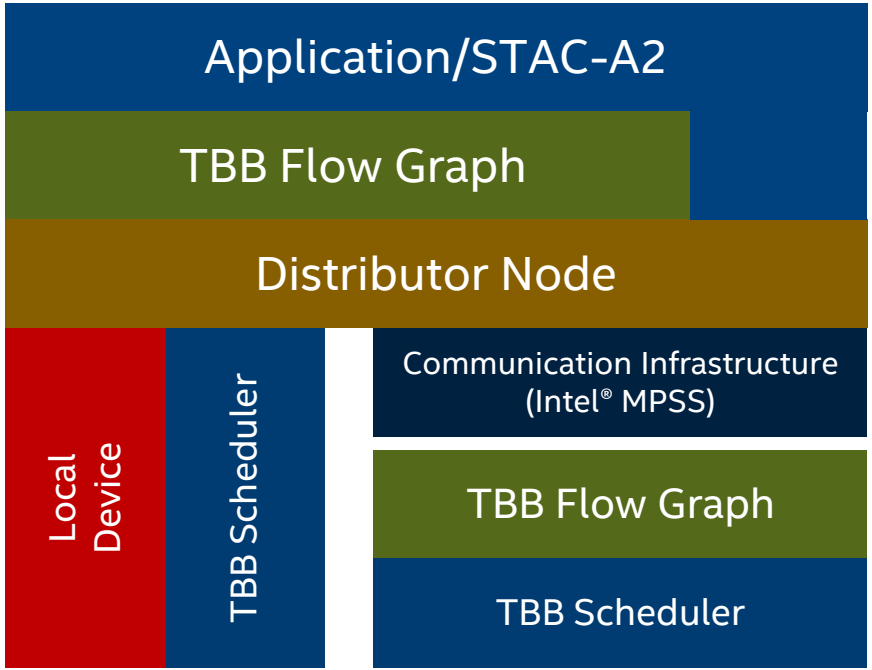


Heterogeneous flow Graph Vision: A coordination layer for heterogeneity that provides flexibility, retains optimization opportunities and composes with existing models



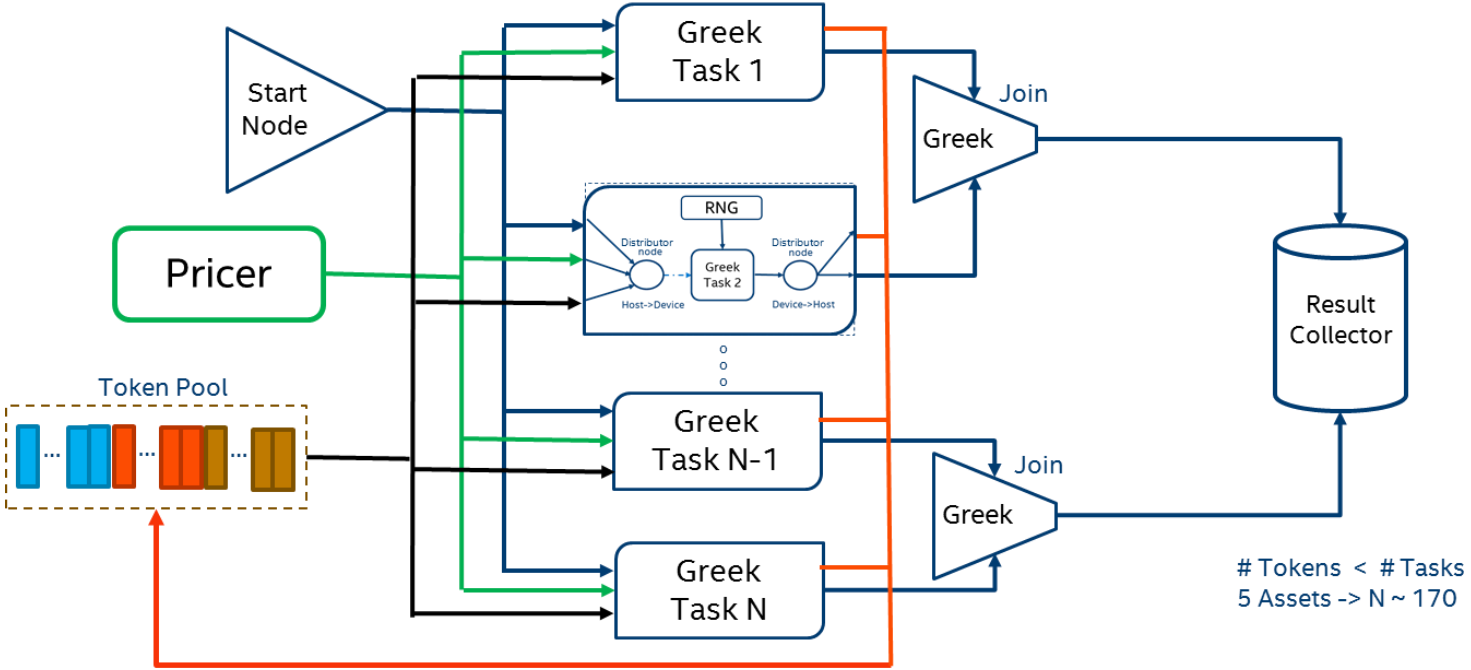
STAC-A2 Implementation with distributor_node

- Compute-intense analytic workload involved in pricing and risk management
- Implemented with Intel® TBB flow graph, TBB parallel algorithms and OpenMP vectorization
- Uses asynchronous support in flow graph, distributor_node, to offload to the Intel® Xeon Phi coprocessor
- Using a token-based system enables dynamic load balancing between CPU and coprocessor
- Which Greeks are calculated on which resource is determined dynamically based on resource availability



https://stacresearch.com/news/2015/11/03/stac-report-stac-a2-system-dual-xeon-phi-cards

Intel® TBB flow graph implementation of STAC-A2



https://stacresearch.com/news/2015/11/03/stac-report-stac-a2-system-dual-xeon-phi-cards

Compilation

Viral annotation required for code to be compiled for the co-processor

Topics

- Hardware
  - Multi-core
  - SIMD
  - Heterogeneous processing
  - Heterogeneous memory
- programmability
  - TBB for shared memory parallelism
  - TBB heterogeneous and distributed flow graph
  - Data layout template
  - SIMD in OpenMP, what is needed in C++

Capabilities in Vector Programming

```
#pragma omp simd
for(i = 0; i<N; ++i)
{
    a[i] = b[i]+c[i] ;
}
```

P007[56]R0: for_each(index) with VEC policy

1. Vector Loops
   a) The syntax means that a loop is a vector loop
   b) Used mostly at the application level
   c) Syntax can look like OpenMP*, Intel® Cilk™, Intel® Threading Building Blocks, etc.
   d) The loop is single threaded and consistent with SIMD execution
   e) Additional syntax for more capabilities
2. Vector Functions
   a) The function is compiled as if it is part of the body of a vector loop
   b) For use in larger projects and for libraries
   c) Organizations interested in methodological parallel programming
   d) Additional syntax for more capabilities

We have no solutions for SIMD function for C++ yet:

How do you call your own function from an STL algorithm with VEC policy

```
#pragma omp declare simd
vec_add (float *a,float *b,float *c int i)
{
    a[i] = b[i]+c[i] ;
}
```

P0076R0: Vector execution policy

```
for_loop( vec, 0, n, [&](int i) {
    A[i] = A[i+1] + B[i];
    C[i] -= 2*A[i];
});
```

OpenMP Equivalent

```
#pragma omp simd for
for( int i=0; i<n; ++i ) {
    A[i] = A[i+1] + B[i];
    C[i] -= 2*A[i];
}
```

A single threaded vector loops
Allow vectorization of loops that cannot be parallelized (forward dependencies)
Allow vectorization when multi-threading is undesired
Useful for CPUs with SIMD, not so much for GPUs with SIMT

Reduction

```
extern float s; extern int t;
for_loop( par, 0, n,
    reduction_plus(s),
    reduction_bit_and(t),
    [&](int i, float& s_, int& t_) {
        s_ += A[i]*B[i];
        t_ &= C[i];
    });
// s and t have final reduction values here.
```

OpenMP Equivalent

```
extern float s; extern int t;
#pragma omp parallel for reduction(+:s) reduction(&:t)
for( int i=0; i<n; ++i ) {
    s += A[i]*B[i];
    t &= C[i];
}
```

Induction Variables

```
extern int j, k;
for_loop( vec, n, 0,
    induction(j, jstep),
    induction(k, -kstep),
    [&](int i, int j_, int k_) {
        A[i] = B[j_]*C[k_];
    });
// j and k have correct final values here.
```

OpenMP 4.5 Equivalent

```
extern int j, k;
#pragma omp simd linear(j:jstep, k:-kstep)
for( int i=0; i<n; ++i) {
    A[i] = B[j]*C[k];
    j += jstep;
    k -= kstep;
}
```

Implementation

Write a for loop in the include file

Use #pragma omp simd with clauses as needed

to get the compiler to generate SIMD instructions

The proposal allows the following as future extensions

Scatter write: `a[b[x]] = d[x];`

Histogram: `a[b[x]]++;`

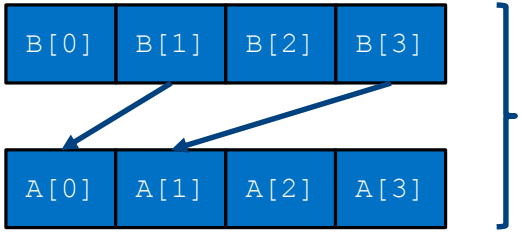
Expand: `if (c[i]) a[i] = b[i] * d[j++];`

Compress: `if (c[i]) a[j++] = b[i] * d[i];`

Source Level Example

Compress

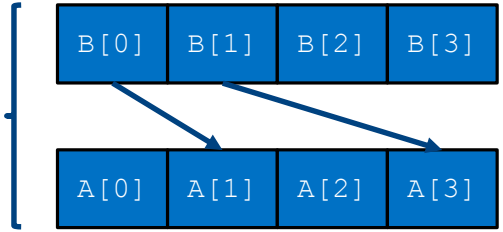
```
count = 0;
#pragma omp simd
for(i){
  if (cond(i)){
#pragma omp ordered simd
    {
      A[count] = B[i];//compress
      count++;
    }
  }
}
```



Expand

```
count = 0;
#pragma omp simd
for(i){
  if (cond(i)){
#pragma omp ordered simd
    {
      A[i] = B[count];//expand
      count++;
    }
  }
}
```

When cond(i) is
[F T F T]



The proposal does not cover vector functions

Arguments can be qualified as uniform / linear

Chunk size may be limited for correctness / performance reasons

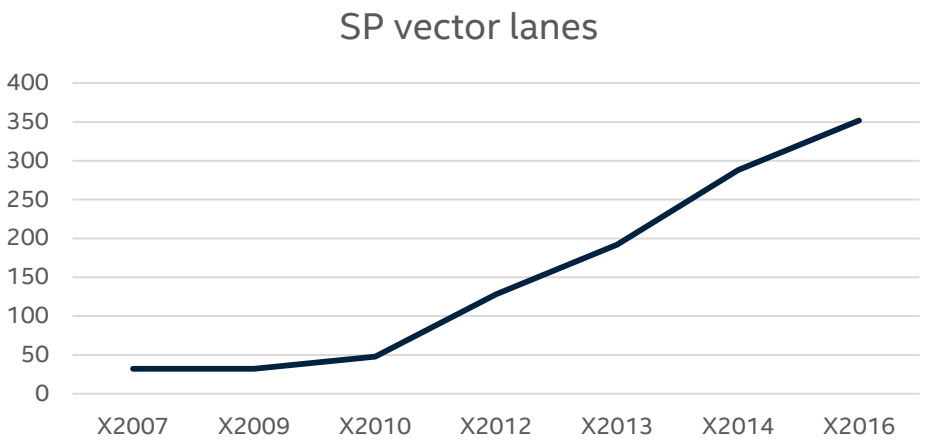
Hidden mask argument may be passed if the function is called under a conditional expression

Multiple variants may be required

Current gap: type system integration

```
#pragma omp declare simd
vec_add (float *a,float *b,float *c int i)
{
    a[i] = b[i]+c[i] ;
}
```

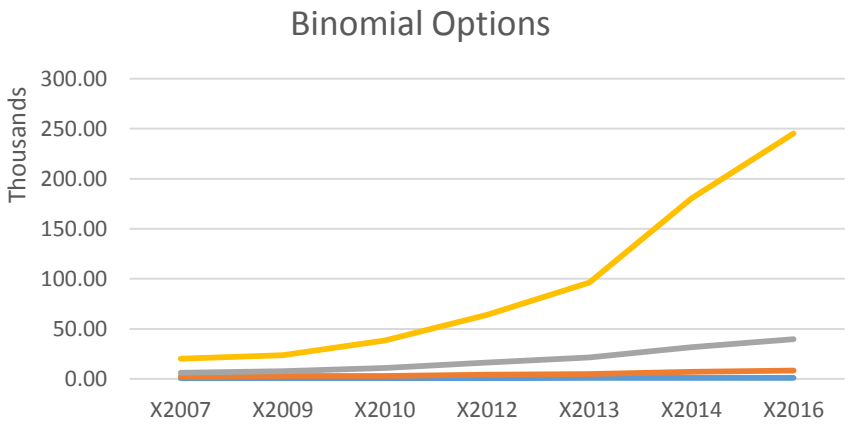
	Cores	SIMD	LANES
X2007	8	128	32
X2009	8	128	32
X2010	12	128	48
X2012	16	256	128
X2013	24	256	192
X2014	36	256	288
X2016	44	256	352



(1) Incremental growth in CPU resources

(2) Improvements in compilers and parallel frameworks

(3) Parallelization techniques



intel®

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.

- A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.
- Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.
- The products described in this document 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.
- Intel product plans in this presentation do not constitute Intel plan of record product roadmaps. Please contact your Intel representative to obtain Intel's current plan of record product roadmaps.
- Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: http://www.intel.com/products/processor_number.
- Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.
- Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm
- IVB, KNC, MIC 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
- Intel, Xeon, Xeon Phi, Cilk , Ultrabook, Sponsors of Tomorrow 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.

Compiler - notes

-xhost compiler switch will compile for KNL.
Compilation on KNL will be faster than on KNC, but will be slower than on Intel® Xeon® Processor based systems. The compilation speed depends largely on the size of applications. For large apps, we recommend cross-compiling. The general recommendation is do in the same manner as on KNC.

Source Codes

Monte Carlo available here.

Black Scholes available here

Binomial Options available here

CUDA versions of the codes available with the CUDA SDK code samples version 7.5
https://developer.nvidia.com/cuda-75-downloads-archive

System configurations

System	System type/name	OS version	Memory	CPU	Cache
Harper town	Intel PCSD Production System	Fedora release 20 (Heisenbug) 3.13.6-200.fc20.x86_64	32GB ~ 16x2GB DDR2 800MHz	4 cores Intel(R) Xeon(R) CPU X5472 @ 3.00GHz	L1: 32K L2: 6144K
Nehalem	Supermicro Production X8DNT+	Fedora release 20 (Heisenbug) 3.13.6-200.fc20.x86_64	48GB ~12 x 4GB DDR3 1333	4 cores Intel(R) Xeon(R) CPU X5570 @ 2.9GHZ	L1 :32K L2 :256K L3 :8192K
WSM	Supermicro Intel SDP	Fedora release 20 (Heisenbug) 3.15.10-200.fc20.x86_64	48GB ~12 x 4GB DDR3 1333	6 cores Intel(R) Xeon(R) CPU X5680 @ 3.33GHz	L1 : 32K L2 : 256K L3 : 12288K
SNB	Dell Power Edge R720	Fedora release 20 (Heisenbug) 3.15.10-200.fc20.x86_64	64GB ~ 8x8GB DDR3 1600 MHz	8 cores Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz	L1: 32K L2:256K L3:20480K
IVB	Intel SDP	Red Hat Enterprise Linux Server release 7.1 (Maipo)	64GB ~ 8x8GB DDR3 1867 MHz	12 cores Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz	L1: 32K L2: 256K L3-30720K
HSW	PCSD production system from Demo Depot	Fedora release 20 (Heisenbug) 3.15.10-200.fc20.x86_64	128GB~ 8x16 GB DDR4 2133MHZ Running at 1067 MHz	18 cores Intel ® Xeon(R) CPU E5-2699 v3 @ 2.30GH z	L1: 32K L2: 256K L3: 46080K
BDW-18 Cores	Intel SDP	Red Hat Enterprise Linux Server release 7.0 (Maipo) 3.10.0-123.el7.x86_64	256GB~ 16 x 16GB DDR4 2400 MHz	18 cores Intel ® Xeon(R) CPU E5-2697 v4 @ 2.30GHz	L1: 32K L2: 256K L3: 46080K
BDW-22 Cores	Intel SDP	CentOS Linux release 7.2.1511 3.10.0-327.el7.x86_64	128GB~8x 16GB DDR4 2133 MHz	22 cores Intel ® Xeon(R) CPU E5-2699 v4 @ 2.20GHz	L1: 32K L2: 256K L3: 56320K