UNLEASH THE POWER OF AVX-512 THROUGH ARCHITECTURE , COMPILER AND CODE MODERNIZATION

Xinmin Tian, Robert Geva, and Bob Valentine
Intel Corporation
September 11, 2016

PACT 2016 Tutorial, Haifa, Israel

Agenda

Section I - AVX-512 Architecture Insights

Section II - Intel® Compiler: Putting SIMD Vectorization to Work

Section III - Code Modernization: Best Practices for Vector Programming

AVX512 ARCHITECTURE INSIGHTS

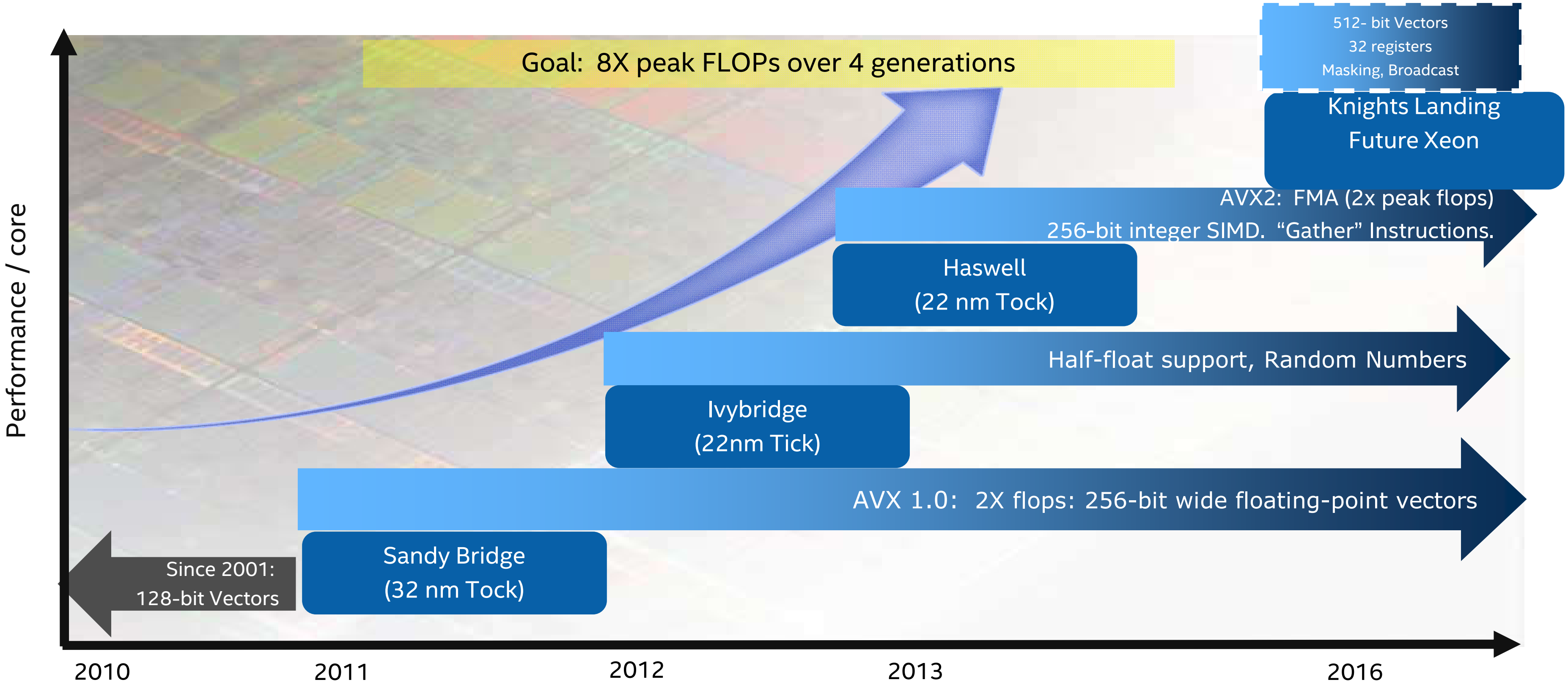
Robert Valentine – Senior Principal Engineer
Intel Corporation
September 11, 2016

PACT 2016 Tutorial, Haifa, Israel

Section I: Agenda

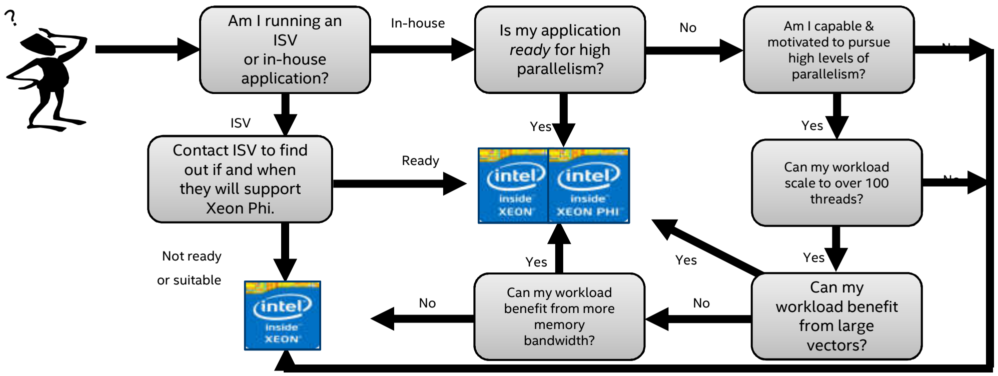
- ✓ Introduction: Intel® ISA Roadmap
- ✓ Deep dive: AVX1/2/AVX512 ISA
- ✓ AVX-512 F: Common ISA Extension
- ✓ AVX-512 ERI & PRI: Intel® Xeon Phi™ Product Only
- ✓ Xeon additions to AVX-512 F
- ✓ Summary

Intel® Advanced Vector Extensions



Optimization Notice

Consistent Developer Tools and Programming Models



Am I running an ISV or in-house application?

In-house

Is my application *ready* for high parallelism?

No

Am I capable & motivated to pursue high levels of parallelism?

No

ISV

Contact ISV to find out if and when they will support Xeon Phi.

Ready

Yes

Yes

Can my workload scale to over 100 threads?

No

Not ready or suitable

Yes

Yes

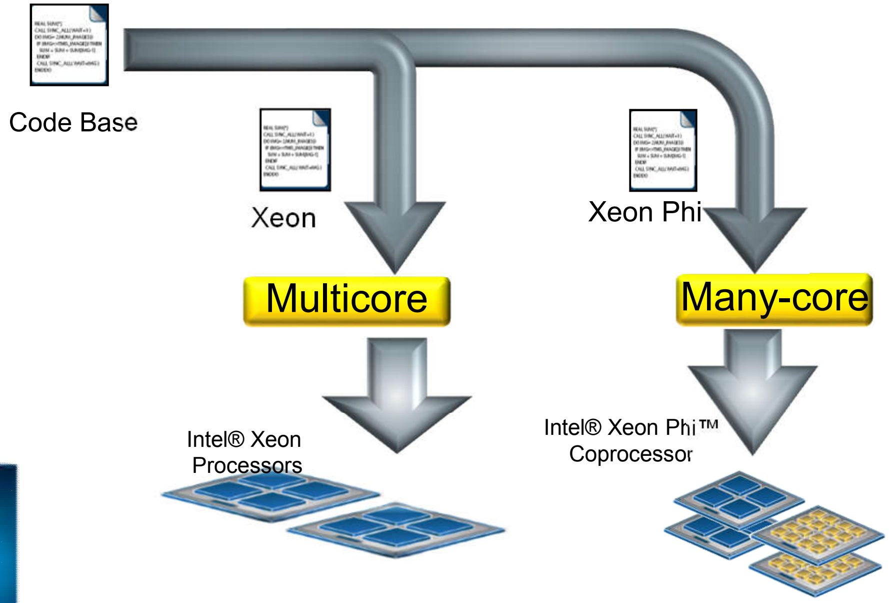
Yes

Can my workload benefit from more memory bandwidth?

No

No

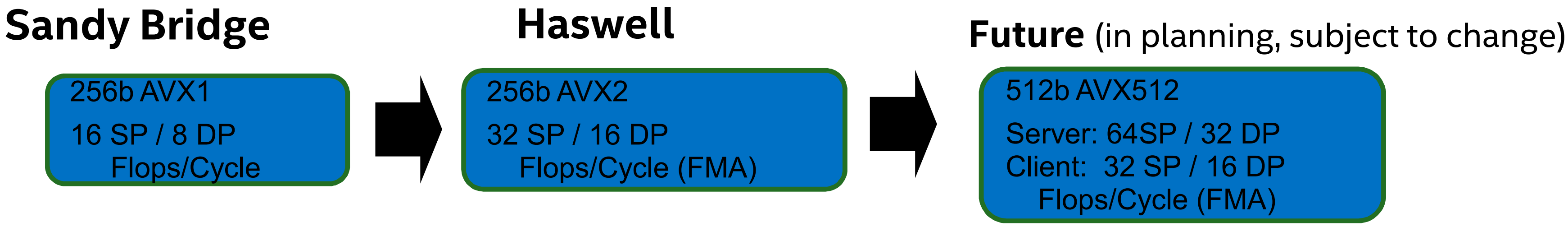
Can my workload benefit from large vectors?



Standards Parallel Programming Models Vectorize, Parallelize, & Optimize

Optimization Notice

Intel® AVX Technology



Sandy Bridge

256b AVX1
16 SP / 8 DP
Flops/Cycle

Haswell

256b AVX2
32 SP / 16 DP
Flops/Cycle (FMA)

Future (in planning, subject to change)

512b AVX512
Server: 64SP / 32 DP
Client: 32 SP / 16 DP
Flops/Cycle (FMA)

AVX	AVX2
256-bit basic FP	Float16 (IVB 2012)
16 registers	256-bit FP FMA
NDS (and AVX128)	256-bit integer
Improved blend	PERMD
MASKMOV	Gather
Implicit unaligned	

AVX512
512-bit FP/Integer
32 registers
8 mask registers
Embedded rounding
Embedded broadcast
Scalar/SSE/AVX “promotions”
Native media additions
HPC additions
Transcendental support
Gather/Scatter

SNB-2011

HSW-2013

Future Processor (Knight Landing & future Xeon)

Optimization Notice

AVX512 Big Picture

- ✓ Deep dive: AVX1/2/AVX512 ISA
- ✓ AVX-512 F: Common ISA Extension
- ✓ AVX-512 ERI & PRI: Intel® Xeon Phi™ Product Only
- ✓ Xeon additions to AVX-512 F

AVX512 big picture

AVX512F

- ‘Foundation’ of architecture, required for any AVX512 implementation
  - Many D/Q/SP/DP promotions from AVX2 with AAVX512 features
    - Masking, 32 registers, embedded broadcast or rounding, 512-bit Vector Length
  - New instructions added to accelerate HPC workloads
- Implementations add features to AVX512F “base”
  - “base” will grow as MIC/Xeon converge on features

	AVX512CD	Conflict Detect : instructions tailored for vectorizing loops with potential address conflicts
	AVX512ER	Exponential and Reciprocal : 'wide' approximateion of Log (22 bits) and RCP/RSQRT (28 bits)
	AVX512PF	Prefetch : Multi-address prefetch instructions using gather/scatter semantics
	AVX512DQ	Additional D/Q/SP/DP instructions (converts, transcendental support, etc)
	AVX512BW	512-bit Byte/Word support (promotions from AVX2, some additions)
	AVX512VL	Vector Length Orthogonality : ability to operate on sub-512 vector sizes

Xeon & Xeon Phi^{TM} New ISA: What Is Where?

Complex & versatile big cores

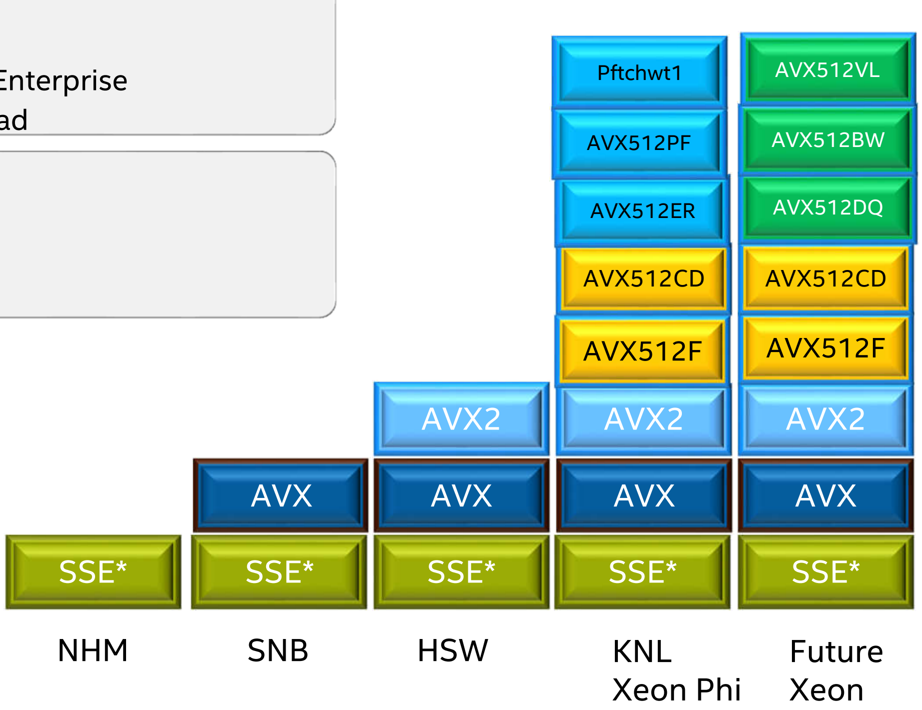
- Big focus on latency and single-thread
- State-of-the-art SIMD support for HPC and Enterprise
- Best balance of performance for any workload

Small & efficient cores

- Big focus on throughput and many-threads
- State-of-the-art SIMD support for HPC
- Industry performance-per-watt leadership

KNL and Xeon are the first "*unification*" platforms:

AVX512F is the common SIMD foundation for HPC software development



NHM	SNB	HSW	KNL Xeon Phi	Future Xeon
			Pftchwt1	AVX512VL
			AVX512PF	AVX512BW
			AVX512ER	AVX512DQ
			AVX512CD	AVX512CD
			AVX512F	AVX512F
		AVX2	AVX2	AVX2
	AVX	AVX	AVX	AVX
SSE*	SSE*	SSE*	SSE*	SSE*

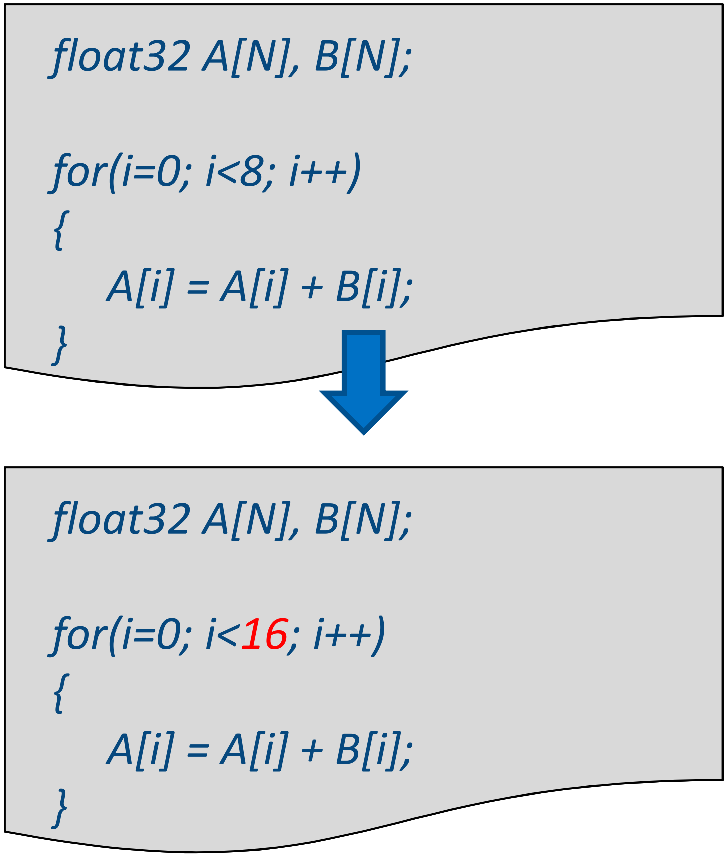
AVX-512 features (I): More & Bigger Registers

AVX: VADDPS YMM0, YMM3, [mem]

- **Up to 16 AVX registers**
  - **8 in 32-bit mode**
- **256-bit width**
  - **8 x FP32**
  - **4 x FP64**

AVX-512: VADDPS ZMM0, ZMM24, [mem]

- **Up to 32 AVX registers**
  - **8 in 32-bit mode**
- **512-bit width**
  - **16 x FP32**
  - **8 x FP64**



```
float32 A[N], B[N];

for(i=0; i<8; i++)
{
    A[i] = A[i] + B[i];
}
```

```
float32 A[N], B[N];

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

But you need many more features to use all that real estate effectively…

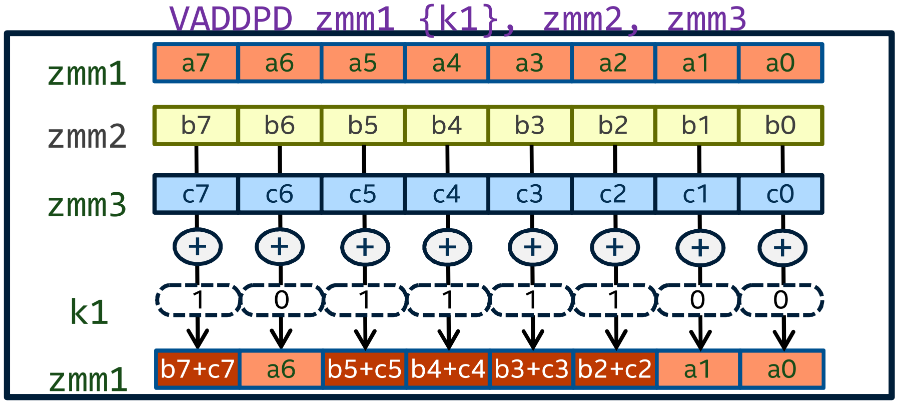
AVX-512 Mask Registers

8 Mask registers of size 64-bits

- k1-k7 can be used for predication
  - k0 can be used as a destination or source for mask manipulation operations

4 different mask granularities.
For instance, at 512b:

- Packed Integer Byte use mask bits [63:0]
  - `VPADDB zmm1 {k1}, zmm2, zmm3`
- Packed Integer Word use mask bits [31:0]
  - `VPADDW zmm1 {k1}, zmm2, zmm3`
- Packed IEEE FP32 and Integer Dword use mask bits [15:0]
  - `VADDPS zmm1 {k1}, zmm2, zmm3`
- Packed IEEE FP64 and Integer Qword use mask bits [7:0]
  - `VADDPD zmm1 {k1}, zmm2, zmm3`


VADDPD zmm1 {k1}, zmm2, zmm3
zmm1 a7 a6 a5 a4 a3 a2 a1 a0
zmm2 b7 b6 b5 b4 b3 b2 b1 b0
zmm3 c7 c6 c5 c4 c3 c2 c1 c0
+ + + + + + + +
k1 1 0 1 1 1 1 0 0
zmm1 b7+c7 a6 b5+c5 b4+c4 b3+c3 b2+c2 a1 a0

		Vector Length		
		128	256	512
element size	Byte	16	32	64
	Word	8	16	32
	Dw ord/SP	4	8	16
	Qw ord/DP	2	4	8

AVX-512 Features (II): Masking

VADDPS ZMM0 {k1}, ZMM3, [mem]

- Mask bits used to:

  1. *Suppress individual elements read from memory*
     - hence not signaling any memory fault
  2. *Avoid actual independent operations within an instruction happening*
     - and hence not signaling any FP fault
  3. *Avoid the individual destination elements being updated,*
     - or alternatively, force them to zero (zeroing)

```
for (I in vector length)
{
   if (no_masking or mask[I]) {
      dest[I] = OP(src2, src3)
   } else {
      if (zeroing_masking)
         dest[I]  = 0
      else
         // dest[I] is preserved
   }
}
```

Caveat: vector shuffles do not suppress memory fault exceptions
mask refers to “output” not to “input”

Why True Masking?

Memory fault suppression

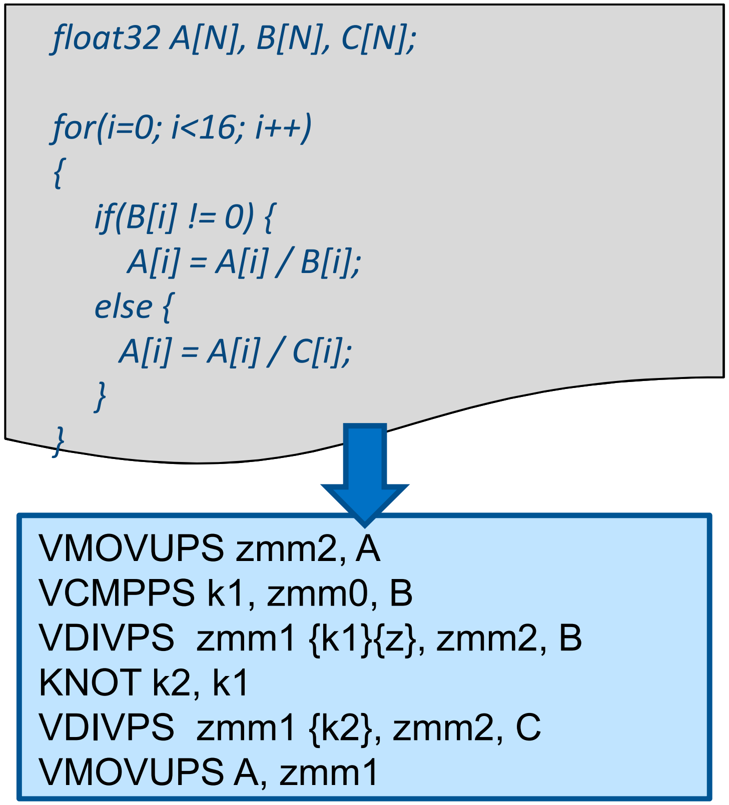
- Vectorize code without touching memory that the correspondent scalar code would not touch
  - Typical examples are if-conditional statements or loop remainders
  - AVX is forced to use VMASKMOV* (risc)

MXCSR flag updates and fault handlers

- Avoid spurious floating-point exceptions without having to inject neutral data

Zeroing/merging

- Use zeroing to avoid false dependencies in OOO architecture
- Use merging to avoid extra blends in if-then-else clauses (predication) for great code density



```
float32 A[N], B[N], C[N];

for(i=0; i<16; i++)
{
    if(B[i] != 0) {
       A[i] = A[i] / B[i];
    else {
      A[i] = A[i] / C[i];
    }
}
```

```
VMOVUPS zmm2, A
VCMPPS k1, zmm0, B
VDIVPS  zmm1 {k1}{z}, zmm2, B
KNOT k2, k1
VDIVPS  zmm1 {k2}, zmm2, C
VMOVUPS A, zmm1
```

Embedded Broadcasts and Masking Support

VFMADD231PS zmm1, zmm2, C {1to16}

- Scalars *from memory* are first class citizens
  - Broadcast one scalar from memory into all vector elements before operation
- Memory fault suppression avoids fetching the scalar if no mask bit is set to 1

```
float32 A[N], B[N], C;

for(i=0; i<8; i++)
{
    if(A[i]!=0.0)
       A[i] = A[i] + C* B[i];
}
```

Other "tuples" supported

- Memory only touched if at least one consumer lane needs the data
- For instance, when broadcast a tuple of 4 elements, the semantics check for every element being really used
  - E.g.: element 1 checks for mask bits 1, 5, 9, 13, …

```
VBROADCASTSS zmm1 {k1}, [rax]
VBROADCASTF64X2 zmm2 {k1}, [rax]
VBROADCASTF32X4 zmm3 {k1}, [rax]
VBROADCASTF32X8 zmm4, {k1}, [rax]
…
```

AVX-512 Features:
Embedded Rounding Control & SAE (Suppress All Exceptions)

- MXCSR.RC can be overridden on a per instruction basis (Embedded Rounding Control)
  - VADDPS ZMM1 {k1}, ZMM2, ZMM3 {rne-sae}
  - VADDSS XMM1 {k1}, XMM2, XMM3 {rrtz-sae}
- "Suspend All Exceptions" (always implied by using Embedded Rounding Control)

  NO MXCSR updates / exception reporting for any element

Restricted to :
FP instructions
512-bit or scalar
Reg-reg operands

Expected usage of this feature

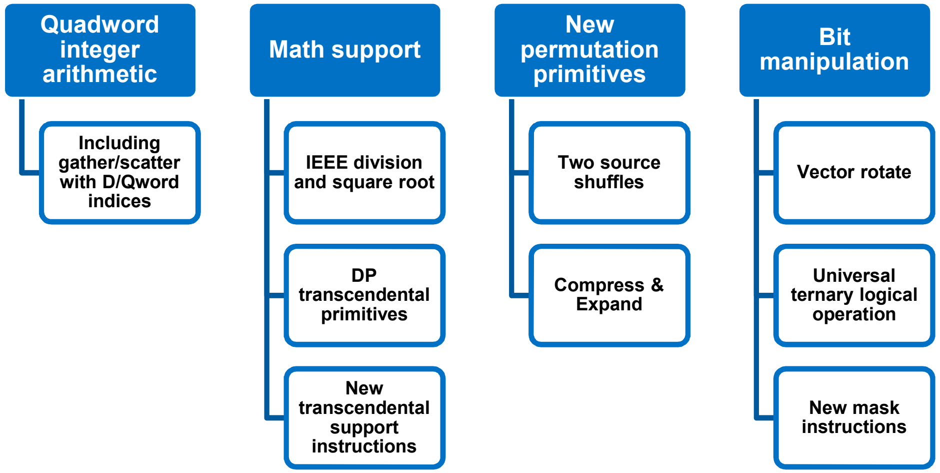
- Library codes can control effect of rounding and updates to MXCSR until the end stages of complex SW routines
  - E.g.: avoid spurious overflow/underflow reporting in intermediate computations
  - E.g: make sure that RM=rne regardless of the contents of MXCSR
- Saving, modifying and restoring MXCSR is generally slower and more and cumbersome
  - Must use LDMXCSR to change fault masks, clear sticky bits or set a default rounding mode
  - Do not need to use MXCSR OR embedded rounding for truncating FP conversion to int (use CVTT* instructions)

AVX-512 F:
Common Xeon Phi (KNL) and Xeon Vector ISA Extension

AVX-512 Foundation is the common SIMD foundation for HPC software development
First on KNL
Planned on a future Xeon

AVX-512 F Designed for HPC

- Promotions of many AVX and AVX2 instructions to AVX-512
    - 32-bit and 64-bit floating-point instructions from AVX
        - Scalar and 512-bit
    - 32-bit and 64-bit integer instructions from AVX2
- Many new instructions to speedup HPC workloads



Quadword integer arithmetic
- Including gather/scatter with D/Qword indices

Math support
- IEEE division and square root
- DP transcendental primitives
- New transcendental support instructions

New permutation primitives
- Two source shuffles
- Compress & Expand

Bit manipulation
- Vector rotate
- Universal ternary logical operation
- New mask instructions

Quadword Integer Arithmetic

Long int and packed pointer manipulation
64-bit integer trending towards becoming a first class citizen
Removes the need for expensive SW emulation sequences

Note: VPMULQ and int64 <-> FP converts not in AVX-512 F

Instruction	Description
`VPADDQ zmm1 {k1}, zmm2, zmm3`	INT64 addition
`VPSUBQ zmm1 {k1}, zmm2, zmm3`	INT64 subtraction
`VP{SRA,SRL,SLL}Q zmm1 {k1}, zmm2, imm8`	INT64 shift (imm8)
`VP{SRA,SRL,SLL}VQ zmm1 {k1}, zmm2, zmm3`	INT64 shift (variable)
`VP{MAX,MIN}Q zmm1 {k1}, zmm2, zmm3`	INT64 max, min
`VP{MAX,MIN}UQ zmm1 {k1}, zmm2, zmm3`	UINT64 max, min
`VPABSQ zmm1 {k1}, zmm2, zmm3`	INT64 absolute value
`VPMUL{DQ,UDQ} zmm1 {k1}, zmm2, zmm3`	32x32 = 64 integer multiply

Math Support

Package to aid with Math library writing

- Good value upside in financial applications
- Available in PS, PD, SS and SD data types
- Great in combination with embedded RC

30

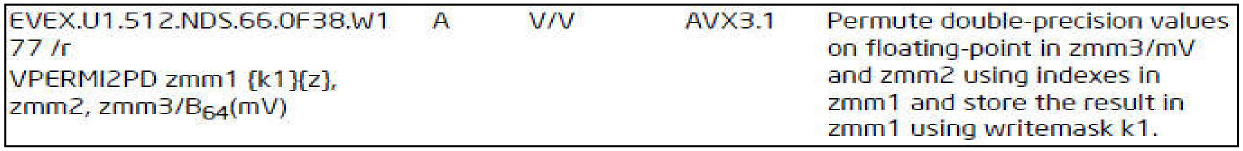
Instruction		
VGETXEXP$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Obtain exponent in FP format
VGETMANT$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Obtain normalized mantissa
VRNDSCALE$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2, imm8	Round to scaled integral number
VSCALEF $_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2, zmm3	$X*2^{y,}$ X <= getmant, Y <= getexp
VFIXUPIMM$_{\{PS,PD,SS,SD\}}$	zmm1, zmm2, zmm3, imm8	Patch output numbers based on inputs
VRCP14$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Approx. reciprocal() with rel. error 2^{-14}
VRSQRT14$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	Approx. rsqrt() with rel. error 2^{-14}
VDIV$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2, zmm3	IEEE division
VSQRT$_{\{PS,PD,SS,SD\}}$	zmm1 {k1}, zmm2	IEEE square root

New 2-Source Shuffles

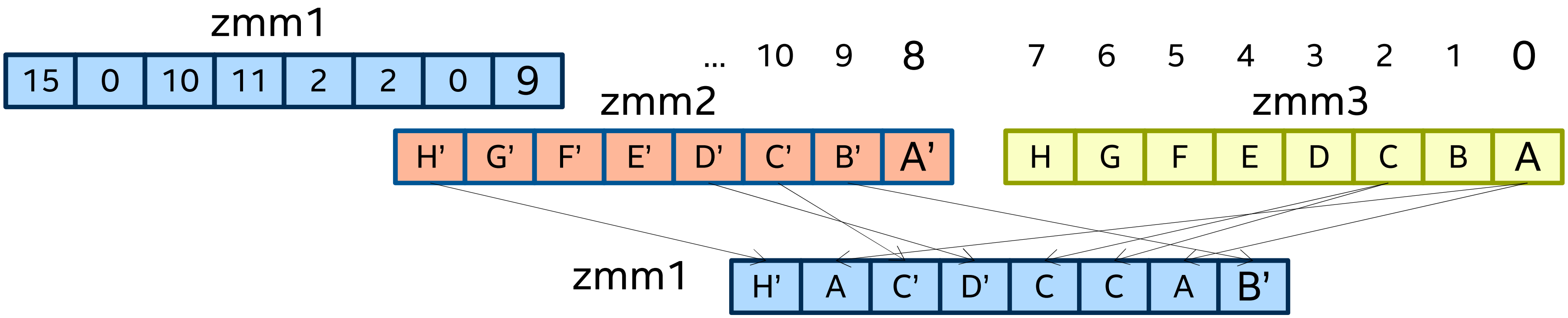
2-Src Shuffles
VSHUF{PS,PD}
VPUNPCK{H,L}{DQ,QDQ}
VUNPCK{H,L}{PS,PD}
VPERM{I,D}2{D,Q,PS,PD}
VSHUF{F,I}32X4

Long standing customer request

- 16/32-entry table lookup (transcendental support)
- AOS ⇔ SOA support, matrix transpose
- Variable VALIGN emulation



EVEX.U1.512.NDS.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/B_{64}(mV)	A	V/V	AVX3.1	Permute double-precision values on floating-point in zmm3/mV and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.



Gather & Scatter

D/Q/SP/DP element types

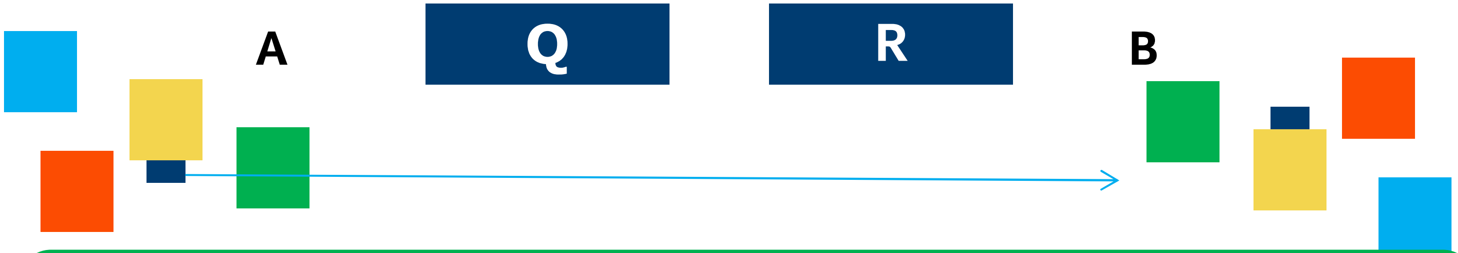
D/Q indices

Instruction can partially execute

- k-reg Mask used as completion mask

```
for(j=0, i=0; i<N; i++)
{
        B[R[i]] = A[Q[i]];
}
```

```
VMOVDQU64 zmm1, Q[rsi]
VMOVDQU64 zmm2, R[rsi]
VGATHERQQ  zmm0 {k2}, [rax+zmm1*8]
VSCATTERQQ [rax+zmm2*8] {k3}, zmm0
```



G/S implementation attempts to ‘max out’ DCU BW

Performance gains come from vectorizing REST of algorithm

Algorithm shown could get some gain (24 load dispatches → 10 per 8 elements)

Expand & Compress

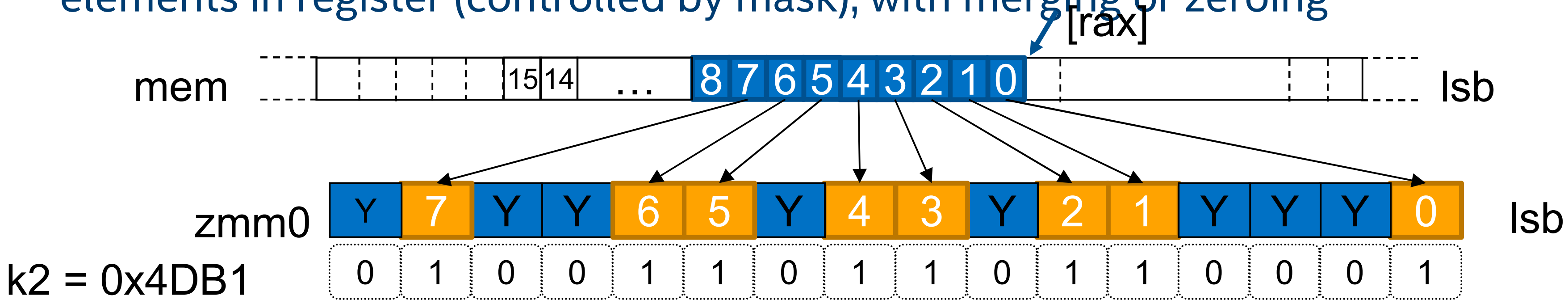
Allows vectorization of conditional loops

- 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(C[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



zmm0	Y	7	Y	Y	6	5	Y	4	3	Y	2	1	Y	Y	Y	0
k2 = 0x4DB1	0	1	0	0	1	1	0	1	1	0	1	1	0	0	0	1

Optimization Notice

Bit Manipulation

Basic bit manipulation operations on mask and vector operands

- Useful to manipulate mask registers
- Have uses in cryptography algorithms

Instruction	Description
`KUNPCKBW k1, k2, k3`	Interleave bytes in k2 and k3
`KSHIFT{L,R}W k1, k2, imm8`	Shift bits left/right using imm8
`VPROR{D,Q} zmm1 {k1}, zmm2, imm8`	Rotate bits right using imm8
`VPROL{D,Q} zmm1 {k1}, zmm2, imm8`	Rotate bits left using imm8
`VPRORV{D,Q} zmm1 {k1}, zmm2, zmm3/mem`	Rotate bits right w/ variable ctrl
`VPROLV{D,Q} zmm1 {k1}, zmm2, zmm3/mem`	Rotate bits left w/ variable ctrl

VPTERNLOG – Ternary Logic Instruction

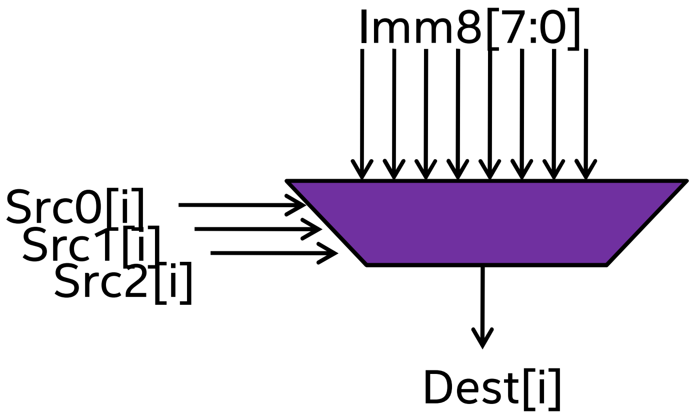
- Take every bit of three sources to obtain a 3-bit index N
  - Obtain Nth bit from imm8

VPTERNLOGD zmm0 {k2}, zmm15, zmm3/[rax], imm8

Any arbitrary truth table of 3 values can be implemented
andor, ***andxor***, ***vote***, ***parity***, bitwise-***cmov***, etc
each column in the right table corresponds to imm8

S1	S2	S3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

ANDOR	VOTE	(S1)?S3:S2
0	0	0
1	0	1
0	0	0
1	1	1
0	0	0
1	1	0
1	1	1
1	1	1



AVX-512 ERI & AVX-512 PRI: Xeon Phi Only

Xeon Phi Only Instructions

Set of segment-specific instruction extensions

- First appear on KNL
- Will be supported in all future Xeon Phi processors
- May or may not show up on a later Xeon processor

Address two HPC customer requests

- Ability to maximize memory bandwidth
  - Hardware prefetching is too restrictive
  - Conventional software prefetching results in instructions overhead
- Competitive support for transcendental sequences
  - Mostly division and square root
  - Differentiating factor in HPC/TPT

KNL AVX512 additions

CPUID	Instructions	Motivation
AVX-512 PRI	PREFETCHWT1	Reduce ring traffic in core-to-core data communication
	VGATHERPF{D,Q}{0,1}PS	Reduce overhead of software prefetching: *dedicate side engine to prefetch sparse structures while devoting the main CPU to pure raw flops*
	VSCATTERPF{D,Q}{0,1}PS	
AVX-512 ERI	VEXP2{PS,PD}	Speed-up key FSI workloads: Black-Scholes, Montecarlo
	VRCP28{PS,PD}	Key building block to speed up most transcendental sequences (in particular, division and square root): Increasing precision from 14=>28 allows to reduce one complete Newton-Raphson iteration
	VRSQRT28{PS,PD}	

Summary of AVX512 on KNL

AVX-512 F: new 512-bit vector ISA extension

- Common between Xeon and Xeon Phi (KNL)

AVX-512 CDI Conflict detection instructions

- Improves autovectorization of Histogram data patterns
- On Xeon Phi first

AVX-512 ERI & PRI

- 28-bit transcendentals and new prefetch instructions
- On Xeon Phi only

Xeon additions to AVX512F

AVX512DQ

Complete Qword support

- VPMULLQ packed 64x64 → 64
- Packed/Scalar converts of signed/unsigned to SP/DP
- Arithmatic shift right
- Etc

Extend mask architecture to word and byte

- Byte masks are natural for packed Qword operands

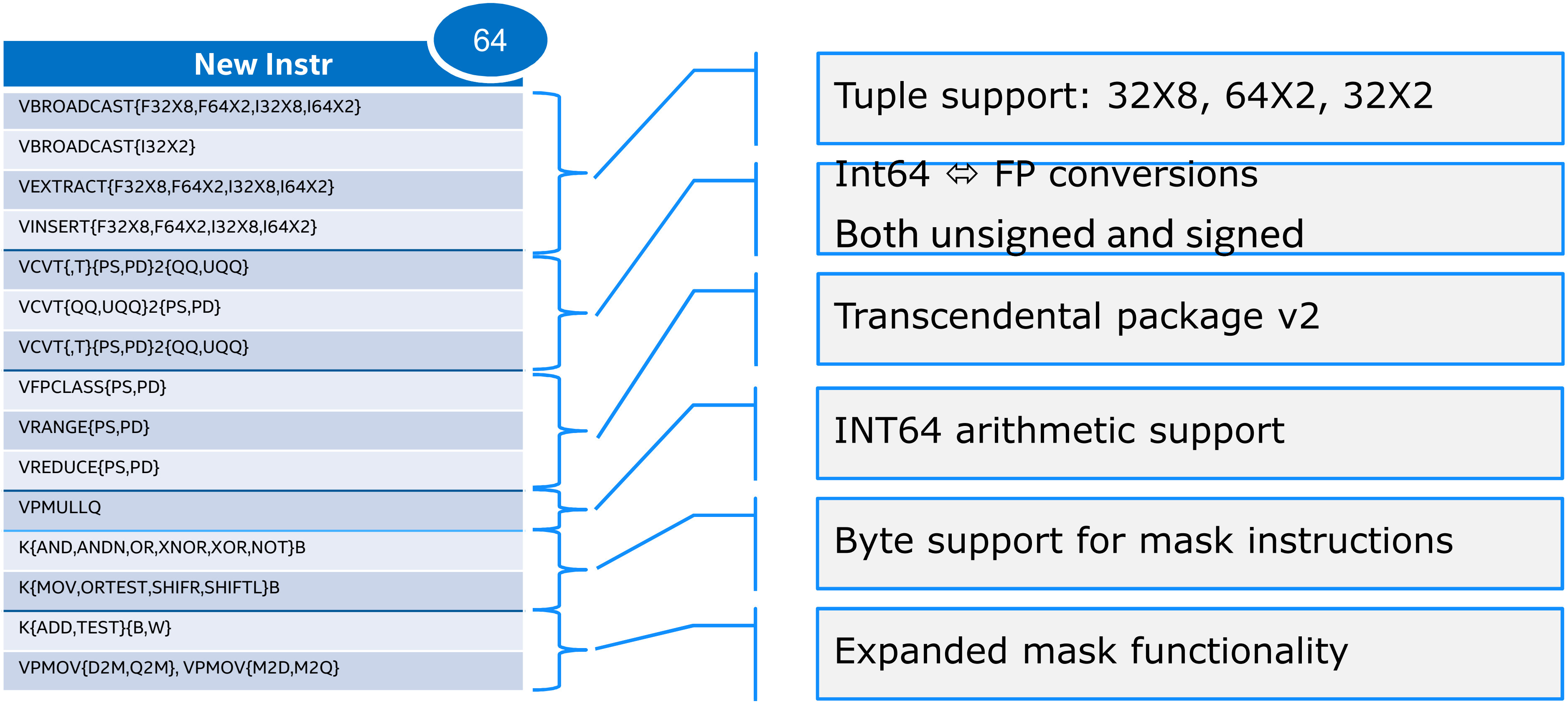
Minor additions to transcendental support

Convert AVX512 mask ←→ 'SSE/AVX' mask

'aggregate datatype' support

- Broadcast/insert/extract complex singles etc

AVX512DQ : additional HPC focus



64

New Instr
VBROADCAST{F32X8,F64X2,I32X8,I64X2}
VBROADCAST{I32X2}
VEXTRACT{F32X8,F64X2,I32X8,I64X2}
VINSERT{F32X8,F64X2,I32X8,I64X2}
VCVT{,T}{PS,PD}2{QQ,UQQ}
VCVT{QQ,UQQ}2{PS,PD}
VCVT{,T}{PS,PD}2{QQ,UQQ}
VFPCLASS{PS,PD}
VRANGE{PS,PD}
VREDUCE{PS,PD}
VPMULLQ
K{AND,ANDN,OR,XNOR,XOR,NOT}B
K{MOV,ORTEST,SHIFR,SHIFTL}B
K{ADD,TEST}{B,W}
VPMOV{D2M,Q2M}, VPMOV{M2D,M2Q}

Tuple support: 32X8, 64X2, 32X2

Int64 ⇔ FP conversions

Both unsigned and signed

Transcendental package v2

INT64 arithmetic support

Byte support for mask instructions

Expanded mask functionality

AVX512BW

Full support for Byte/Word operations

- MMX/SSE2/AVX2 re-promoted to AVX512 semantics

Mask operations extended to 32/64 bits

- 32-bit mask refers to AVX512 ‘short’ operands
- 64-bit mask refers to AVX512 byte operands

Loads/Stores/Broadcastsfor AVX512 semantics

Permute architecture extended to words

- Vpermw, vpermi2w, vpermt2w

New PSAD instruction,etc

AVX512BW : Byte and Word Support

131

AV512BW
VPBROADCAST{B,W}
VPSRLDQ, VPSLLDQ
VP{SRL,SRA,SLL}{V}W
VPMOV{WB,SWB, USWB}
VPTESTM{B,W}
VPMADW
VPTESTNM{B,W}
VDBPSADBW
VPERMW, VPERM{I,T}2W
VMOVDQU{8,16}
VPBLENDM{B,W}
{KAND,KANDN}{D,Q}
{KOR,KXNOR,KXOR}{D,Q}
KNOT{D,Q}
KORTEST{D,Q}

AV512BW
KTEST{D,Q}
KSHIFT{L,R}{D,Q}
KUNPACK{WD,DQ}
KADD{D,Q}
VPMOV{B2M,W2M,M2B,M2W}
VPCMP{,EQ,GT}{B,W,UB,UW}
VP{ABS,AVG}{B,W}
VP{ADD,SUB}{,S,US}{B,W}
VPALIGNR
VP{EXTR,INSR}{B,W}
VPMADD{UBSW,WD}
VP{MAX,MIN}{S,U}{B,W}
VPMOV{SX,ZX}BW
VPMUL{HRS,H,L}W
VPSADBW

AV512BW
VPSHUFB, VPSHUF{H,L}W
VP{SRA,SRL,SLL}{,V}{W}
VPUNPCK{H,L}{BW,WD}

Optimization Notice

AVX512VL : Vector Length Orthogonality

Allow AVX512 instructions to operate on sub-vectors (lower 256/128 bits)

- Eases code generation for mixed data types
  - Partial masks are functionally correct, why not use them?
    - VL is in static in opcode, provides information EARLY in pipeline
      - Clock gating of unneeded execution elements / buses
      - Disabling RF read ports
      - Preventing ‘false overlap/forwarding’ from being detected in memory
    - Creating partial masks wastes instruction BW

AVX512VL is NOT a “list of instructions”

- “orthogonal feature’ applying to “all” AVX512 instructions
  - obvious caveats when instruction has implicit 256/512 width

AVX512VL : Down-promotions

318 Out of 450 AVX512 Instructions

VL orthogonality		
V{ADD,MUL,SUB}{PS,PD}	VF{N}MADD{132,213,231}{PS,PD}	VPERMIL{PS,PD}, VSHUF{PS,PD}
VALIGN{D,Q}	VF{N}MSUB{132,213,231}{PS,PD	VP{MAX,MIN}{D,Q,UD,UQ}
VBLENDM{PS,PD}, VPBLENDM{D,Q}	VFMADDSUB{132,213,231}{PS,PD}	VPMOX{SX,ZX}{B,W}{D,Q}
VBROADCAST{SS,SD,F32X4,I32X4}	VFMSUBADD{132,213,231}{PS,PD}	VPMOX{SX,ZX}DQ
VCMP{SS,SD}	VGATHER{D,Q}{PS,PD}	VPMUL{DQ,UDQ,LD}
VCOMPRESS{PS,PD}, VPCOMPRESS{D,Q}	VPGATHER{D,Q}{D,Q}	VP{SLL,SRL,SRA}{,V}{D,Q}
VCVT{DQ,UDQ}2{PS,PD}	V{MAX,MIN}{PS,PD}	VPTESTM{D,Q}
VCVT{,T}{PS,PD}2{DQ,UDQ}	VMOV{APS,UPS,DQA32,DQA64}	VPUNPCK{H,L}{DQ,QDQ}
VCVT{PS2PD,PD2PS}	[illegible]	[illegible]SQRT}14{PS,PD}
VCVT{PS2PH,[illegible]	[illegible]	[illegible]
VDIV{PS,PD}	VP{ABS,ADD,SUB}{D,Q}	VPTERNLOG{D,Q}
VEXPAND{PS,PD}, VPEXPAND{D,Q}	VP{AND,ANDN,OR,XOR}{D,Q}	VPMOVQ{,S,US}Q{QB,QW,QD,DB,DW}
VEXTRACT{F32X4,I32X4}	VPCMP{,EG,GR}{D,Q,UD,UQ}	VSHUF{F32X4,F64X2,I32X4,I64X2}
V{MAX,MIN}{PS,PD}	VPERM{D,Q,PS,PD}	VPERM{T,I}2{D,Q,PS,PD}

Etc probably more than are shown

Optimization Notice

Summary of Xeon AVX512 Additions

More Qword support

- Packed converts, VPMULLQ etc

Support for mixing AVX and AVX512 style masks

- VPMOVM2*, VPMOV*2M

All HLL datatypes at maximum SIMD width

- # elements = VL / element_size

VL aids mixing datatypes

- VL = # elements * element_size

VL specifies memory access sizes exactly

- Masks provide architectural support, but HW prefers a ‘static’ knowledge

Summary

AVX512 is a comprehensive addition to intels SIMD Instruction set

- ~2x performance on BLAS routines
- new features to increase the vectorization coverage (masks, VPCOMRESS)
- embedded rounding and new instructions accelerate math libraries

Knights Landing emphasis on HPC

- support for D/Q/SP/DP and additional specialized instructions

Xeon adds support for all HLL datatypes

AVX512 is designed for compilers as well as programmers

PUTTING EXPLICIT VECTOR PROGRAMMING TO WORK FOR INTEL XEON AND XEON PHI ARCHITECTURES

Xinmin Tian – Senior Principal Engineer
Intel Compiler and Languages, SSG, Intel Corporation
September 11, 2016

PACT 2016 Tutorial, Haifa, Israel

Section II: Agenda

Seamless Vectorization and Parallelization Integration Showcase

Learnings: Cray*, Intel® Pentium® 4 (90nm) SSE3 and SIMD Vectorization Hurdles

Successes: Putting SIMD Vectorization to Work

- ✓ **Mixed data type** Vectorization
- ✓ **Function** vectorization
- ✓ **Outer loop** vectorization
- ✓ SIMD Loop Vectorization with **Cross-iteration Dependency**
- ✓ **Less-than-full-vector** Vectorization
- ✓ **Predication and Masking**
- ✓ **Gather/Scatter** Optimization

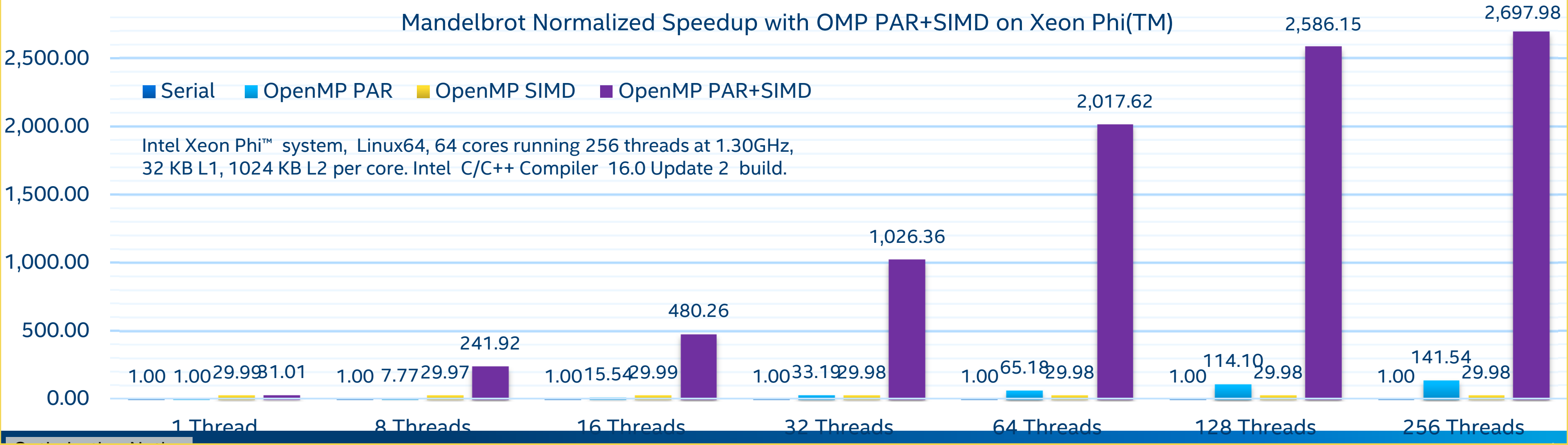
Advances: Tackle C++ Challenges and Beyond C/C++/Fortran

Summary: Close to Metal Performance

Mandelbrot: ~2698x Speedup on Xeon Phi™--Isn’t it Cool?

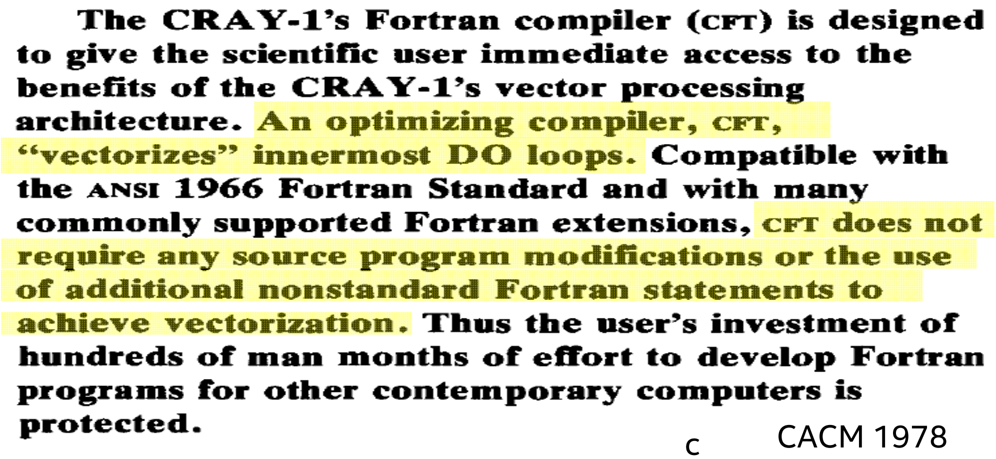
```
#pragma omp declare simd uniform(max_iter), simdlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{   uint32_t count = 1; fcomplex z = c;
    while ((cabsf(z) < 2.0f) && (count < max_iter)) {
       z = z * z + c; count++;
    }
    return count;
}
```

```
#pragma omp parallel for schedule(guided)
for (int32_t y = 0; y < ImageHeight; ++y) {
  float c_im = max_imag - y * imag_factor;
  #pragma omp simd simdlen(32)
  for (int32_t x = 0; x < ImageWidth; ++x) {
    fcomplex in_vals_tmp = (min_real + x * real_factor) + (c_im * 1.0iF);
    count[y][x] = mandel(in_vals_tmp, max_iter);
  }
}
```



Intel Xeon Phi™ system, Linux64, 64 cores running 256 threads at 1.30GHz,
32 KB L1, 1024 KB L2 per core. Intel C/C++ Compiler 16.0 Update 2 build.

Learnings: Compiler Vectorization in 1978



The CRAY-1's Fortran compiler (CFT) is designed to give the scientific user immediate access to the benefits of the CRAY-1's vector processing architecture. An optimizing compiler, CFT, "vectorizes" innermost DO loops. Compatible with the ANSI 1966 Fortran Standard and with many commonly supported Fortran extensions, CFT does not require any source program modifications or the use of additional nonstandard Fortran statements to achieve vectorization. Thus the user's investment of hundreds of man months of effort to develop Fortran programs for other contemporary computers is protected.

Livermore loop #1
Small loop, simple data and control flow

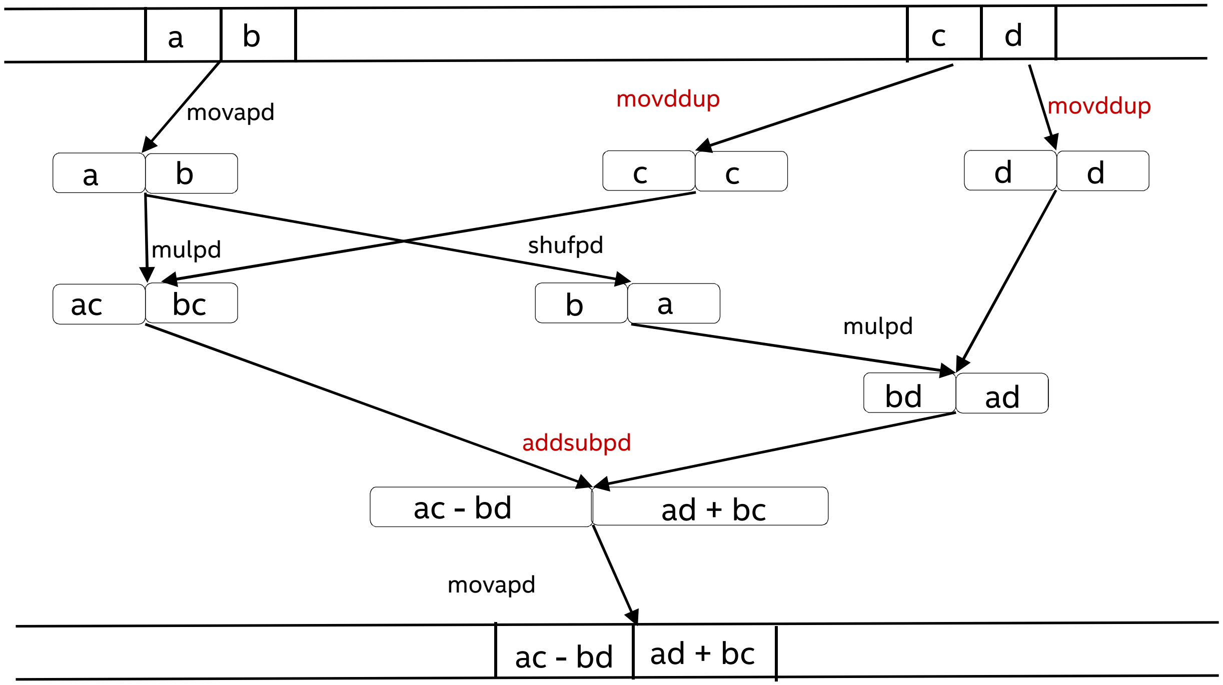
Compiler auto-vectorization becomes reality through dependency analysis

```
c        CACM 1978
c************************************************************************
c***  KERNEL 1      HYDRO FRAGMENT
c************************************************************************
c
cdir$ ivdep
1001    DO 1 k = 1,n
   1       X(k)= Q + Y(k) * (R * ZX(k+10) + T * ZX(k+11))
c
```

Compiler vectorization "solved" in 1978

Learnings: 2004 Intel® Pentium® 4 SSE3 on 90nm

Complex Multiplication with SSE3: $(a + ib)(c + id) = (ac - bd) + i(ad + bc)$



Memory

Ops:

- 3 SIMD loads
- 1 SIMD store
- 3 Arithmetic Ops
- 1 shuffle Ops

Ops not available in SSE2

- movddup
- addsubpd

Memory

4 Scalar loads, 6 Arithmetic Ops, 2 Scalar stores

Performance can be improved up to ~75%, SPEC2000FP/168.wupwise 10-15%

Optimization Notice

Learnings: Program Factors Impact on Vectorization

Loop-carried dependencies

```
DO I = 2, N
   A(I) = A(I-1) + B(I)
ENDDO
```

Function calls

```
for (i = 1; i < nx; i++) {
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp);
}
```

Pointer aliasing

```
void scale(int *a, int *b)
{
   for (int i = 0; i < 1000; i++)
       b[i] = z * a[i];
}
```

many ......

Unknown loop iteration count

```
struct _x { int d; int bound; };
void doit(int *a, struct _x *x)
{
   for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

Indirect memory access

```
 for (i=0; i<N; i++)
     A[B[i]] += C[i]*D[i]
```

Outer loops

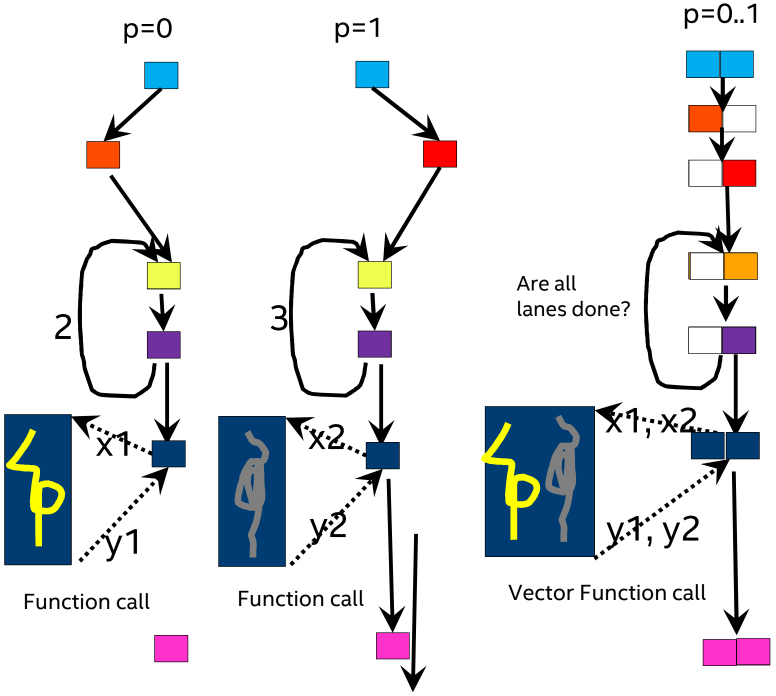
```
for(j = 0; j <= MAX; j++) {
  for(i = 0; i <= MAX; i++) {
    D[i][j] += 1;
  }
}
```

Learnings: SIMD Vectorization Hurdles

```
#pragma omp simd reduction(+:....)
for(p=0; p<N; p++) {
  // Blue work
  if(...) {
     // Green work
  } else {
     // Red work
  }
  while(...) {
     // Gold work
     // Purple work
  }
  y = foo (x);
  Pink work
}
```

Two fundamental problems:

- ✓ Data divergence
- ✓ Control divergence



Vector code generation has become a more difficult problem increasing need for user guided explicit vectorization that maps concurrent execution to simd hardware

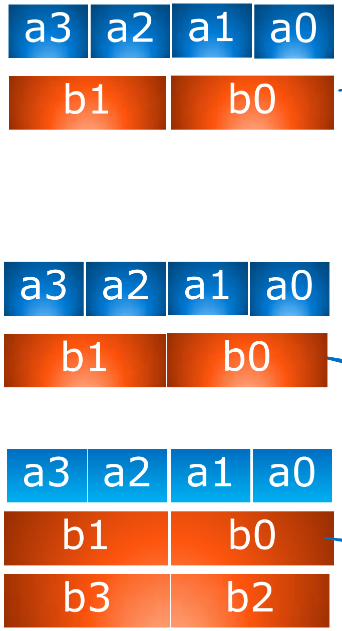
Optimization Notice

Successes: Putting SIMD Vectorization to Work

Intel brings ICC Vectorization Technology to LLVM Vectorizer

Mixed Data Type Vectorization

```
void foo(int n, float *A, double *B) {
  int i;
  float t = 0.0f;
#pragma omp simd
  for (i=0; i<n; i++) {
    A[i] = t;
    B[i] = t;
    t += 1.0f;
  }
}
```



Naïve: use full vectors
4 != 2. Give up (bad)

Match the number of elements.

- ✓ **A[i] =** ... for 2 or 4 elements at a time
- ✓ **B[i] =** ... for 2 or 4 elements at a time

mixed.c(5) (col. 3): remark: LOOP WAS VECTORIZED.

Match: 2=2. Good

Match: 4=2x2. Good

Optimization Notice

Function Vectorization

```
#pragma omp declare simd
float sfoo(float x)
{ … …
}
```

Scalar C function

Compiler created

```
__m128 vfoo(__m128 vx)
{….
}
```

Vector C function

sfoo(x0)->r0
sfoo(x1)->r1
sfoo(x2)->r2
sfoo(x3)->r3
sfoo(x4)->r4
… …

Scalar execution

sfoo(x0)->r0	sfoo(x1)->r1	sfoo(x2)->r2	sfoo(x3)->r3
sfoo(x4)->r4	sfoo(x5)->r5	sfoo(x6)->r6	sfoo(x7)->r7
sfoo(x8)->r8 … …	sfoo(x9)->r9	… …	.. …

vfoo(x0…x3)->r0…r3
vfoo(X4…X7)->r4…r7
… …

Vector execution

Recursive Function Vectorization

```
#pragma omp declare simd [processor(cpu-id)]
int binsearch(int key, int lo, int hi) {
    int ans;
    if ( lo > hi) {
       ans = -1;
    }
    else {
       int mid = lo + ((hi - lo) >> 1);
       int t = sortedarr[mid];
       if (key == t) {
           ans = mid;
       }
       else if ( key > t) {
           ans = binsearch(key, mid + 1, hi);
       }
       else {
           ans = binsearch(key, lo, mid - 1);
       }
    }
    return ans;
}

#pragma omp simd
for (int i=0; i<M; i++) {
   ans[i] = binsearch(keys[i], 0, N-1);
}
```

OpenMP* SIMD PROCESSOR Clause

New **`PROCESSOR`** clause extension to **`#pragma omp declare simd`** (to define a SIMD routine) to target a specific processor

- Similar to Intel® Cilk™ Plus extensions for declaring SIMD functions
- Available for C/C++ and Fortran
- Intel extension – NOT part of official OpenMP specification
- Helpful to allow programmers to leverage e.g. Intel® AVX-2 and Intel® AVX-512 beyond default Intel® SSE2 support (YMM+ZMM registers/operands additionally to XMM)

Processor Name Identifiers

- ✓ pentium_4
- ✓ pentium_m
- ✓ pentium_4_sse3
- ✓ core_2_duo_ssse3
- ✓ core_2_duo_sse4_1
- ✓ atom
- ✓ core_i7_sse4_2
- ✓ core_aes_pclmulqdq
- ✓ core_2nd_gen_avx
- ✓ core_3rd_gen_avx

✓	future_cpu_18	// KNF
✓	mic	
✓	future_cpu_19	// KNC
✓	future_cpu_20	// HSW - no TSX
✓	core_4th_gen_avx	// HSW – no TSX
✓	core_4th_gen_avx_tsx	// HSW - TSX
✓	future_cpu_21	// BDW - NO TSX
✓	future_cpu_21_tsx	// BDW - TSX
✓	future_cpu_22	// KNL
✓	future_cpu_23	// SKL

Vortex Code: Outer Loop Vectorization

```
#pragma omp simd   // simd pragma for outer-loop at call-site of SIMD-function
for (int i = beg*16; i < end*16; ++i) {
    particleVelocity_block(px[i], py[i], pz[i], destvx + i, destvy + i, destvz + i, vel_block_start, vel_block_end);
 }

#pragama omp declare simd linear(velx,vely,velz) uniform(start,end) aligned(velx:64, vely:64, velz:64)
static void particleVelocity_block(const float posx, const float posy, const float posz,
                                   float *velx,  float *vely,  float *velz,  int start, int end) {
  for (int j = start; j < end; ++j) {
    const float del_p_x  = posx - px[j];
    const float del_p_y  = posy - py[j];
    const float del_p_z  = posz - pz[j];
    const float dxn= del_p_x * del_p_x + del_p_y * del_p_y + del_p_z * del_p_z +pa[j]* pa[j];
    const float dxctaui  = del_p_y * tz[j] - ty[j] * del_p_z;
    const float dyctaui  = del_p_z * tx[j] - tz[j] * del_p_x;
    const float dzctaui  = del_p_x * ty[j] - tx[j] * del_p_y;
    const float dst      = 1.0f/std::sqrt(dxn);
    const float dst3     = dst*dst*dst;
    *velx           -= dxctaui * dst3;
    *vely           -= dyctaui * dst3;
    *velz           -= dzctaui * dst3;
  }
}
```

KNC performance improvement
over 2X going
from inner to outer-loop vectorization

Optimization Notice

SIMD Loops with Cross-Iteration Dependencies

OpenMP* 4.5: Extend ordered Blocks in SIMD Contexts

C and C++:

```
#pragma omp ordered [simd]
  structured code block
```

Fortran:

```
!$omp ordered [simd]
        structured code block
!$omp end ordered
```

Semantics:

- *The ordered with simd clause construct specifies a structured block in the simd loop or SIMD function that will be executed in the order of the loop iterations w.r.t to dependency constraints or sequence of call to SIMD functions.*

Rules:

- *#pragma omp ordered simd is only allowed inside a SIMD loop or SIMD-enabled function.*
- *#pragma omp ordered simd region must be a single-entry and single-exit code block*
- *The strict ordered execution is only guaranteed for the block itself*
  - ✓ Execution remains weakly ordered w.r.t. to outside of the block or other ordered blocks
    - ✓ Data dependencies between statements of the same block will be correctly resolved
    - ✓ Other non-vector dependencies originating in ordered block still lead to undefined behavior

Ordered SIMD Examples

OK:

```
#pragma omp simd
for (i = 0; i < N; i++)
{
 ...
  #pragma omp ordered simd
  {
    a[indices[i]] += b[i]; // index conflict
  }
  ...
  #pragma omp ordered simd
  {
    if (c[i] > 0)
      q[j++] = b[i];       // compress pattern
  }
  ...
  #pragma omp ordered simd
  {
     lock(L)               // atomic update
       if (x > 10) x = 0;
     unlock(L)
  }
  ...
}
```

Not OK w.r.t serial:

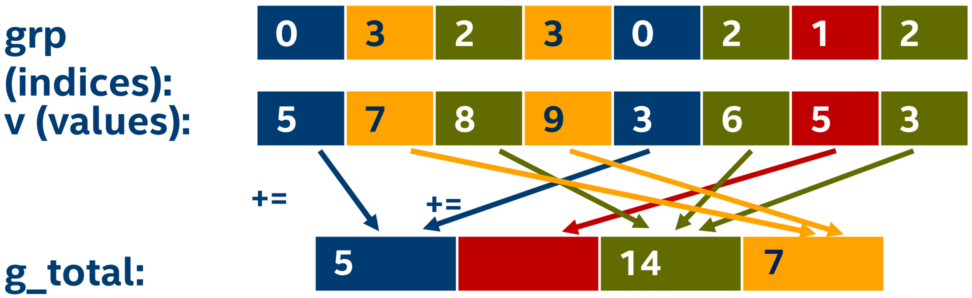
```
#pragma omp simd
for (i = 0; i < N; i++) {
...
#pragma omp ordered simd
  {
    if (c[i] > 0)
      q[j++] = b[i];    // 1st compress
  }
  ...
#pragma omp ordered simd
  {                     // 2nd compress
    if (c[i] > 0)       // Order of stores will
      q[j++] = d[i];    // be changed w.r.t
                        // to serial execution
  }
}
```

Compiler won't complain!

ORDERED SIMD not always best Approach

```
#pragma omp simd
for(int i=0; i < VL; i++) {
   …
    val = values[i];
    grp = groups[i];
#pragma omp ordered simd // conflict
    { g_total[grp] += val; }
    …
}
```

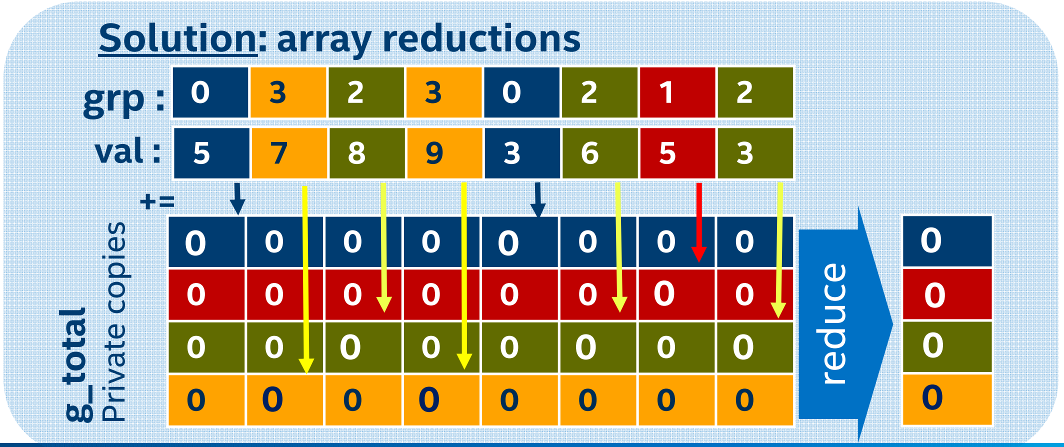
```
 #pragma omp simd reduction(+:g_total)
 for(int i=0; i < VL; i++) {
     …
     val = values[i];
     grp = groups[i];
     g_total[grp] += val;
     …
 }
```



grp (indices):	0	3	2	3	0	2	1	2
v (values):	5	7	8	9	3	6	5	3

g_total:	5		14	7

Solution: array reductions



grp :	0	3	2	3	0	2	1	2
val :	5	7	8	9	3	6	5	3

0	0	0	0	0	0	0	0		0
0	0	0	0	0	0	0	0		0
0	0	0	0	0	0	0	0		0
0	0	0	0	0	0	0	0		0

Optimization Notice

Less-Than-Full-Vector Vectorization

```
float foo(float *y, int n)
{  int k; float x = 10.0f;
  #pragma omp simd
  for (k = 0; k < n; k++) {
    x = x + fsqrt(y[k])
  }
  return x
}
```

```
misalign = &y[0] & 63
peeledTripCount = (63 – misalign)/sizeof(float)
x = 10.0f;
do k0 = 0, peeledTripCount-1  // peeling loop
  x = x + fsqrt(y[k0])
enddo
x1_v512 = (m512)0
x2_v512 = (m512)0
mainTripCount = n – ((n – peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
     x1_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1:16]),x1_v512)
     x2_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo
// perform vector add on two vector x1_v512 and x2_v512
x1_v512 = _mm512_add_ps(x1_v512, x2_512);

// perform horizontal add on all elements of x1_v512, and
// the add x for using its value in the remainder loop
x = x + _mm512_hadd_ps(x1_512)
do k2 = mainTripCount, n  // Remainder loop
  x = x + fsqrt(y[k2])
enddo
```

Less-Than-Full-Vector Vectorization

```
misalign = &y[0] & 63
peeledTripCount = (63 – misalign) / sizeof(float)
x = 10.0f;
// create a vector: <0,1,2,…15>
k0_v512 = _mm512_series_pi(0, 1, 16)

// create vector: all 16 elements are peeledTripCount
peeledTripCount_v512 =
                  _mm512_broadcast_pi32(peeledTripCount)
x1_v512 = (m512)0
x2_v512 = (m512)0
do k0 = 0, peeledTripCount-1, 16
  // generate mask for vectorizing peeling loop
  mask = _mm512_compare_pi32_mask_lt(k0_v512,
                                 peeledTriPCount_v512)
  x1_v512 = _mm512_add_ps_mask(
                _mm512_fsqrt(y[k0:16]), x1_v512, mask)
enddo
```

```
mainTripcount = n – ((n – peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
  x1_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1:16]),    x1_v512)
  x2_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo

// create a vector: <mainTripCount, mainTripCount+1 … mainTripCount+15>
k2_v512   = _mm512_series_pi(mainTripCount, 1, 16)

// create a vector: all 16 elements has the same value n
n_v512    = _mm512_broadcast_pi32(n)
step_v512 = _mm512_broadcast_pi32(16)

do k2 = mainTripCount, n, 16          // vectorized remainder loop
   mask    = _mm512_compare_pi32_mask_lt(k2_v512, n_v512)
   x1_v512 = _mm512_add_ps_mask(
                          _mm512_fsqrt(y[k2:16]), x1_v512, mask)
   k2_v512 = _mm512_add_ps(k2_v512, step_v512)
enddo

x1_v512 = _mm512_add_ps(x1_v512, x2_512);

 // perform horizontal add on 8 elements and final reduction sum to write
//  the result back to x.
x = x + _mm512_hadd_ps(x1_512)
```

Optimization Notice

Predication and Masking

No conditional execution inside SIMD vector loop

- ✓ Except special vector conditions
- ✓ Conditions transformed to vector predicates
- ✓ Control flow is flattened

Predicated execution

- ✓ Masking
- ✓ Blending with speculation

Masking on loads/stores

- ✓ Available on AVX512
- ✓ Limited availability in Intel® MIC Arch, AVX2, and AVX
- ✓ Emulated on SSE

Blending

- ✓ Safety to be proved
- ✓ Only reverse conditions

Before vectorizer

```
#pragma omp simd
for (i=0; i<n; i++) {
   if (A[i]>0) {
      B[i] = C[i] + 20;
   }
   else {
      B[i] = C[i] + 100;
   }
}
```

Flattened masked

```
M = (A[i:i+3] > 0);
B[i:i+3]{M}= C[i:i+3]{M}+{M}20;
B[i:i+3]{~M}= C[i:i+3]{~M}+{~M}100;
```

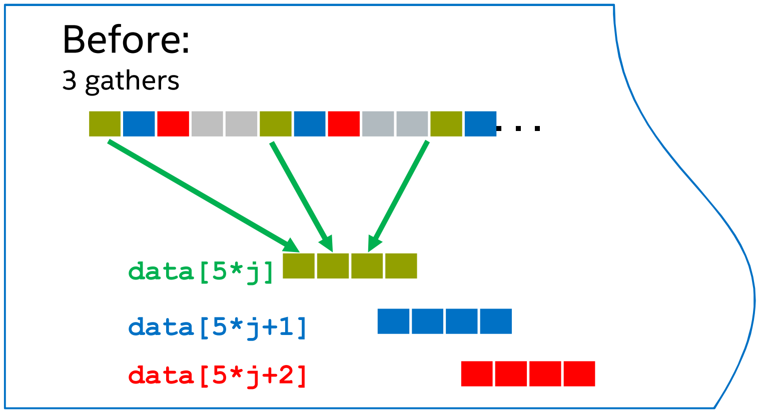
Blended if you are lucky

```
M = (A[i:i+3] > 0);
B[i:i+3] = (C[i:i+3]+20)&M | (C[i:i+3]+100)&~M;
```

Avoid Branchy Code → Improve SIMD Vector Efficiency

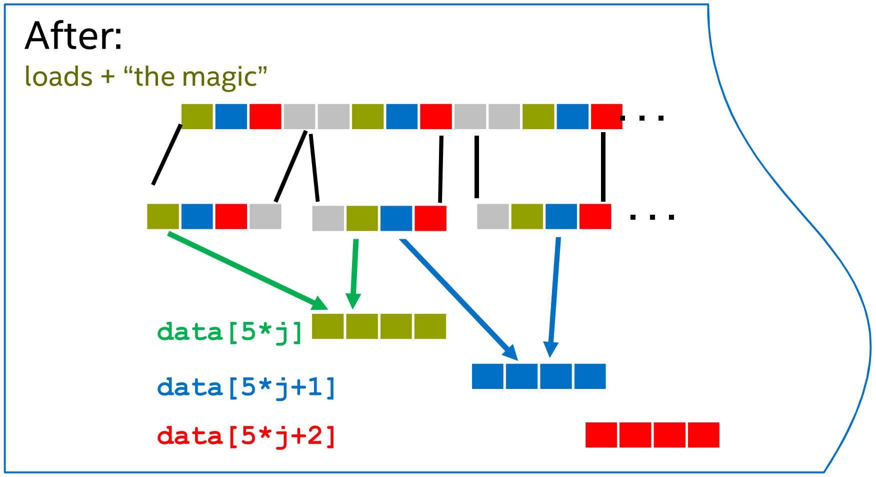
Neighboring Gather/Scatter Optimization

```
for (int i = 0; i < size; ++i) {
  #pragma omp simd
  for (int j = i + 1; j < size; ++j) {
    pij = pi - data[5 * j];
    qij = qi - data[5 * j + 1];
    rij = ri - data[5 * j + 2];
    ...
```



Before:

3 gathers



After:

loads + "the magic"

2x kernel speed-up on HSW

Optimization of neighboring Gathers

- Complete support for unmasked strided (d[i]) and indexed (d[ind[i]]) loads of 1, 2, 4, 8 and 16-byte elements for SSE2-AVX512
- Provides more effective CPU resources usage **for cases with data locality**
- May require additional source changes to enable the pattern recognition (e.g. restrict, base/index hoisting, loads grouping)
- Also reflected in optimization report:

```
remark #34030: adjacent sparse (strided) loads
optimized for speed.
Details: stride { 12 }, types { F32-V512, F32-
V512, F32-V512 }, number of elements { 16 },
select mask { 0x000000007 }.
```

```
struct {
    TYPE f0;
    …
    TYPE fN;
} d[];

for (int i = 0; i < size; ++i)
    tmp += d[i].f0 + … + d[i].fN;
```

Before (TYPE=FLOAT, N=2)

```
vgatherdps  0(%rcx,%zmm0,4), %zmm6{%k1}
vgatherdps  4(%rcx,%zmm0,4), %zmm7{%k2}
vgatherdps  8(%rcx,%zmm0,4), %zmm9{%k3}
```

Now (TYPE=FLOAT, N=2)

```
vmovups     (%rcx), %zmm10
vmovups     64(%rcx), %zmm9
vmovups     128(%rcx), %zmm14
vpermi2ps   %zmm9, %zmm10, %zmm7
vpermi2ps   %zmm9, %zmm10, %zmm8
vpermt2ps   %zmm9, %zmm1, %zmm10
vpermi2ps   %zmm7, %zmm14, %zmm11
vpermi2ps   %zmm8, %zmm14, %zmm12
vpermt2ps   %zmm10, %zmm0, %zmm14
```

OpenMP SIMD Linear(ref/val/uval)

Rationale:

- For implicitly reference linear parameters it is nice to have reference as linear

OpenMP 4.5 syntax:

- Linearity specification for references vs. values
- linear(val(var):[step]) – the value is linear even if passed by reference
  - If passed by reference the vector of references is passed
- linear(uval(var):[step]) – value passed by reference is linear
  - The reference to the first lane is passed, other values constructed using step
- linear(ref(var):step) – for parameters passed by reference the underlying reference is linear
  - Access will be sequential or strided depending on step
- Original linear(var:[step]) – the same as linear(val(var):[step])

```
!$omp declare simd
REAL FUNCTION FOO(X, Y)
REAL, VALUE :: Y   << by reference
REAL, VALUE :: X   << by reference
FOO = X + Y      << gathers!!!!
END FUNCTION FOO
…
!omp$ simd private(X,Y)
DO I= 0, N
    Y = B(I)
    X = A(I)
    C(I) += FOO(X, Y)
ENDDO
```

```
!$omp declare simd linear(ref(x),ref(y))
REAL FUNCTION FOO(X, Y)
REAL, VALUE :: Y    << by reference
REAL, VALUE :: X    << by reference
FOO = X + Y      << sequential reads
END FUNCTION FOO
…
!omp$ simd private(X,Y)
DO I= 0, N
    Y = B(I)
    X = A(I)
    C(I) += FOO(X, Y)
ENDDO
```

Linear(ref/val/uval) Examples

Things to remember:

- linear(ref(x:[step])) matches to unit/non-unit stride arguments if step match
- linear(ref(x)) matches to private arguments: these are allocated sequentially
- linear(uval(x)) is preferred to linear(val(x)) for by-reference passed read-only linears: uval facilitates more efficient parameter passing. If both specified uval is matched

linear(ref):

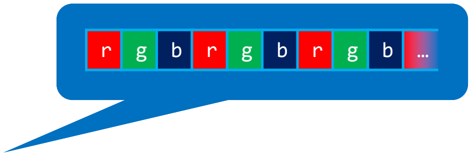
```
#pragma omp declare simd linear(ref(p))
void add_one(int& p) { p += 1; }

int a[NN];

#prgma omp simd linear(p)
for (i = 0; i < NN; i++) {
  add_one(*p);  <<< unit-stride load
  p++;
}

#prgma omp simd private(p)
for (i = 0; i < NN; i++) {
  p = a[i];
  add_one(p);  <<< private
  b[i] = p;
}

#prgma omp simd
for (i = 0; i < NN; i++) {
  add_one(i);  <<< match, incorrect
}
```

Linear(val)/linear(uval):

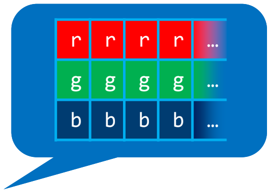
```
interface
   real function func1(x, i)
!$omp declare simd(func1) uniform(x) linear(val(i):1)
      real(8), intent(inout) :: x(*)
      integer, intent(in), value :: i
   end function func1
   real function func2(x, i)
!$omp declare simd(func2) uniform(x) linear(uval(i):1)
      real(8), intent(inout) :: x(*)
      integer, intent(in), value :: i
   end function func2
end interface
…
!$omp simd linear(k:1)
  do i=1, n
     x(i) = func1(x, k)   << k passed as vector of refs
     x(i) = func2(x, k)   << k passed as single ref
     k = k + 1
  enddo
```

SIMD Data Layout Template (SDLT) Library

```
#include <stdio.h>
#include <iostream>

#define N 1024
typedef struct RGBs {
   float r;   float g;  float b;
} RGBTy;

void main()
{
 RGBTy a[N];

 #pragma omp simd
 for(int k=0; k<N; k++) {
   a[k].r = k*1.5;     a[k].g = k*2.5;     a[k].b = k*3.5;
 }
 std::cout << "k ="  << 10 <<
          ", a[k].r ="  << a[10].r <<
          ", a[k].g ="  << a[10].g <<
          ", a[k].b ="  << a[10].b << std::endl;
}
```



```
#include <stdio.h>
#include <sdlt/primitive.h>
#include <sdlt/soa1d_container.h>
#define N 1024
typedef struct RGBs {
   float r;   float g;   float b;
} RGBTy;
SDLT_PRIMITIVE(RGBTy, r, g, b)
void main()
{
 sdlt::soa1d_container<RGBTy> aContainer(N);
 auto a = aContainer.access();
 #pragma omp simd
 for(int k=0; k<N; k++) {
   a[k].r() = k*1.5;  a[k].g() = k*2.5;  a[k].b() = k*3.5;
 }
 std::cout << "k ="  << 10 <<
          ", a[k].r ="  <<  a[10].r() <<
          ", a[k].g ="  << a[10].g() <<
          ", a[k].b ="  << a[10].b() << std::endl;
}
```



SDLT: AOS vs. SOA

AOS AVX512 ASM Code

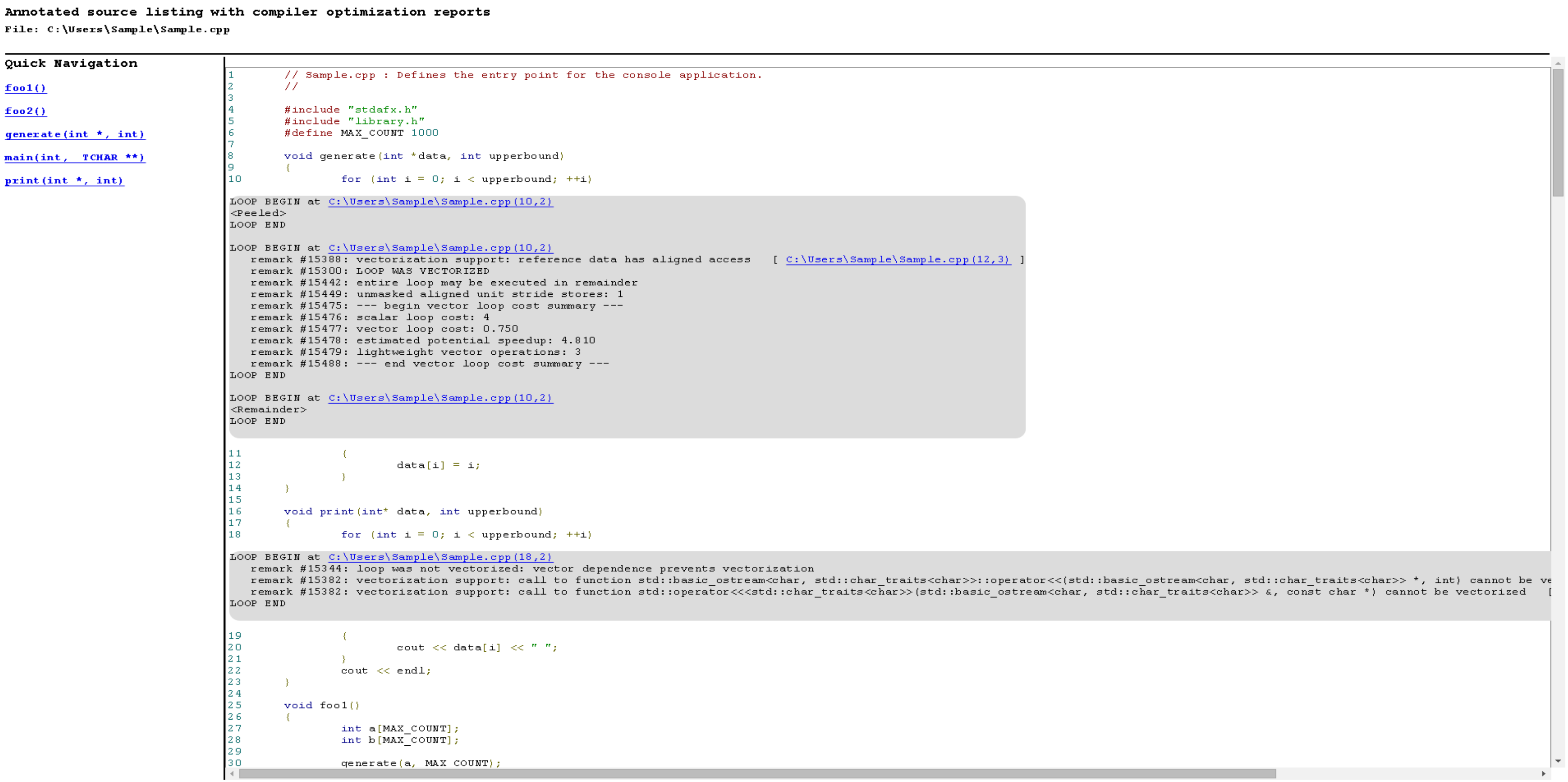
```
..B1.3:             # Preds ..B1.3 ..B1.2
  vcvtdq2ps %zmm5, %zmm11          #18.15
  lea       (%rsp,%rax), %rcx      #18.6
  vcvtdq2ps %zmm4, %zmm12          #18.15
  vpaddd    %zmm6, %zmm5, %zmm5    #16.3
  vpaddd    %zmm6, %zmm4, %zmm4    #16.3
  vmulps    %zmm11, %zmm3, %zmm7   #18.17
  vmulps    %zmm12, %zmm3, %zmm8   #18.17
  vmulps    %zmm11, %zmm2, %zmm9   #19.17
  vmulps    %zmm12, %zmm2, %zmm10  #19.17
  vmulps    %zmm11, %zmm1, %z      #20.17
  vmulps    %zmm12, %zmm1, %zmm1   #20.17
  kxnorw    %k0, %k0, %k1          #18.6
  kxnorw    %k0, %k0, %k2          #18.6
  kxnorw    %k0, %k0, %k3          #19.6
  kxnorw    %k0, %k0, %k4          #19.6
  kxnorw    %k0, %k0, %k5          #20.6
  kxnorw    %k0, %k0, %k6          #20.6
  vscatterdps %zmm7, (%rcx,%zmm0,4){%k1}      #18.6
  vscatterdps %zmm8, 192(%rcx,%zmm0,4){%k2}   #18.6
  addl      $32, %edx              #17.12
  lea       4(%rsp,%rax), %rsi                #18.6
  vscatterdps %zmm9, (%rsi,%zmm0,4){%k3}      #19.6
  lea       8(%rsp,%rax), %rdi                #18.6
  vscatterdps %zmm10, 192(%rsi,%zmm0,4){%k4} #19.6
  vscatterdps %zmm13, (%rdi,%zmm0,4){%k5}     #20.6
  vscatterdps %zmm14, 192(%rdi,%zmm0,4){%k6} #20.6
  addq      $384, %rax             #17.12
  cmpl      $1024, %edx            #17.12
  jb        ..B1.3  # Prob 82%     #17.12
```

SOA / SDLT AVX512 ASM Code: scatter instructions are all gone

```
..B1.5:                    # Preds ..B1.5 ..B1.4
    vpaddd    %zmm4, %zmm3, %zmm12          #19.3
    vcvtdq2ps %zmm3, %zmm7                  #21.17
    vcvtdq2ps %zmm12, %zmm10                #21.17
    vmulps    %zmm7, %zmm2, %zmm5           #21.19
    vmulps    %zmm7, %zmm1, %zmm6           #22.19
    vmulps    %zmm7, %zmm0, %zmm8           #23.19
    vmulps    %zmm10, %zmm2, %zmm3          #21.19
    vmulps    %zmm10, %zmm1, %zmm9          #22.19
    vmulps    %zmm10, %zmm0, %zmm11         #23.19
    vmovups   %zmm5, (%rsi,%rcx,4)          #21.15
    vmovups   %zmm6, (%rdx,%rcx,4)          #22.15
    vmovups   %zmm8, (%rax,%rcx,4)          #23.15
    vmovups   %zmm3, 64(%rsi,%rcx,4)        #21.15
    vmovups   %zmm9, 64(%rdx,%rcx,4)        #22.15
    vmovups   %zmm11, 64(%rax,%rcx,4)       #23.15
    vpaddd    %zmm4, %zmm12, %zmm3          #19.3
    addq      $32, %rcx                     #21.6
    cmpq      $1024, %rcx                   #21.6
    jb        ..B1.5  # Prob 82%            #21.6
```

Optimization Notice

Programmer Friendly Optimization Report



Annotated source listing with compiler optimization reports

File: C:\Users\Sample\Sample.cpp

Quick Navigation

- foo1()
- foo2()
- generate(int *, int)
- main(int, TCHAR **)
- print(int *, int)

```
        // Sample.cpp : Defines the entry point for the console application.
        //

        #include "stdafx.h"
        #include "library.h"
        #define MAX_COUNT 1000

        void generate(int *data, int upperbound)
        {
                for (int i = 0; i < upperbound; ++i)

LOOP BEGIN at C:\Users\Sample\Sample.cpp(10,2)
<Peeled>
LOOP END

LOOP BEGIN at C:\Users\Sample\Sample.cpp(10,2)
   remark #15388: vectorization support: reference data has aligned access   [ C:\Users\Sample\Sample.cpp(12,3) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 4
   remark #15477: vector loop cost: 0.750
   remark #15478: estimated potential speedup: 4.810
   remark #15479: lightweight vector operations: 3
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at C:\Users\Sample\Sample.cpp(10,2)
<Remainder>
LOOP END

                {
                        data[i] = i;
                }
        }

        void print(int* data, int upperbound)
        {
                for (int i = 0; i < upperbound; ++i)

LOOP BEGIN at C:\Users\Sample\Sample.cpp(18,2)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15382: vectorization support: call to function std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>> *, int) cannot be ve
   remark #15382: vectorization support: call to function std::operator<<<std::char_traits<char>>(std::basic_ostream<char, std::char_traits<char>> &, const char *) cannot be vectorized   [
LOOP END

                {
                        cout << data[i] << " ";
                }
                cout << endl;
        }

        void foo1()
        {
                int a[MAX_COUNT];
                int b[MAX_COUNT];

                generate(a, MAX_COUNT);
```

Optimization Notice

Optimization Report Improvements

- ✓ Significant improvement in variable names and memory references reporting

```
16.0: remark #15346: vector dependence: assumed ANTI dependence between  line 108 and  line 116
17.0: remark #15346: vector dependence: assumed ANTI dependence between *(s1) (108:2) and *(r+4) (116:2)
```

- ✓ More precise non-vectorization reasons
  - ○ E.g.: “exception handling for function call prevents vectorization”

- ✓ Gather and partial scalarization reasons reporting (-qopt-report:5)

```
16.0: remark #15328: vectorization support: gather was emulated for the variable xyBase: indirect access
[scalar_dslash_fused.cpp(334,27)]

17.0:  remark #15328: vectorization support: gather was emulated for the variable
<xyBase[xbOffset][c][s][1]>, indirect access, part of index is conditional
[scalar_dslash_fused.cpp(334,27)]
```

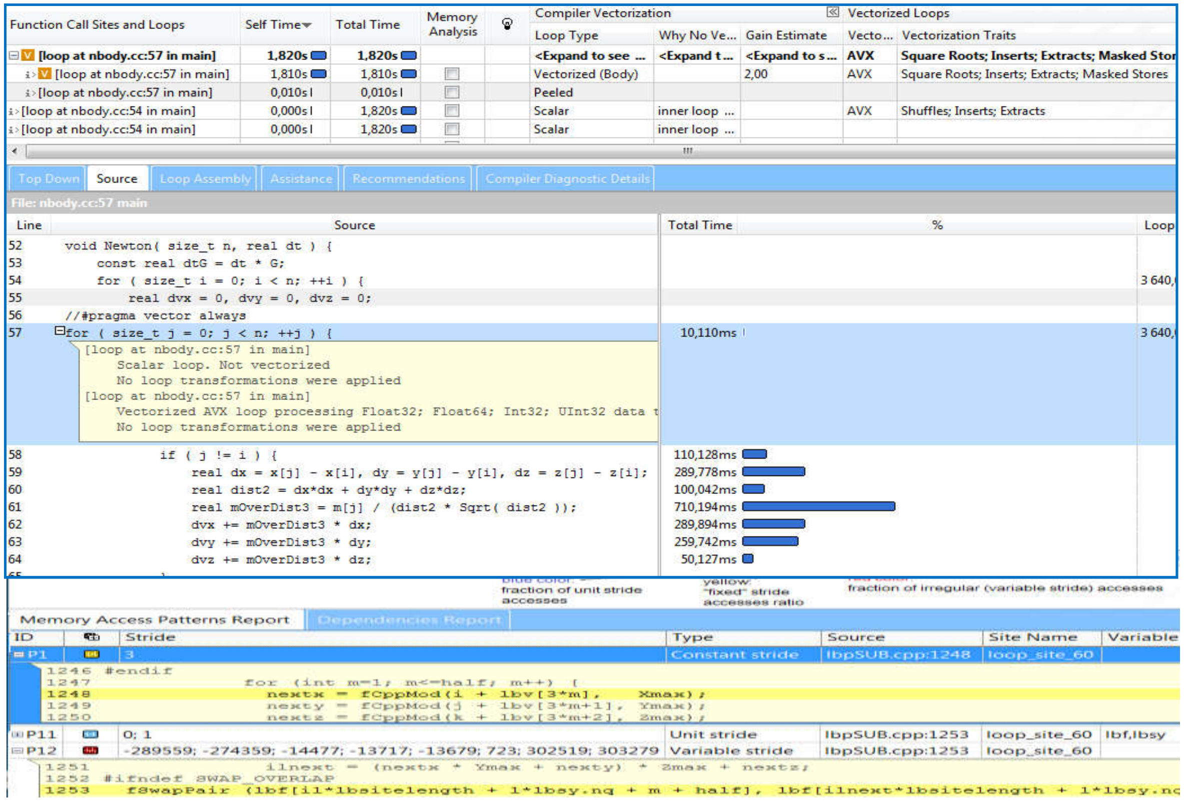
Other reasons are:

- ○ read from memory
- ○ nonlinearly computed
- ○ is result of a call to function
- ○ is linear but may overflow ← either in unsigned indexing or in address computation
- ○ is private ← memory privatization in explicit vectorization or serialized computation

Vectorization Advisor

Assist code vectorization for Intel® SIMD (Zakhar A. Matveev)

- ✓ **All the data you need in one place**
  - ✓ *Combines **Intel Compiler opt-report** with **dynamic profile**.*
- ✓ **Detects "hot" un-vectorized or "under vectorized" loops.**
- ✓ **Identify performance penalties and recommend fixes**
  - ✓ *Explicit advices with "true intelligence" including **OpenMP4.x***
- ✓ **Memory layout (stride) analysis**
- ✓ **Increase the confidence that vectorization is safe**



Function Call Sites and Loops	Self Time▾	Total Time	Memory Analysis		Compiler Vectorization			Vectorized Loops	
					Loop Type	Why No Ve...	Gain Estimate	Vecto...	Vectorization Traits
⊟ V [loop at nbody.cc:57 in main]	1,820s	1,820s			<Expand to see ...	<Expand t...	<Expand to s...	AVX	Square Roots; Inserts; Extracts; Masked Stor
V [loop at nbody.cc:57 in main]	1,810s	1,810s	□		Vectorized (Body)		2,00	AVX	Square Roots; Inserts; Extracts; Masked Stores
[loop at nbody.cc:57 in main]	0,010s	0,010s	□		Peeled				
[loop at nbody.cc:54 in main]	0,000s	1,820s	□		Scalar	inner loop ...		AVX	Shuffles; Inserts; Extracts
[loop at nbody.cc:54 in main]	0,000s	1,820s	□		Scalar	inner loop ...			

File: nbody.cc:57 main

```
Line  Source                                                   Total Time   %   Loop
52      void Newton( size_t n, real dt ) {
53          const real dtG = dt * G;
54          for ( size_t i = 0; i < n; ++i ) {                                  3 640,
55              real dvx = 0, dvy = 0, dvz = 0;
56      //#pragma vector always
57      ⊟for ( size_t j = 0; j < n; ++j ) {                    10,110ms         3 640,
         [loop at nbody.cc:57 in main]
             Scalar loop. Not vectorized
             No loop transformations were applied
         [loop at nbody.cc:57 in main]
             Vectorized AVX loop processing Float32; Float64; Int32; UInt32 data t
             No loop transformations were applied
58                  if ( j != i ) {                              110,128ms
59                      real dx = x[j] - x[i], dy = y[j] - y[i], dz = z[j] - z[i];   289,778ms
60                      real dist2 = dx*dx + dy*dy + dz*dz;      100,042ms
61                      real mOverDist3 = m[j] / (dist2 * Sqrt( dist2 ));   710,194ms
62                      dvx += mOverDist3 * dx;                  289,894ms
63                      dvy += mOverDist3 * dy;                  259,742ms
64                      dvz += mOverDist3 * dz;                   50,127ms
```

blue color: fraction of unit stride accesses
yellow: "fixed" stride accesses ratio
red color: fraction of irregular (variable stride) accesses
Memory Access Patterns Report
Dependencies Report

ID		Stride	Type	Source	Site Name	Variable
⊟ P1		3	Constant stride	lbpSUB.cpp:1248	loop_site_60	
⊞ P11		0; 1	Unit stride	lbpSUB.cpp:1253	loop_site_60	lbf,lbsy
⊟ P12		-289559; -274359; -14477; -13717; -13679; 723; 302519; 303279	Variable stride	lbpSUB.cpp:1253	loop_site_60	

```
1246 #endif
1247            for (int m=1; m<=half; m++) {
1248              nextx = fCppMod(i + lbv[3*m],    Xmax);
1249              nexty = fCppMod(j + lbv[3*m+1],  Ymax);
1250              nextz = fCppMod(k + lbv[3*m+2],  Zmax);
```

```
1251              ilnext = (nextx * Ymax + nexty) * Zmax + nextz;
1252 #ifndef SWAP_OVERLAP
1253      fSwapPair (lbf[il*lbsitelength + l*lbsy.nq + m + half], lbf[ilnext*lbsitelength + l*lbsy.nq
```

Optimization Notice

Advances: Tackle C++ Challenges and Beyond

Virtual SIMD Functions

✓ Syntax

- Exactly same syntax as for usual vector functions

✓ Inheritance:

- Set of versions inherited and cannot be altered in overrides
- Implication: SIMDness should be introduced along with virtual method, not in overrides

✓ Things to remember:

- Performance depends on divergence:
  - uniform(this) fastest: single call per chunk
  - Different overrides in lanes slowest: loop for each unique call target
- Limitations:
  - Multiple inheritance is not supported
  - Pointers to virtual vector methods are not supported

```
class A {
public:
 #pragma omp declare simd linear(x)
 #pragma omp declare simd uniform(this) linear(x)
    virtual int foo(int x);
};

#pragma omp declare simd uniform(this) linear(x)
int A::foo(int x){ return x+1; }

class B : public A {
public:
  // #pragma omp declare simd linear(x)   -  inherited
  // #pragma omp declare simd uniform(this) linear(x)
   int foo(int x) { return (x*x); }
};

int main() {
  A* b[N], a = new B();
  int sum=0;

  for (int i=0; i < N; i++) {
     b[i] = (i % 6) < 2 ? new A() : new B();
  }

  #pragma omp simd reduction (+:sum)
  for (int i=0; i < N; i++) {
     sum += a->foo(i);  // uniform(this) matched
  }                     // one call per chunk

 #pragma omp simd reduction (+:sum)
  for (int i=0; i < N; i++) {
     sum += b[i]->foo(i);  // linear(x) matched
}                          // 1 or 2 calls per chunk
```

Optimization Notice

SIMD Function Pointers

```
// SIMD vector pointer declaration annotation
#pragma omp declare simd                  // universal but slowest definition matches the use in all three loops
#pragma omp declare simd  linear(in1), linear(ref(in2)), uniform(mul)       // matches the use in the first loop
int (*funcptr)(int* in1, int& in2, int mul);
```

```
int *a, *b, mul, *c;
int *ndx, nn;
...
   // loop examples
   #pragma omp simd
   for (int i = 0; i < nn; i++) {
       c[i] = func(a + i, *(b + i), mul); // in the loop, the first arg is changed linearly,
                                          // the second reference is changed linearly too
                                          // the third parameter is not changed
  }
```

Vector Function Pointers

Enable: **`-simd-function-pointers/ -Qsimd-function-pointers`**

The syntax:

- Apply usual vector function declaration to a function pointer variable or function pointer typedef
- Assign address of compatible SIMD-enabled function to a pointer or pass as parameter

Things to remember

- Performance depends on divergence
- Scalar function pointers and vector function pointers are binary incompatible
- Vector specifications are not part of a type (at least not in current implementation)
  - They may not be used for function overloading or template instantiation. No specific name mangling done for e.g. parameters of such types.
  - Situations that may lead to run-time ambiguities are caught and error "***Error #3757: this use of a vector function type is not fully supported"*** reported
  - If you are sure that no ambiguity possible (e.g. function accepting vector function pointer has distinct name and fully declared before all uses) you may override the error via **–wd3757** command line switch

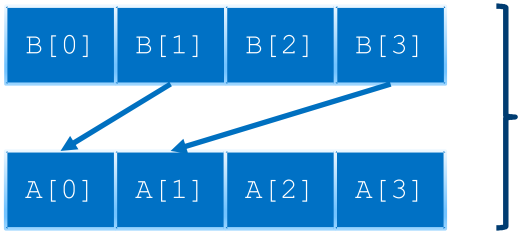
Vectorizing Loop with vcompress/vexpand

Compress

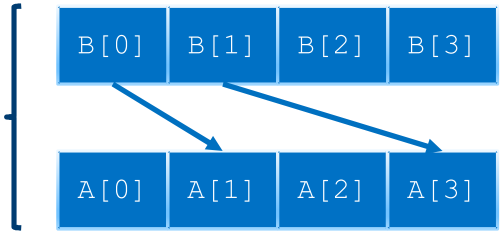
```
count = 0;
#pragma omp simd
for(i){
  if (cond(i)){
   #pragma omp ordered simd
    {
      count++;
      A[count] = B[i];//compress
    }
  }
}
```

Expand

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



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



Optimization Notice

Compress/Expand with Monotonic Semantics

```
count = 0; inc = 1;
#pragma omp simd
for(i=0; i<N; i++) {
#pragma omp ordered simd monotonic(count: inc)  // proposed clause
  {
    if (cond(i)) {
      A[count] += B[i];  // compress
      count+=inc
      B[i] = C[count];    // expand
     }
  }
}
```

Vector Operation with compress/expand

```
{
   st1 = count
   vt2 = maskload(B[i], cond(i))
   compressstore(A[st1], cond(i), vt2)
   count += popcount(cond(i))
   st1 += inc
   vt3 = expandload(C[st1], cond(i))
   maskstore(B[i], cond(i), vt3)
}
```

Integral part of **Intel® Parallel Studio XE**

As a software developer, I care about:		...and my challenges are:	Intel compilers offer:
	Performance – I develop applications that need to execute FAST	Taking advantage of the latest hardware innovations	Developers the full power of the latest x86-compatible processors and instruction sets
	Productivity – I need productivity and ease of use offered by compilers	Finding support for the leading languages and programming models	Support for the latest Fortran, C/C++, and OpenMP* standards; compatibility with leading compilers and IDEs
	Scalability – I develop and debug my application locally, and deploy my application globally	Maintaining my code as core counts and vector widths increase at a fast pace	Scalable performance without changing code as newer generation processors are introduced

Summary: Close to Metal Performance via Explicit SIMD Programming

The reality:

- There is **no one single solution** that would make all programmers happy after decades of trying.
- There is **no free lunch** for effectively utilizing SIMD HW in multicore CPUs, accelerators and GPUs.
- There are **many emerging programming models** for multicore CPUs, accelerators and GPUs.
- Programming languages and compilers are driven by hardware and application
- The incremental approach of applying the learnings from HPC and graphics is working

Simple programming language extensions for computational use of SIMD Hardware

Portable and consistent SIMD programming model across CPU, Coprocessors and GPUs

CODE MODERNIZATION: BEST PRACTICES IN VECTOR PROGRAMMING

Robert Geva – Senior Principal Engineer
Intel Corporation

September 11, 2016

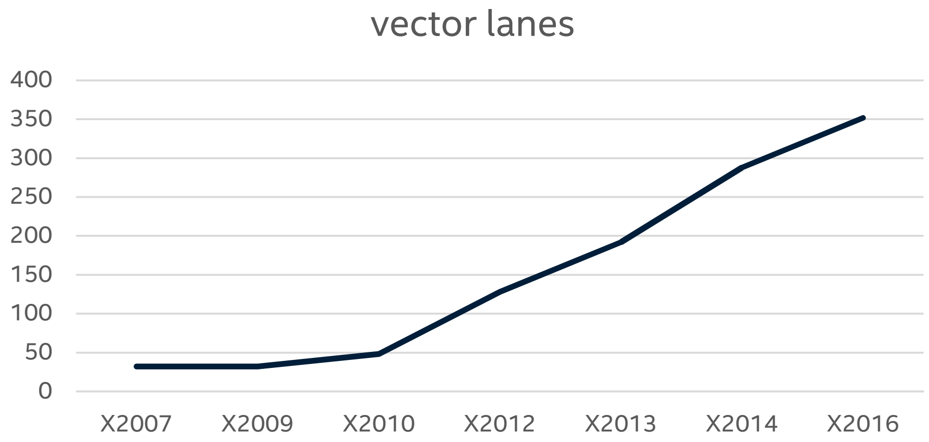
PACT 2016 Tutorial, Haifa, Israel

Section III: Agenda

Introduction	•HW support growing over time •Performance impact of SIMD •Vector programming as part of parallel programming
Case studies	•Based on joint projects with Intel's customers •Myth debunking: vectorization is about adding #pragma to the right loops
The proposal being processed at the C++ standard	•Why OpenMP is inadequate •What stays similar to OpenMP •How it is different •Future extensions to the proposal
Design patterns	•Best practices in using the vector syntax in algorithms •AVX512 specific patterns
Performance portability	•How to write GPGPU kernels in OpenMP •Myth debunking: why the same code for CPU ad GPU cannot be optimal for the CPU •Performance data: methodology, case studies and results

Parallel computing with Intel Architecture

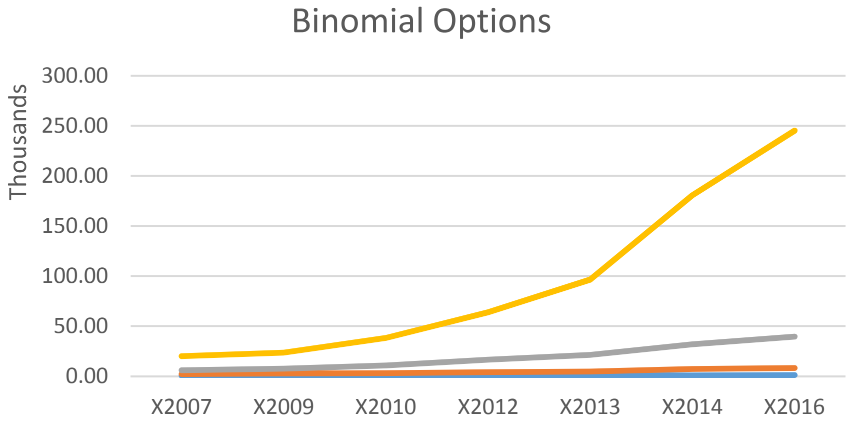
	Cores	SIMD	LANES
X2007	8	128	32
X2009	8	128	32
X2010	12	128	48
X2012	16	256	128
X2013	24	256	192
X2014	36	256	288
X2016	44	256	352



(1) Incremental growth in CPU resources

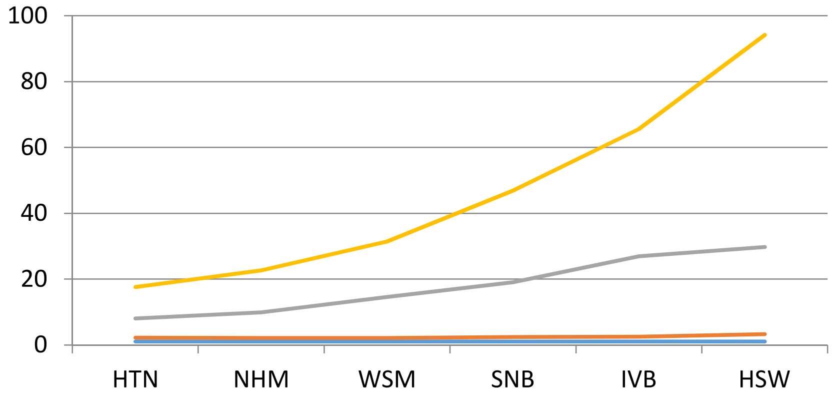
(2) Improvements in compilers and parallel frameworks

(3) Better Parallelization techniques

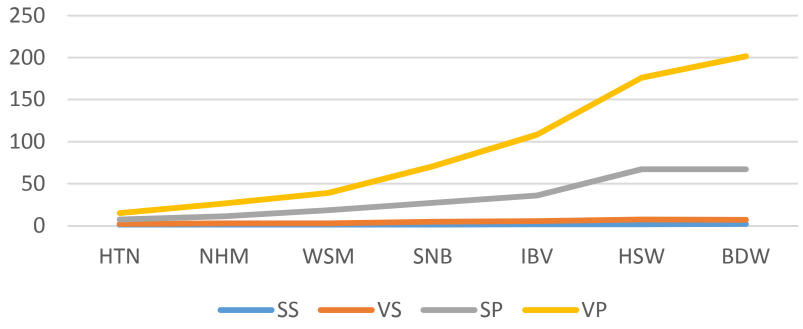


Parallel performance over time

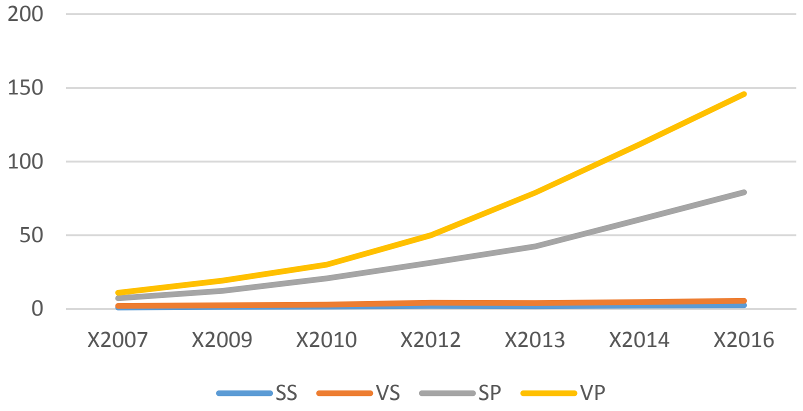
LIBOR Market Model normalized



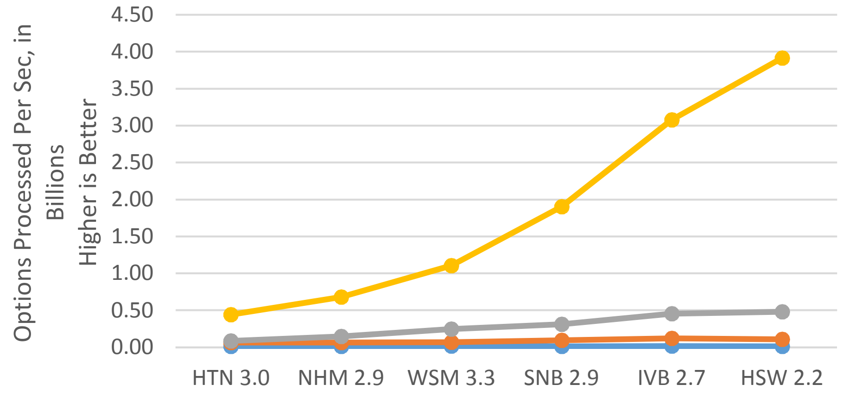
Monte Carlo Asian Options



Monte Carlo American Options

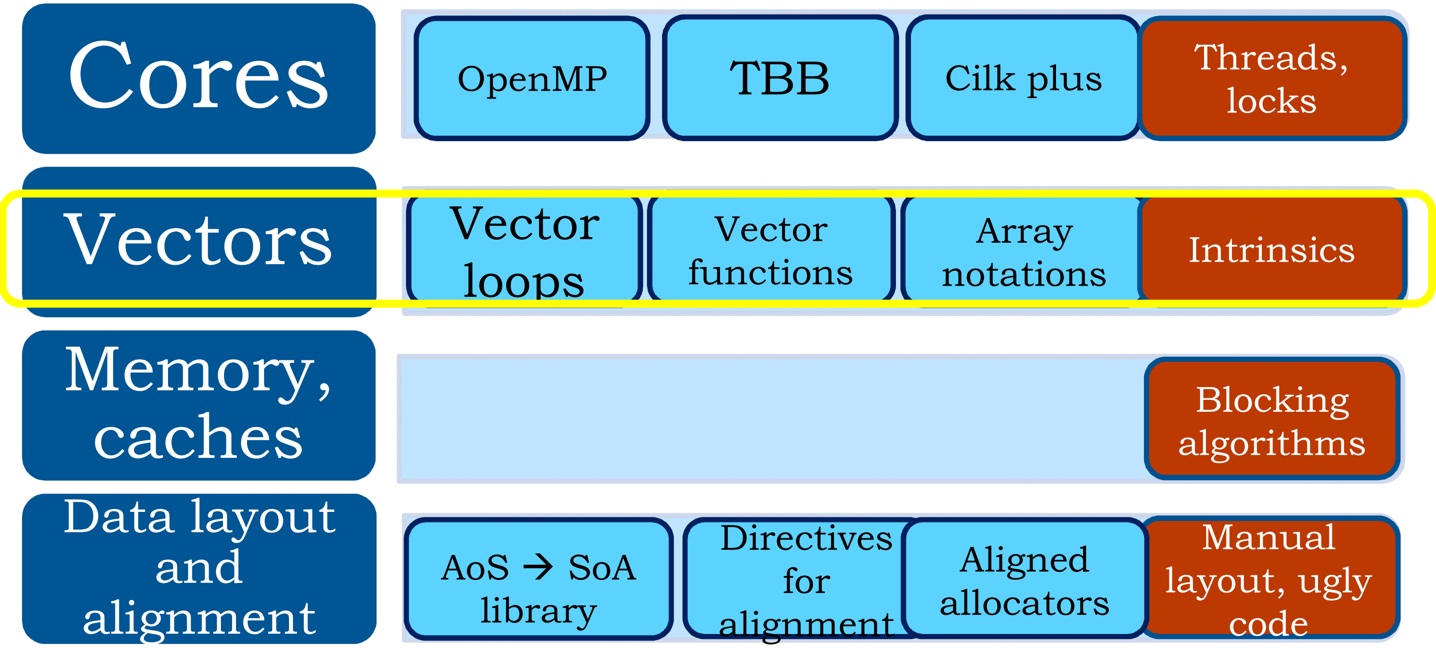


Black Scholes DP



Parallel Programming for Intel® Architecture

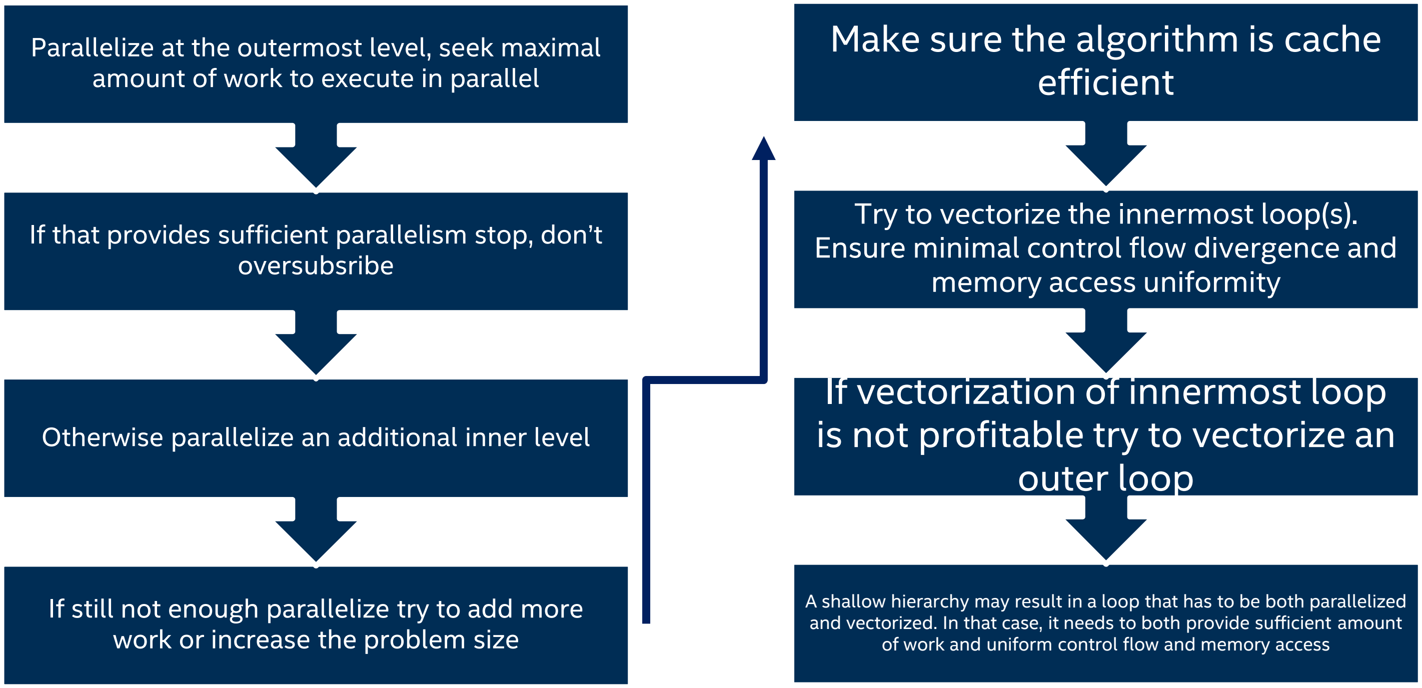
(or, in general, for normal CPUs)



Cores	OpenMP	TBB	Cilk plus	Threads, locks
Vectors	Vector loops	Vector functions	Array notations	Intrinsics
Memory, caches				Blocking algorithms
Data layout and alignment	AoS → SoA library	Directives for alignment	Aligned allocators	Manual layout, ugly code

4 considerations to take care of when writing an efficient, unconstrained parallel program

#1 Best Practice in Parallelizing a Loop Hierarchy



Parallelize at the outermost level, seek maximal amount of work to execute in parallel

If that provides sufficient parallelism stop, don't oversubsribe

Otherwise parallelize an additional inner level

If still not enough parallelize try to add more work or increase the problem size

Make sure the algorithm is cache efficient

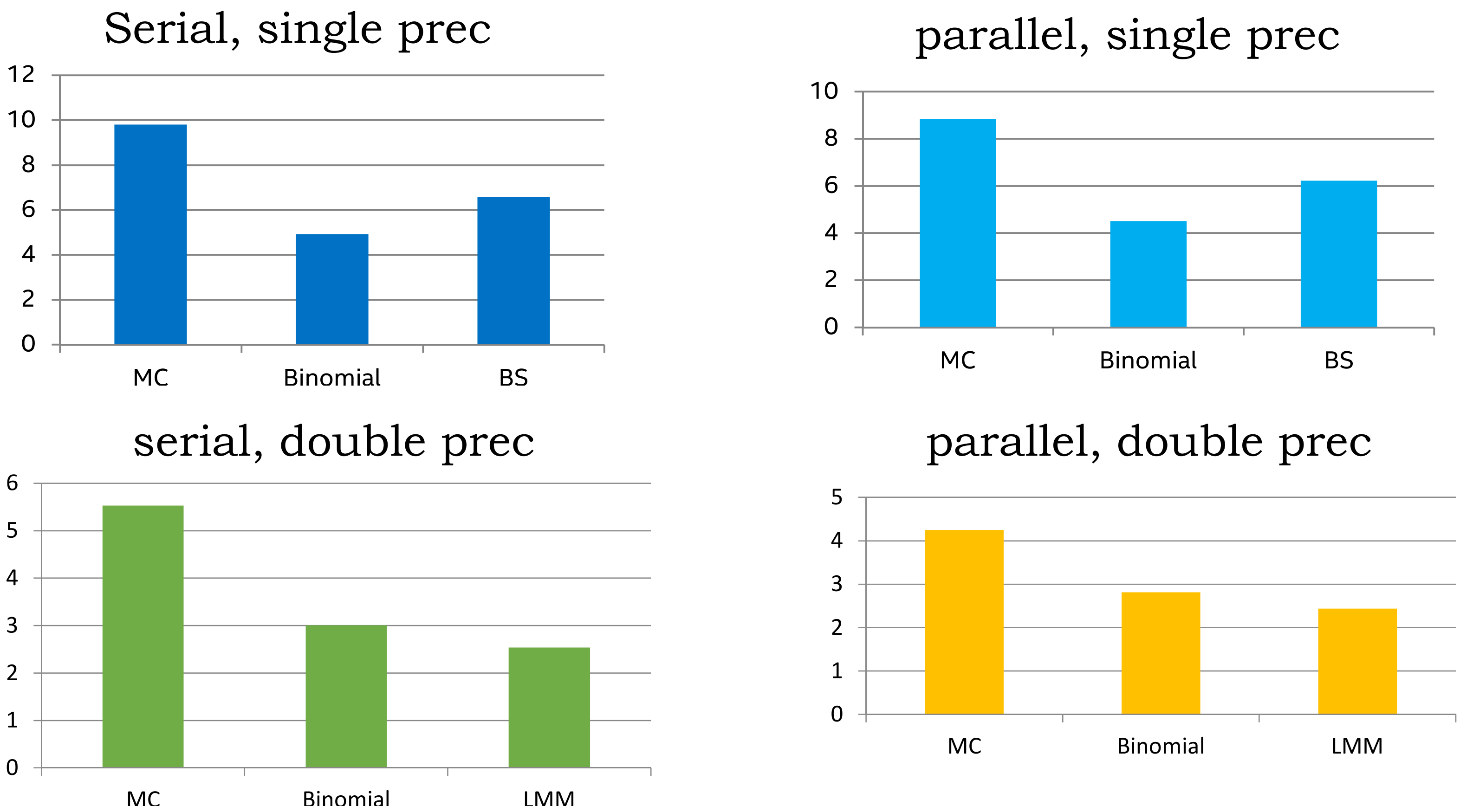
Try to vectorize the innermost loop(s). Ensure minimal control flow divergence and memory access uniformity

If vectorization of innermost loop is not profitable try to vectorize an outer loop

A shallow hierarchy may result in a loop that has to be both parallelized and vectorized. In that case, it needs to both provide sufficient amount of work and uniform control flow and memory access

Vectorize Innermost, Parallelize Outermost (VIPO)

Performance with vector parallelism


Serial, single prec

parallel, single prec

serial, double prec

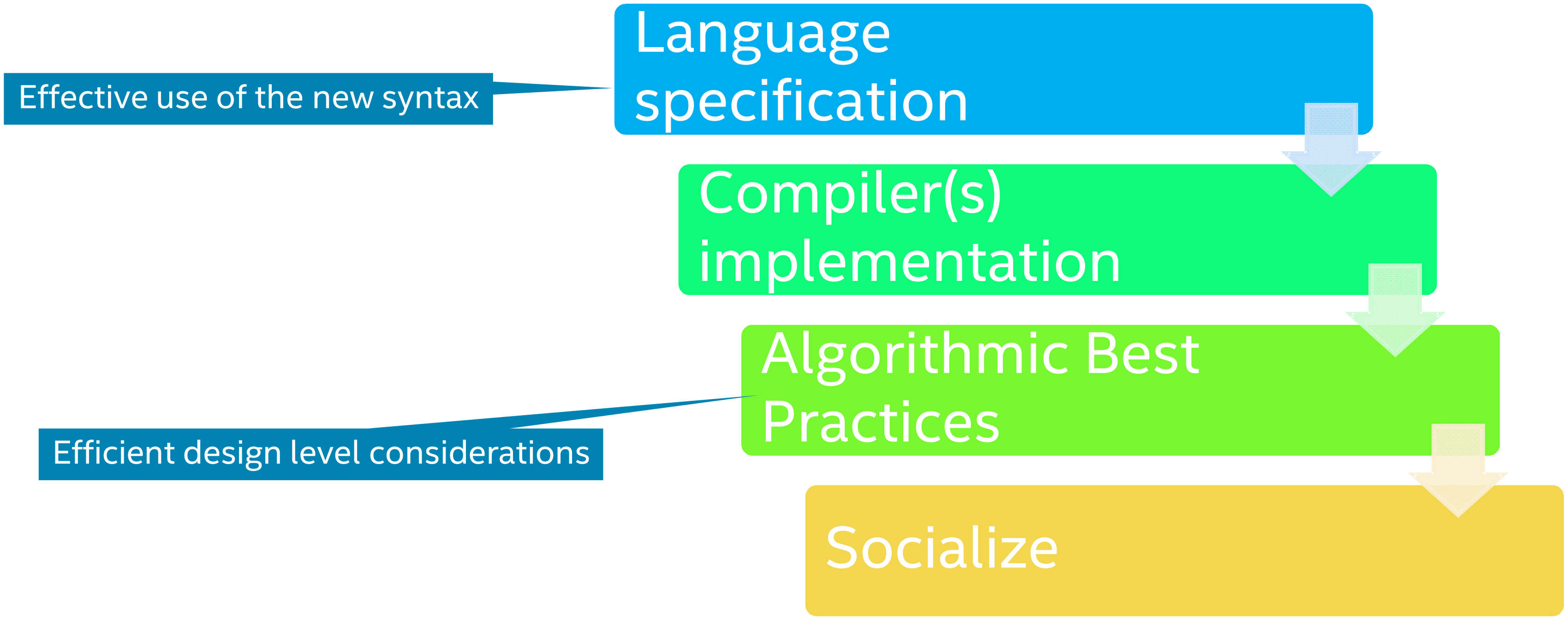
parallel, double prec

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.
Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions.
Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup slides for configuration details. **For more information go to http://www.intel.com/performance**

Configuration

SOURCE: INTEL MEASURED RESULTS AS OF March, 2014

Vector Programming (Part of Parallel Programming)



Effective use of the new syntax

Language specification

Compiler(s) implementation

Algorithmic Best Practices

Efficient design level considerations

Socialize

Capabilities in Vector Programming

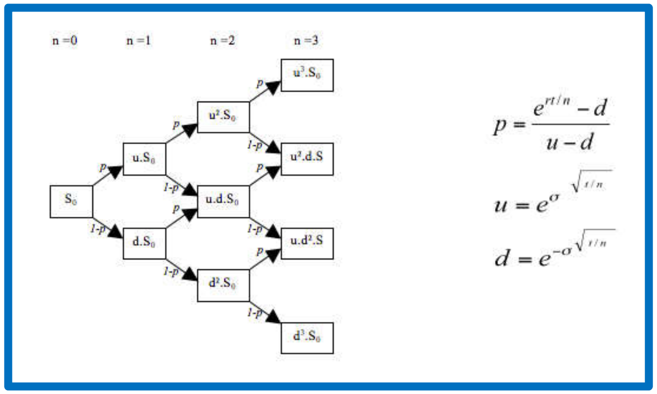
```
#pragma omp simd
for(i = 0; i<N; ++i)
{
    a[i] = b[i]+c[i] ;
}
```

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

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

Case studies

Case Study: Trinomial options


n =0
n =1
n =2
n =3
S₀
u.S₀
d.S₀
u².S₀
u.d.S₀
d².S₀
u³.S₀
u².d.S
u.d².S
d³.S₀
p
1-p
$p = \frac{e^{rt/n} - d}{u - d}$
$u = e^{\sigma \sqrt{t/n}}$
$d = e^{-\sigma \sqrt{t/n}}$

sec		How was it done
31		GCC Baseline
28.7	1.1X	ICC Baseline
19.0	1.6X	Vectorize critical loop.
14.3	2.2X	Use AVX
11.6	2.7X	Add aligned vectors
0.84	37X	Parallelize with OMP
0.60	**51X**	Move to Intel® Xeon Phi coprocessor

Vectorize the inner loop

```
#pragma omp simd
for (int n = 0; n < 2*m + 1; n++)
{
   vCurrent[n] = fmax( BsAnalyticIntrinsicValue(TreeGetAdjustedSpot(m, n, params), kMod,  isCall),
       discount*(ExpectedValue(vCurrent[n], vCurrent[n + 1], vCurrent[n + 2], params)));
}
```

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.
Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions.
Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases

Asian Options example: Vectorize an outer loop

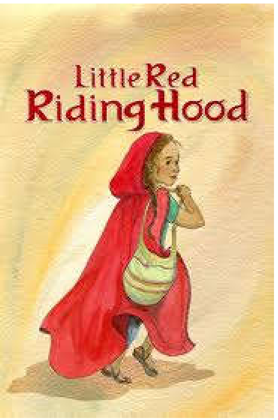
```
#pragma omp parallel for simd reduction(+:val) reduction(+:val2)
for(int pos = 0; pos < RAND_N; pos++)
{
    __declspec(align(64)) tfloat simStepResult[SIMSTEPS+1];
    simStepResult[0] = 1.0;
    tfloat avgMean = 0.0;
    for (int simStep =0; simStep < SIMSTEPS; simStep++)
    {
        location = pos*SIMSTEPS + simStep;
        simStepResult[simStep+1] = simStepResult[simStep]*EXP(MuByT + VBySqrtT*l_Random[location]);
        avgMean += simStepResult[simStep+1];
    }

    //Use Arithmetic Mean
    avgMean *= Sval/SIMSTEPS;
    tfloat callValue = max((avgMean - Xval), 0);
    val  += callValue;
    val2 += callValue * callValue;
}
```

LIBOR case study: Vectorize an outer loop with function calls

```
#pragma omp declare simd
static void path_calc_b1(REAL *z, REAL *L, REAL *L2, const REAL* lambda)
{
  int   i, n;
  REAL sqez, lam, con1, v, vrat;
  memcpy(L2, L, NN*sizeof(REAL));
  for (n = 0; n < NMAT; n++) {
    sqez = SQRT_DELTA * z[n];
    v =REAL(0);
    for (i=n+1; i<NN; i++) {
      lam  = lambda[i-n-1];
      con1 = DELTA * lam;
      v   += con1 * L[i] / (REAL(1) + DELTA * L[i]);
      vrat = std::exp(con1 * v + lam * (sqez - REAL(0.5) * con1));
      L[i] = L[i] * vrat;
      L2[i+(n+1)*NN] = L[i];
    }
  }
}
```

A few lines of code removed to fit into the page

```
#pragma omp simd reduction(+: sumv) reduction(+: sumlb)
for (path=0; path<numPaths; path++) {
    path_calc_b1(ptrZ, L, L2, lambda);
    path_calc_b2(L_b, L2, lambda);
}
```



Myth: To vectorize, I have to find the suitable loops and add #pragmas to them.

OpenMP parallelism: wrong way

```
nbnds=jend-jstart+1 ! [jstart,jend)
!$OMP PARALLEL FOR collapse(2)
DO j=1,nbnds
  DO ir=1,nrxxs
    rho(ir,j)=CONJG(exxbuff(ir,j+jstart))*temppsi(ir)/Omega
  ENDDO
ENDDO
FFTm(rho) ! Batch 3D FFT

!$OMP PARALLEL FOR collapse(2)
DO j=1,nbnds
  DO ir=1,nrxxs
    vc(ir,j)=facb(ir)*rho(ir,j)*x(j+jstart)*y
  ENDDO
ENDDO

invFFTm(vc) ! Batch 3D FFT
!some more on vc
!$OMP PARALLEL FOR
Do ir=1,nrxxs
  Do j=1,nbnds
    result(ir)=result(ir)+vc(ir,j)*exxbuff(ir,j+jstart)
  ENDDO
ENDDO
```

collapse(2) introduced to expose parallelism over nbnds*nrxxs, but

- temppsi shared by j
- cannot be linearlized
- poor cache use

Blocking: Parallelism, SIMD and Cache blocking

```
nbnds=jend-jstart+1 ! [jstart,jend)
nblocks=2048
!$OMP PARALLEL FOR collapse(2)
DO irb=1,nrxxs,nblocks
  DO j=1,nbnds
    irmax=min(nrxxs,irb+nblocks)
!DIR$ vector nontemporal(rho)
    DO ir=irb,irmax
      rho(ir,j)=CONJG(exxbuff(ir,j+jstart))*temppsi(ir)/Omega
    ENDDO
  ENDDO
ENDDO
```

Empirically determined for this loop

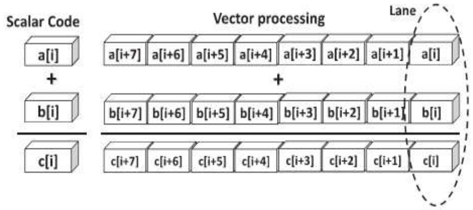
- irb-j loop better than j-irb
- nblocks=2048
- streaming stores (to be explored further later after dungeon)

Other seemingly similar loops favor j-irb

Reduction kernels benefit from blocking

STAC-A2: breaking dependencies

```
void sum(const int* in1, const int* in2, std::size_t
size, int* out)
{
    #pragma omp simd
    for(int  i=0; i<size; ++i ){
        out[i] = in1[i] + in2[i];
    }
}
```



Scalar Code
Vector processing
Lane
a[i]
+
b[i]
c[i]
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
+
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Original Code

```
for (unsigned p = 0; i < nPaths; ++p)
{
    double mV[nTimeSteps];
    double mY[nTimeSteps];
  …..
  for (unsigned int t = 0; t < nTimeSteps; ++t){
       double currState = mY[t] ; // Backward dependency
       ….
       double logSpotPrice = func(currState, …);
       mY[t+1][p] = logSpotPrice * A[t];
       mV[t+1][p] = logSpotPrice * B[t] + C[t] * mV[t][p];
       price[t][p] = logSpotPrice*D[t] +E[t] * mV[t][p];
   }
}
```

Modified Code

```
double mV[nTimeSteps][nPaths];
double mY[nTimeSteps][nPaths];
…
for (unsigned int t = 0; t < nTimeSteps; ++t){
    #pragma omp simd
    for (unsigned p = 0; i < nPaths; ++p)
    {
       double currState = mY[t][p] ;
       ….
       double logSpotPrice = func(currState, …);
       mY[t+1][p] = logSpotPrice * A[t];
       mV[t+1][p] = logSpotPrice * B[t] + C[t] * mV[t][p];
       price[t][p] = logSpotPrice*D[t] +E[t] * mV[t][p];
    }
}
```

Optimization Notice

Intel® TBB parallel blocks, using ranges

```
tbb::parallel_for(blocked_range<int>(0, nPaths),
    [&](const blocked_range<int>& r) {
        cosnt int block_size = r.size();
        double mV[nTimeSteps][block_size];
        double mY[nTimeSteps][block_size];
        …
        for (unsigned int t = 0; t < nTimeSteps; ++t){
            #pragma omp simd
            for (unsigned p = 0; i < block_size; ++p)
            {
                double currState = mY[t][p] ;
                ….
                double logSpotPrice = func(currState, …);
                mY[t+1][p] = logSpotPrice * A[t];
                mV[t+1][p] = logSpotPrice * B[t] + C[t] * mV[t][p];
                price[t][r.begin()+p] = logSpotPrice*D[t] +E[t] * mV[t][p];
           }
}
```

PDE solver

```
void solve_tridigonal (double* x, const int N, double* a, double* b, double
*c, double* cprime)
{
  cprime[0] = c[0] / b[0];
  x[0] = x[0] / b[0];

  for (int in = 1; in < N; in++) {
    REAL m = REAL(1.0) / (b[in] - a[in] * cprime[in - 1]);
    cprime[in] = c[in] * m;
    x[in] = (x[in] - a[in] * x[in - 1]) * m;
  }

  for (int in = N - 2; in-- > 0; )
    x[in] = x[in] - cprime[in] * x[in + 1];
}
```

Most of the time is in the Thomas algorithm, solving a tridiagonal matrix

Efficient and serial

But there is not a single matrix, there are many of them.

```
//Solve 2D PDE
for(int j=1; j<OUTER-1; j++)
{

 for(int i=1; i<INNER-1; i++)
 {
 //Create RHS
  H[i] = v1[i][j];
 }
 h[1] = h[1] - a[1]*v2[0][j];
 h[INNER-2] = h[INNER-2] - c[INNER-2]*v2[INNER-1][j];;

 //Solve Tridiagonal system using Thomas Algorithm
 solve_tridigonal (&h[1], INNER-2, &a[1], &b[1], &c[1], &scratch[1]);
 //copy RHS to v2
 for(int i=1; i<INNER-1; i++)
 {
  v2[i][j] = h[i];
 }
}
```

There is a loop of calls to the solver

Widen the outer loop stride – add a dimension - make space for a new inner loop

```
//Solve 2D PDE
for(int j=1; j<OUTER-1; j+=DIMNSZ)
{
  for(int i=1; i<INNER-1; i++)
  {
#pragma simd
  for(int j1=0; j1<DIMNSZ; j1++)
  {
    int cntJ = j+j1;
    h[i][j1] = v1[i][cntJ];
  }
}
#pragma simd
for(int j1=0; j1<DIMNSZ; j1++)
{
  int cntJ = j+j1;
  h[1][j1] = h[1][j1] - a[1]*v2[0][cntJ];
  h[INNER-2][j1] = h[INNER-2][j1] - c[INNER-2]*v2[INNER-1][cntJ];
}
//Solve Tridiagonal system using Thomas Algorithm
solve_tridigonal_simd(&h[1], INNER - 2, &a[1], &b[1], &c[1], &scratch[1]);
```

Add a dimension

Add a dimension

```
  for(int i=1; i<INNER-1; i++)
  {
#pragma simd
    for(int j1=0; j1<DIMNSZ; j1++)
    {
      int cntJ = j+j1;
      v2[i][cntJ] = h[i][j1];
    }
  }
}
```

Optimization Notice

The new inner loop can vectorize a the new dimension, where there are no dependencies

```
void solve_tridigonal_simd (double x[][DIMNSZ], const int N, double* a, double* b, double *c, double *
cprime)
{
  cprime[0] = c[0] / b[0];

  #pragma simd
  for(int j=0; j<DIMNSZ; j++)
  {
     x[0][j] = x[0][j] / b[0];
  }
  /* loop from 1 to N - 1 inclusive */
  for (int in = 1; in < N; in++)
  {
double tmpA = a[in];
double tmpB = b[in];
double tmpC = c[in];
double m = REAL(1.0) / (tmpB - tmpA * cprime[in - 1]);
cprime[in] = tmpC * m;
```

```
#pragma simd
    for (int j =0; j<DIMNSZ; j++)
    {
      x[in][j] = (x[in][j] - tmpA * x[in - 1][j]) *m;
    }
 }
 for (int in = N - 2; in-- > 0; )
 {
    double cPrime = cprime[in];
    #pragma simd
    for (int j =0; j<DIMNSZ; j++)
    {
      x[in][j] = x[in][j] - cPrime * x[in + 1][j];
    }
                }

}
```

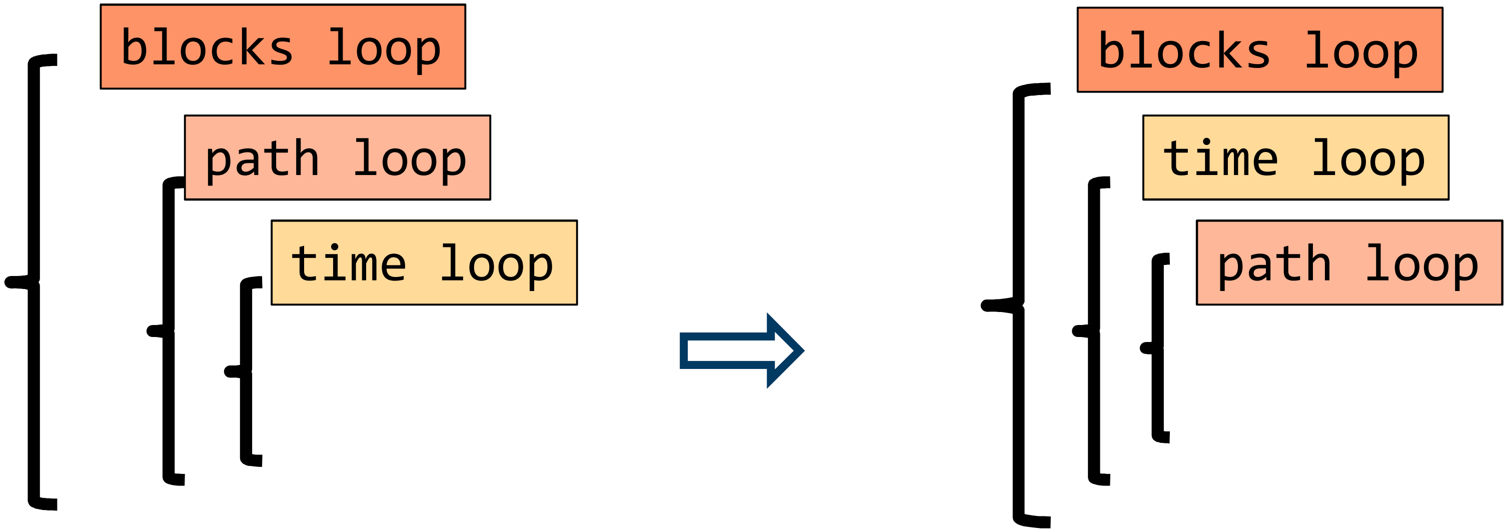
Citi | Market Quantitative Analysis

FX LSV MONTE CARLO CASE STUDY

Thomas Trenner

Robert Geva

Loop interchange leads to canonical loop hierarchy



blocks loop

path loop

time loop

⇨

blocks loop

time loop

path loop

blocks loops – data independent
path loop – data independent
time loop – value at t depends
on value at t-1

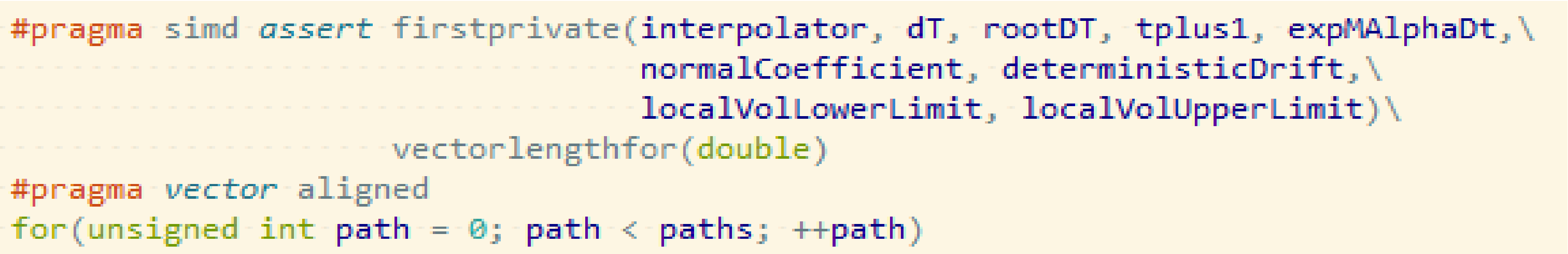
Parallelize the outer loop
keep the middle loop sequential
vectorize the inner loop:
This works best!

Results

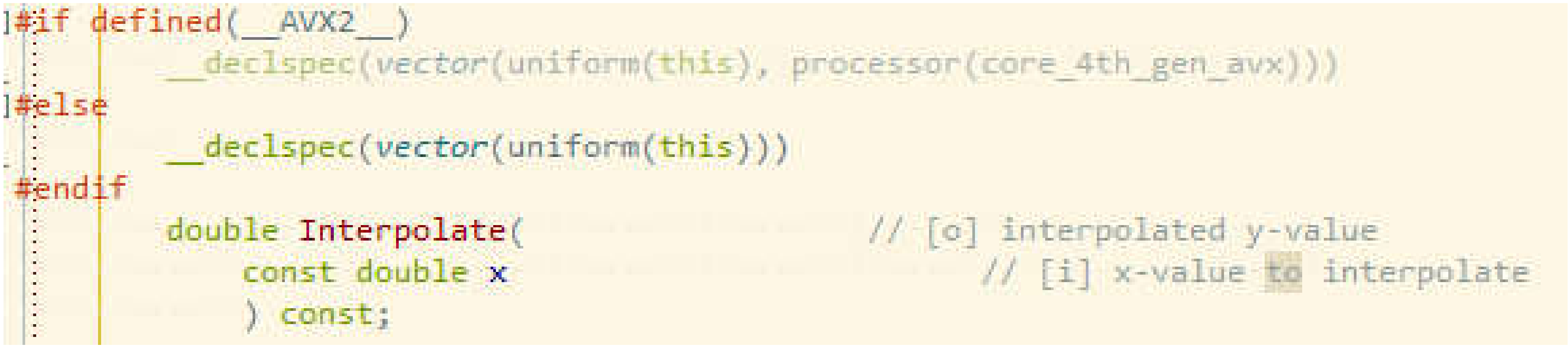
Step	Thread count	Time (Secs)	Measurements were taken on an E5-2697 Haswell with • ICC v 15.0.1 • RHEL 6.5
1	1	82.6	Baseline Results
2	1	24.6	Loop interchange
3	1	11.3	With vectorization **2.2x Vectorization Speedup.**
4	1	9.9	+MKL RNG.
5	1	9.4	+TBB allocator **8.8x Single Threaded Speedup.**
6	28	0.47	+TBB Threads
7	28	0.45	+Turbo mode enabled. **183x Multi threaded Speedup.**

Optimization Notice

Vectorization



```
#pragma simd assert firstprivate(interpolator, dT, rootDT, tplus1, expMAlphaDt,\
                                  normalCoefficient, deterministicDrift,\
                                  localVolLowerLimit, localVolUpperLimit)\
                    vectorlengthfor(double)
#pragma vector aligned
for(unsigned int path = 0; path < paths; ++path)
```



```
#if defined(__AVX2__)
        __declspec(vector(uniform(this), processor(core_4th_gen_avx)))
#else
        __declspec(vector(uniform(this)))
#endif
        double Interpolate(                           // [o] interpolated y-value
            const double x                                // [i] x-value to interpolate
            ) const;
```

Concepts used in vectorising the loop

#pragma simd:

- semantically correct to re-associate the order of evaluation, leading to vectorization

First private:

- each iteration gets a private copy of the object, initialized by the value it has prior to the loop

Assert:

- abort compilation with an error message if the loop is not vectorized

Vectorlengthfor(type):

- The size of type determines how many loop iteration to vectorize across

declspec(vector):

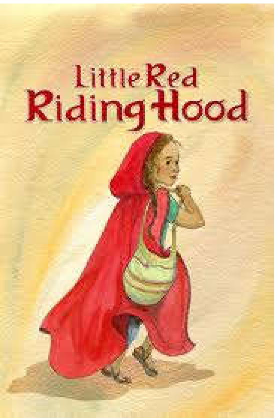
- A vector function. Compiled and execute as if a body of a vector loop

Uniform(parameter):

- All values of parameter in one vector invocation of the function are the same

Processor(code_4th_gen_avx):

- use YMM to pass arguments, (the default is XMM)



Reality: in most of real life cases, the loop that ended up vectorized, did not exist in the original code. It resulted due to one of the restructuring best practices

Design patterns

A simple patterns

```
#pragma omp parallel simd for
for (int i = 0; i < m_optionCount; i++)
    BlackScholesBodyCPU(&resultCallGen[i], &resultPutGen[i],
    stockPrice[i], optionStrike[i], optionYears[i],R,m_V);
```

```
#prgma omp declare simd
void BlackScholesBodyCPU(
   float* call, //Call option price
   float* put,  //Put option price
   float Sf,   //Current stock price
   float Xf,   //Option strike price
   float Tf,   //Option years
   float Rf,   //Riskless rate of return
   float  Vf    //Stock volatility
){
   float S = Sf, X = Xf, T = Tf, R = Rf, V = Vf;
   float CNDD1, CNDD2, sqrtT, expRT
    sqrtT = sqrtf(T);
   d1 = (logf(S / X) + (R + 0.5f * V * V) * T) / (V * sqrtT);
   d2 = d1 - V * sqrtT;
   CNDD1 = CND(d1);
   CNDD2 = CND(d2);
   expRT = expf(- R * T);
   *call = (FTYPE)(S * CNDD1 - X * expRT * CNDD2);
   *put  = (FTYPE)(X * expRT * (1.0f - CNDD2) - S * (1.0f -
CNDD1));
}
```

A loop with no data dependencies
No control flow
Simple memory access
Could also be parallelized
Scales well

Optimization Notice

A loop with forward dependence

```
float stepsArray[STEPS_CACHE_SIZE];
#pragma omp simd
for (int j = 0; j < STEPS_CACHE_SIZE; j++) {
    float profit = s * expf(vsdt * (2.0f * j - numSteps)) - x;
    stepsArray[j] = profit > 0.0f ? profit : 0.0f;
}
for (int j = 0; j < numSteps; j++) {
    #pragma omp simd
    for (int k = 0; k < NUM_STEPS_ROUND; ++k) {
        stepsArray[k] = pdByr * stepsArray[k + 1] + puByr * stepsArray[k];
    }
}
```

The vector loops propagates values from root to leaves
Looks very similar to original, sequential loop
stepArray[k] depends on stepArray[k+1], vector programming supports it
SIMD != SIMT
Parallelization is at an outer level.

Parameter Qualifiers

```
#pragma omp declare simd
void foo (float &a, int i)
{

    x = a[i];

}
```

Compiling the vector variant will generate multiple expressions of x = a[i] – what are the relationship between the memory accesses?

If the compiler doesn't know better, then they are unrelated.

```
#pragma omp declare simd uniform (a)
void foo(float *a, int i);
```

- a is a pointer
- i is a vector of integers
- a[i] becomes gather/scatter

```
#pragma omp declare simd linear(i)
void foo(float *a, int i);
```

- a is a vector of pointers
- i is a sequence of integers [i, i+1, i+2…]
- a[i] becomes gather/scatter

```
#pragma omp declare simd
uniform(a),linear(i))
void foo(float *a, int i);
```

- a is a pointer
- i is a sequence of integers [i, i+1, i+2…]
- a[i] is a unit-stride load/store

Multiple Variants of a Vector Function

```
#pragma omp declare simd
#pragma omp declare simd uniform(r,op1,op2) linear (i)
Void
vec_add ( float *r, float *op1, float *op2, int i)
{
    r[i] = op1[i] + op2[i];
}
```

Two vector variants and one scalar

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

Call matches the variant with the uniforms

```
#pragma omp simd
for (int i = 0; i<N; ++i) {
    vec_add(a[x1[[i]],b[x2[i]],c[x3[i]],i);
}
```

Call matches the variant w/o the uniforms

Multiple Variants of a Vector Function

```
~~#pragma omp declare simd~~
#pragma omp declare simd uniform(r,op1,op2) linear (i)
Void
vec_add ( float *r, float *op1, float *op2, int i)
{
    r[i] = op1[i] + op2[i];
}
```

Two vector variants and one scalar

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

Call matches the variant with the uniforms

```
#pragma omp simd
for (int i = 0; i<N; ++i) {
    vec_add(a[x1[[i]],b[x2[i]],c[x3[i]],i);
}
```

Call matches the variant w/o the uniforms

Multiple Variants of a Vector Function

```
#pragma omp declare simd
~~#pragma omp declare simd uniform(r,op1,op2) linear (i)~~
Void
vec_add ( float *r, float *op1, float *op2, int i)
{
    r[i] = op1[i] + op2[i];
}
```

Two vector variants and one scalar

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

Call matches the variant with the uniforms

```
#pragma omp simd
for (int i = 0; i<N; ++i) {
    vec_add(a[x1[[i]],b[x2[i]],c[x3[i]],i);
}
```

Call matches the variant w/o the uniforms

Additional SIMD specific capabilities

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];`

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);
  }
```

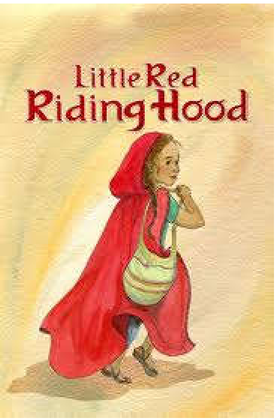
Partition By Weight

```
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; //compressed
    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

Performance portability



Myth: I'd like to write the same code for a CPU and a GPU and have it perform within 5% of optimal.

Questions related to performance portability

A. Interest in maintaining a single source base across CPUs and GPUs

B. Interest in languages that provide good support for both

C. Interest in OpenCL

D. Interest in conversion of CUDA code to CPUs / Xeon Phi

Multiple parallelization related consideration are different, with potentially significant impact, limiting the potential for performance portability

Wider fan out vs more work / worker

Account for memory and cache efficiency, tiling, blocking

Account for cores per socket, hyper threads per core, NUMA effects

Today’s focus: difference in vectorization.

Rationale

SIMT- style kernels are too restrictive

- There are many parallel algorithms and design patterns
- In many cases, the kernel is not the optimal design pattern
- Then, a kernel- only language or a kernel only algorithmic design locks you out of a solution

A trivial case: Black Scholes

Example of kernels being inadequate: Binomial options

Data: writing kernels in OpenMP vs. writing loops in OpenMP,

- same language
- Same compiler
- Same HW
- Large performance impact

GPU kernels and CPU loop hierarchies

CUDA

```
BlackScholesGPU<<<256, 128>>>(
  d_CallResult, d_PutResult, d_OptionStrike, d_StockPrice,
  d_OptionYears, RISKFREE, VOLATILITY, OPT_N);
```

```
__device__
void BlackScholesBodyGPU(
   float& CallResult,
   float& PutResult,
   float S, //Stock price
   float X, //Option strike
   float T, //Option years
   float R, //Riskless rate
   float V  //Volatility rate
){
   float sqrtT, expRT;
   float d1, d2, CNDD1, CNDD2;
   sqrtT = sqrtf(T);
   d1 = (__logf(S / X) + (R + 0.5f * V * V) * T) / (V * sqrtT);
   d2 = d1 - V * sqrtT;
   CNDD1 = cndGPU(d1);
   CNDD2 = cndGPU(d2);
   expRT = __expf(- R * T);
   CallResult = S * CNDD1 - X * expRT * CNDD2;
   PutResult  = X * expRT * (1.0f - CNDD2) - S * (1.0f - CNDD1);
}
```

OpenMP

```
#pragma omp parallel simd for
for (int i = 0; i < m_optionCount; i++)
    BlackScholesBodyCPU(&resultCallGen[i], &resultPutGen[i],
    stockPrice[i], optionStrike[i], optionYears[i],R,m_V);
```

```
#prgma omp declare simd
void BlackScholesBodyCPU(
   float* call, //Call option price
   float* put,  //Put option price
   float Sf,    //Current stock price
   float Xf,    //Option strike price
   float Tf,    //Option years
   float Rf,    //Riskless rate of return
   float  Vf     //Stock volatility
){
   float S = Sf, X = Xf, T = Tf, R = Rf, V = Vf;
   float CNDD1, CNDD2, sqrtT, expRT
    sqrtT = sqrtf(T);
   d1 = (logf(S / X) + (R + 0.5f * V * V) * T) / (V * sqrtT);
   d2 = d1 - V * sqrtT;
   CNDD1 = CND(d1);
   CNDD2 = CND(d2);
   expRT = expf(- R * T);
   *call = (FTYPE)(S * CNDD1 - X * expRT * CNDD2);
   *put  = (FTYPE)(X * expRT * (1.0f - CNDD2) - S * (1.0f -
CNDD1));
}
```

The “kernel” is the same as the body of a parallel and vector loop

Vectorize the inner loop independently of the outer loop. Vectorising with FORWARD dependencies

```
float stepsArray[STEPS_CACHE_SIZE];
#pragma omp simd
for (int j = 0; j < STEPS_CACHE_SIZE; j++) {
    float profit = s * expf(vsdt * (2.0f * j - numSteps)) - x;
    stepsArray[j] = profit > 0.0f ? profit : 0.0f;
}
for (int j = 0; j < numSteps; j++) {
    #pragma omp simd
    for (int k = 0; k < NUM_STEPS_ROUND; ++k) {
        stepsArray[k] = pdByr * stepsArray[k + 1] + puByr * stepsArray[k];
    }
}
```

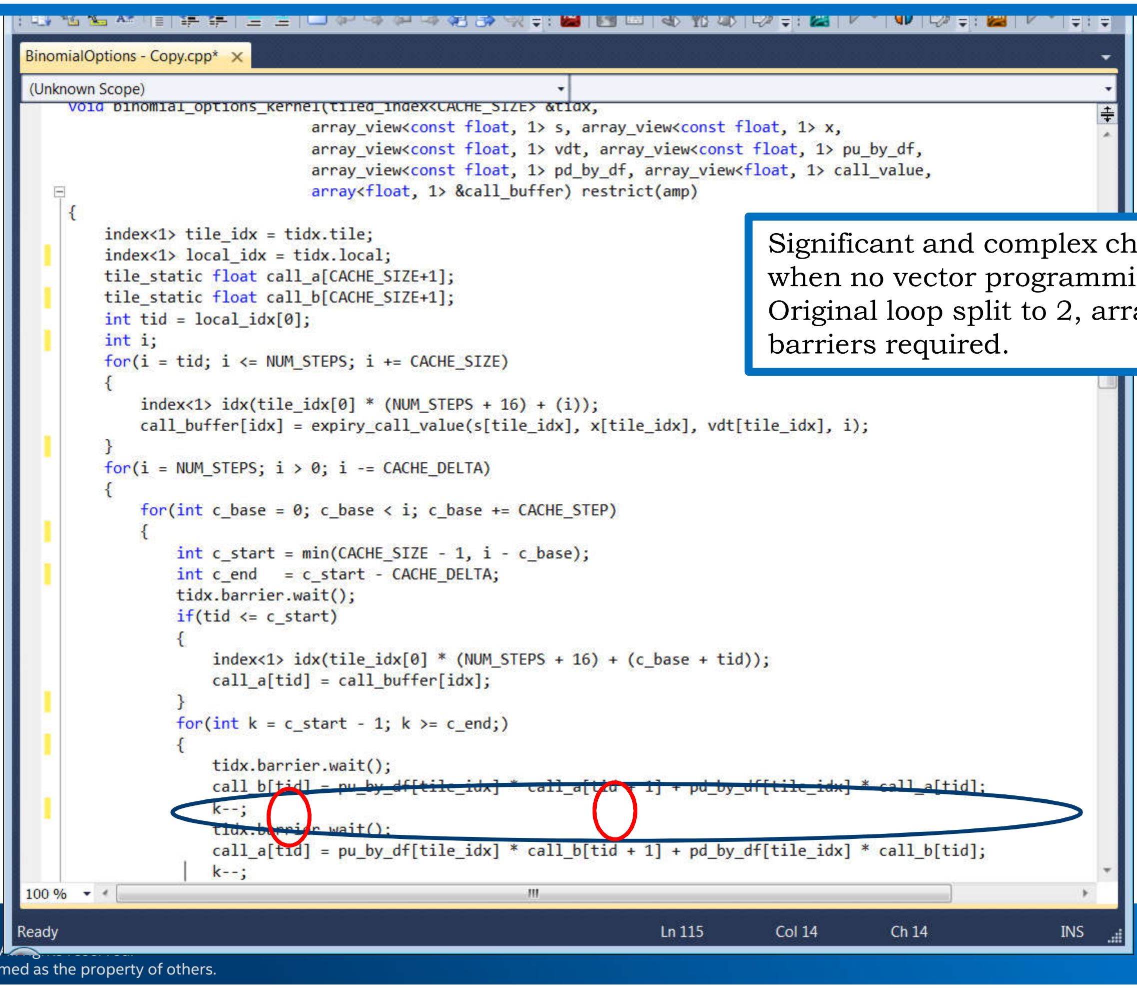
CUDA version

```
//Calculations within shared memory
for(int k = c_start - 1; k >= c_end;){
   //Compute discounted expected value
   __syncthreads();
   if(tid <= k)
      callB[tid] = puByDf * callA[tid + 1] + pdByDf * callA[tid];
   k--;
   //Compute discounted expected value
   __syncthreads();
   if(tid <= k)
      callA[tid] = puByDf * callB[tid + 1] + pdByDf * callB[tid];
   k--;
}
```

since the order of thread scheduling is complex and best viewed as simply undefined, the reduction primitive is double-buffered, ensuring by means of __syncthreads() that results from the previous stage are ready before they are used in the next,

Source: http://www.andrew.cmu.edu/user/dayoonc/binomialOptions.pdf

C++ AMP Sample Code for binomial options

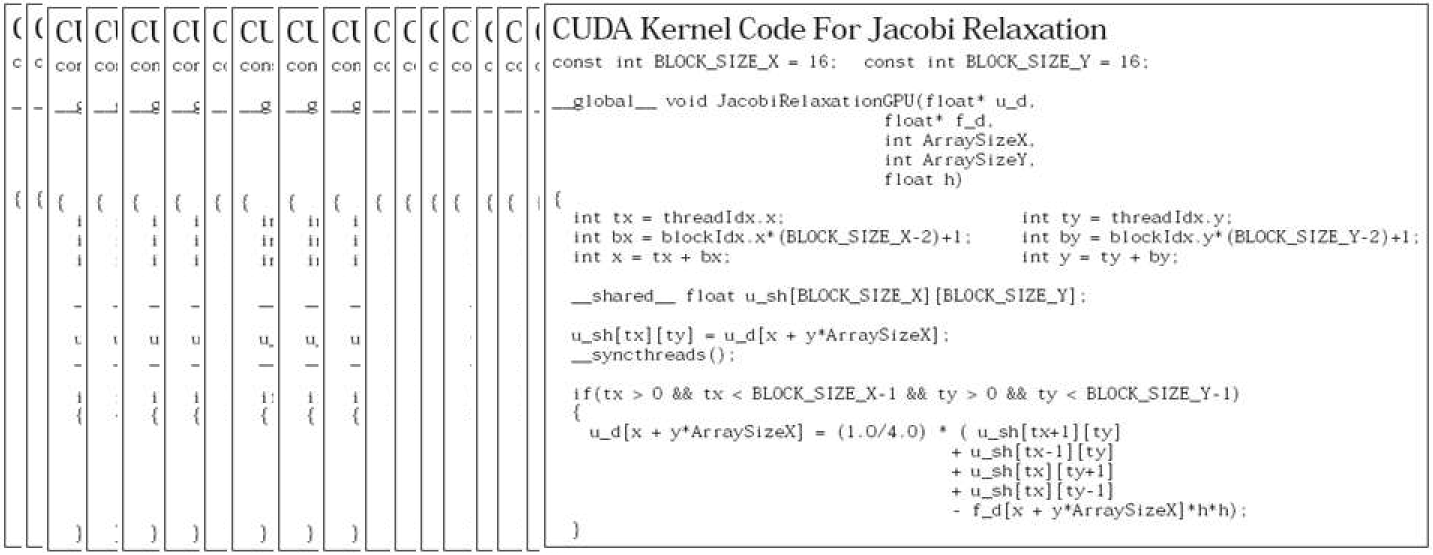


```
void binomial_options_kernel(tiled_index<CACHE_SIZE> &tidx,
                           array_view<const float, 1> s, array_view<const float, 1> x,
                           array_view<const float, 1> vdt, array_view<const float, 1> pu_by_df,
                           array_view<const float, 1> pd_by_df, array_view<float, 1> call_value,
                           array<float, 1> &call_buffer) restrict(amp)
{
    index<1> tile_idx = tidx.tile;
    index<1> local_idx = tidx.local;
    tile_static float call_a[CACHE_SIZE+1];
    tile_static float call_b[CACHE_SIZE+1];
    int tid = local_idx[0];
    int i;
    for(i = tid; i <= NUM_STEPS; i += CACHE_SIZE)
    {
        index<1> idx(tile_idx[0] * (NUM_STEPS + 16) + (i));
        call_buffer[idx] = expiry_call_value(s[tile_idx], x[tile_idx], vdt[tile_idx], i);
    }
    for(i = NUM_STEPS; i > 0; i -= CACHE_DELTA)
    {
        for(int c_base = 0; c_base < i; c_base += CACHE_STEP)
        {
            int c_start = min(CACHE_SIZE - 1, i - c_base);
            int c_end   = c_start - CACHE_DELTA;
            tidx.barrier.wait();
            if(tid <= c_start)
            {
                index<1> idx(tile_idx[0] * (NUM_STEPS + 16) + (c_base + tid));
                call_a[tid] = call_buffer[idx];
            }
            for(int k = c_start - 1; k >= c_end;)
            {
                tidx.barrier.wait();
                call_b[tid] = pu_by_df[tile_idx] * call_a[tid + 1] + pd_by_df[tile_idx] * call_a[tid];
                k--;
                tidx.barrier.wait();
                call_a[tid] = pu_by_df[tile_idx] * call_b[tid + 1] + pd_by_df[tile_idx] * call_b[tid];
                k--;
```

Significant and complex changes required when no vector programming available
Original loop split to 2, array split to 2, barriers required.

GPGPU: only kernels

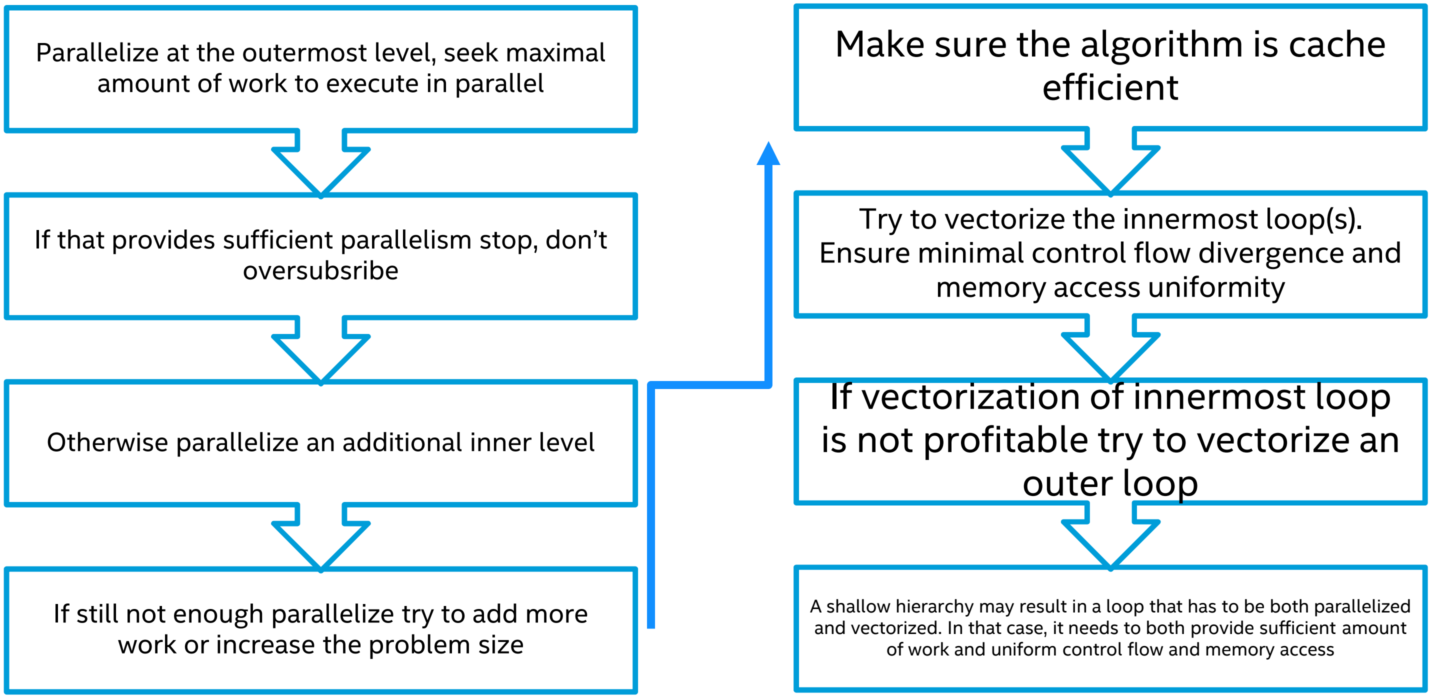
One-Size-Fits-All design pattern:
Write a kernel function, serial code
Then: Invoke many instances in parallel
(not minimizing that a lot of hard work still required, including tiling, etc)



CUDA Kernel Code For Jacobi Relaxation

```
const int BLOCK_SIZE_X = 16;   const int BLOCK_SIZE_Y = 16;

__global__ void JacobiRelaxationGPU(float* u_d,
                                    float* f_d,
                                    int ArraySizeX,
                                    int ArraySizeY,
                                    float h)
{
  int tx = threadIdx.x;                        int ty = threadIdx.y;
  int bx = blockIdx.x*(BLOCK_SIZE_X-2)+1;      int by = blockIdx.y*(BLOCK_SIZE_Y-2)+1;
  int x = tx + bx;                             int y = ty + by;

  __shared__ float u_sh[BLOCK_SIZE_X][BLOCK_SIZE_Y];

  u_sh[tx][ty] = u_d[x + y*ArraySizeX];
  __syncthreads();

  if(tx > 0 && tx < BLOCK_SIZE_X-1 && ty > 0 && ty < BLOCK_SIZE_Y-1)
  {
    u_d[x + y*ArraySizeX] = (1.0/4.0) * ( u_sh[tx+1][ty]
                                        + u_sh[tx-1][ty]
                                        + u_sh[tx][ty+1]
                                        + u_sh[tx][ty-1]
                                        - f_d[x + y*ArraySizeX]*h*h);
  }
```

#1 Best Practice in Parallelizing a Loop Hierarchy



Parallelize at the outermost level, seek maximal amount of work to execute in parallel

If that provides sufficient parallelism stop, don’t oversubsribe

Otherwise parallelize an additional inner level

If still not enough parallelize try to add more work or increase the problem size

Make sure the algorithm is cache efficient

Try to vectorize the innermost loop(s). Ensure minimal control flow divergence and memory access uniformity

If vectorization of innermost loop is not profitable try to vectorize an outer loop

A shallow hierarchy may result in a loop that has to be both parallelized and vectorized. In that case, it needs to both provide sufficient amount of work and uniform control flow and memory access

Vectorize Innermost, Parallelize Outermost (VIPO)

Methodology

Write a few algorithms in two ways:

- Parallelize and vectorize the outer loop – the kernel pattern
- Parallelize the outer loop and vectorize the inner loop – VIPO

Express both patterns in OpenMP® 4.0 and C++

Use the same compiler – ICC

Use the same Hardware, OS, etc

The only difference – the parallelization and vectorization pattern

Compare performance

Skeleton of sequential Binomial Code

```
Binomial()
{
   __declspec(align(1024)) REAL Call[NUM_STEPS + 1];
  //Forward Pass
   for (int i = 0; i <= NUM_Nodes; i++)
  {
    double d = Sx * Exp(vDt * (2.0f* i - NUM_STEPS)) - Xx;
     Call[i] = (d > 0) ? d : 0;
  }
  //Backward pass
  for(int i = NUM_STEPS; i > 0; i--)
  {
     int Num_Nodes = i-1;
     for(int j = 0; j <= Num_Nodes; j++)
          Call[j] = puByDf * Call[j + 1] + pdByDf * Call[j];
  }
}
main()
{
   #pragma omp parallel for
   for (int i=0; i<Nopt; i++)
         Binomial();
}
```

Nopt = 131072 and NUM_STEPS = 1024

Binomial Code Comparison

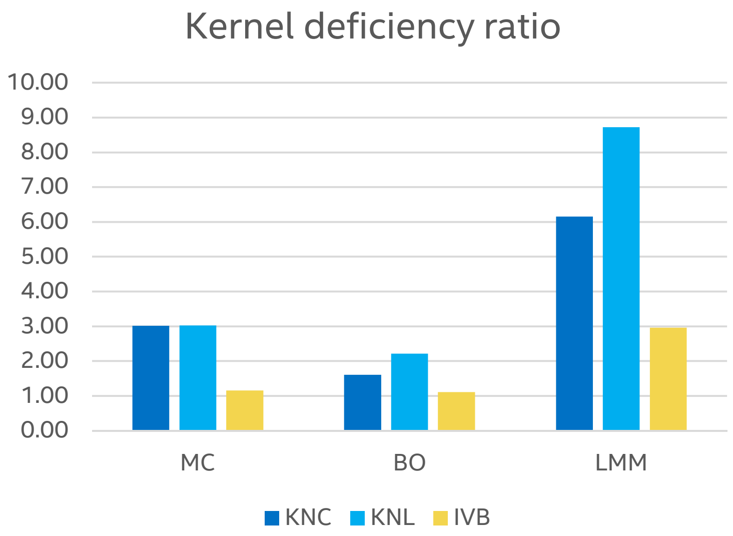
```
__attribute__((vector(vectorlength(DIMNSZ)))
Binomial(…..)
{
   __declspec(align(1024)) double call[NUM_STEPS + 1];
   for (int i = 0; i <= NUM_Nodes; i++) {
       double d = sx * exp(t * (2.0f * i - NUM_STEPS)) - xx;
       call[i] = (d > 0) ? d : 0;
  }
  for(int i = NUM_STEPS; i > 0; i--) {
       int Num_Nodes = i-1;
       for(int j = 0; j <= Num_Nodes; j++)
          call[j] = puByDf * Call[j + 1] + pdByDf * call[j];
   }
}
main()
{
   #pragma omp parallel for
   for (int n=0; n<Nopt; n+= DIMNSZ)
   {
      …….
      #pragma simd
      for (int i = 0; i < DIMNSZ; ++i) {
         Binomial(… );
    }
}
```

```
Binomial(…..)
{
   __declspec(align(1024)) REAL Call[NUM_STEPS + 1];
  #pragma simd
  for (int i = 0; i <= NUM_Nodes; i++) {
    double d = sx * exp(t * (2.0f * i - NUM_STEPS)) - xx;
     Call[i] = (d > 0) ? d : 0;
  }
  for(int i = NUM_STEPS; i > 0; i--) {
      int Num_Nodes = i-1;
      #pragma simd
      for(int j = 0; j <= Num_Nodes; j++)
         call[j] = puByDf * call[j + 1] + pdByDf * call[j];
   }
}

main()
{
   #pragma omp parallel for
   for (int i=0; i<Nopt; i++)
         Binomial(….);
}
```

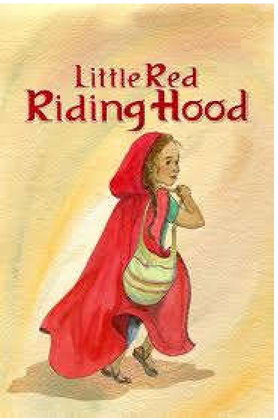
Monte Carlo Pseudo Code Pattern

```
for(opt=0; opt< NumOptions ; opt++)
{
   for(path=0; path<NumPaths; path++)
   {
        for (ts=1; ts<NumSteps; ts++)
        {
            S[ts] = S[ts-1] *exp(…)
        }
    }
```

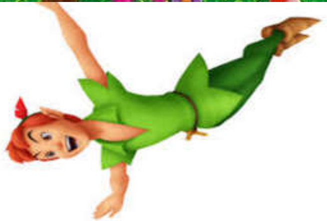
Results



MC	reference	Kernel		loops		KDR
		time	speedup	time	speedup	
KNC	4.19	2.59	1.62	0.86	4.87	3.01
KNL	3.99	2.48	1.61	0.82	4.87	3.02
IVB	3.35	1.23	2.72	1.06	3.16	1.16
BO	reference	Kernel		loops		
		time	speedup	time	speedup	
KNC	0.92	1.37	0.67	0.85	1.08	1.61
KNL	0.36	0.79	0.46	0.36	1.01	2.22
IVB	0.91	1.00	0.91	0.90	1.01	1.11
LMM	reference	kernel		loops		
		time	speedup	time	speedup	
KNC	2223.35	2340.60	0.95	380.10	5.85	6.16
KNL	882.00	898.00	0.98	102.90	8.57	8.73
IVB	1302.29	1414.00	0.92	478.10	2.72	2.96

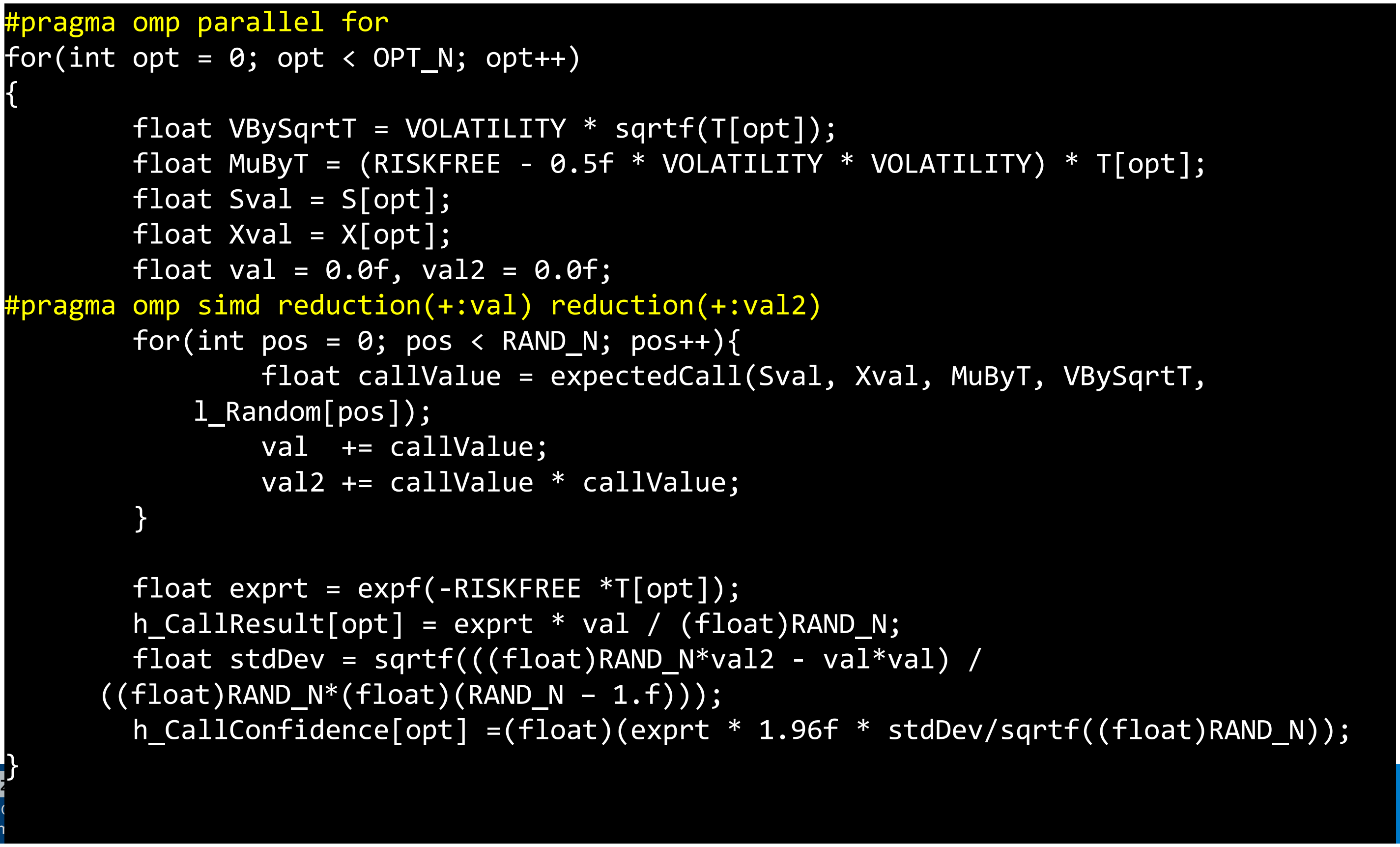


Reality: writing the code preferred by CPU is not possible for GPU. The HW doesn't support it and the languaes do not provide syntax for it. Writing the GPU preferred code for CPU is possible, with significant performance loss



A proposal for c++

The OpenMP syntax – a good solutions for loops



```
#pragma omp parallel for
for(int opt = 0; opt < OPT_N; opt++)
{
        float VBySqrtT = VOLATILITY * sqrtf(T[opt]);
        float MuByT = (RISKFREE - 0.5f * VOLATILITY * VOLATILITY) * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0f, val2 = 0.0f;
#pragma omp simd reduction(+:val) reduction(+:val2)
        for(int pos = 0; pos < RAND_N; pos++){
                float callValue = expectedCall(Sval, Xval, MuByT, VBySqrtT,
            l_Random[pos]);
                val  += callValue;
                val2 += callValue * callValue;
        }

        float exprt = expf(-RISKFREE *T[opt]);
        h_CallResult[opt] = exprt * val / (float)RAND_N;
        float stdDev = sqrtf(((float)RAND_N*val2 - val*val) /
      ((float)RAND_N*(float)(RAND_N – 1.f)));
        h_CallConfidence[opt] =(float)(exprt * 1.96f * stdDev/sqrtf((float)RAND_N));
}
```

Not so much for standard algorithms

```
Std::transform(inp.begin, inp.begin + inp.size(), out.begin(), ptr_fun<double, double>(sqrt));
```

```
void my_tranform(std::vector<int>& src, std::vector<int>& dst, std::function< int(int) > _func) {
    vec::transform(src.begin(), src.end(), dst.begin(), _func);
}
```

The Parallelism TS in C++

```
// Serial sort
std::sort(std::seq, x.begin(), x.end());

// Parallel sort
std::sort(std::par, x.begin(), x.end());

// Dynamically-selected policy
std::execution_policy e = std::seq();
if( x.size()>1024 )
    e = std::par();
std::sort(e, x.begin(), x.end());

std::transform(std::par, b, e, o, ptr_fun(<double,
double>(sqrt));
```

Many STL algorithms are in the proposal:

copy, transform, replace, generate, for_each, all_of, copy_if, find_if, is_sorted, inner_product, remove_if, rotate, binary_search …

These will help with parallelization,
Not with vectorization.

Indexed loops

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

OpenMP Equivalent

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

First, we proposed Indexed loops, even for the parallel execution policy

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];
}
```

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

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;
}
```

Extensions to the vec Policy

```
struct my_policy: vector_execution_policy {
    static const int safelen = 8;
    static const bool vectorize_remainder = true;
};
```

```
for_loop( my_policy(), 0, 1912, [&](int i) {
    Z[i+8] = Z[i]*A;
});
```

OpenMP Equivalent
(without vectorize_remainder)

```
#pragma omp simd safelen(8)
for( int i=0; i<1912; ++i ) {
    Z[i+8] = Z[i]*A;
});
```

Both parallel and vector

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

OpenMP Equivalent

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

Undefined behavior if there is a data race
Undefined behavior if there is a critical section (deadlock)
The loop is both parallelized and vectorized
Same semantics as a “kernel” in GPGPU languages

Less trivial vectorizeable algorithms

Algorithms with certain dependence patterns
do not prevent vectorization of enclosing algorithms

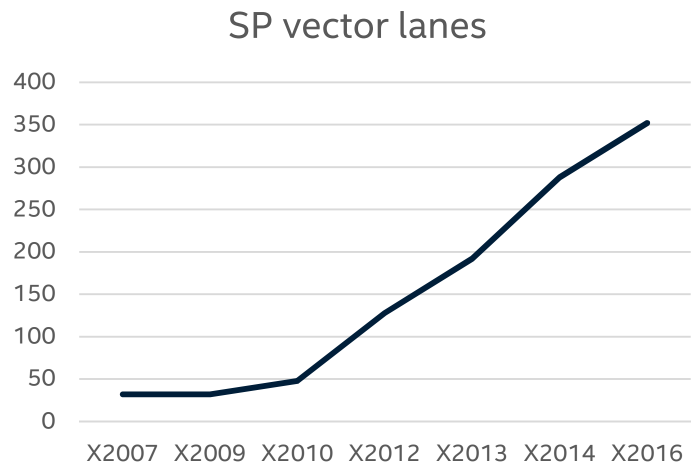
And depending on the target architecture

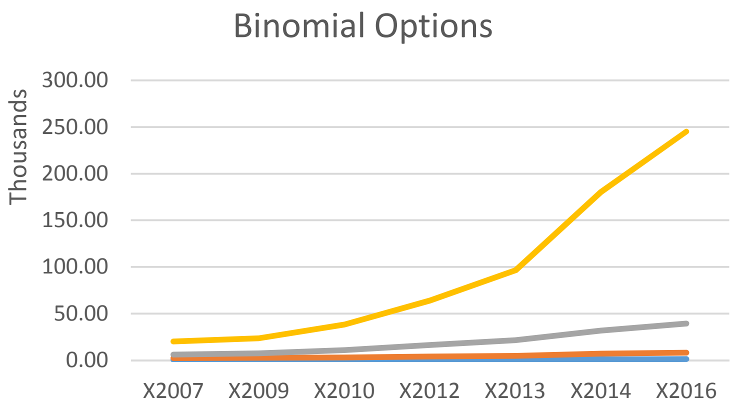
May themselves be vectorized (may or may not be profitable)

Account for future direction of SIMD HW: these are made possible by AVX512

```
// Histogram
a[b[i]]++;
```

```
// compress / expand
if (cond(i)) {
  a[i] = b[i] * c[j++];
}
```

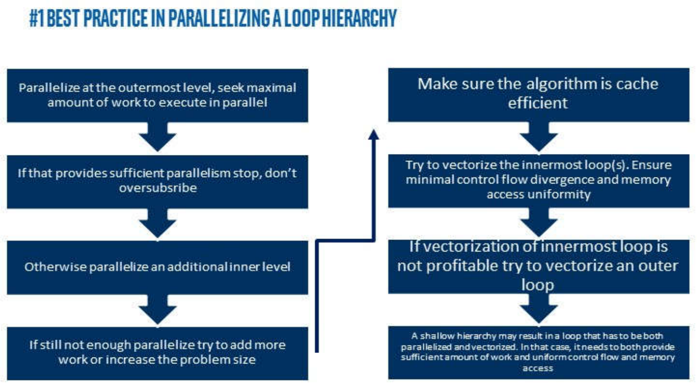




(1) Incremental growth in CPU resources

(2) Improvements in compilers and parallel frameworks

(3) Parallelization techniques



#1 BEST PRACTICE IN PARALLELIZING A LOOP HIERARCHY

Parallelize at the outermost level, seek maximal amount of work to execute in parallel

If that provides sufficient parallelism stop, don't oversubsribe

Otherwise parallelize an additional inner level

If still not enough parallelize try to add more work or increase the problem size

Make sure the algorithm is cache efficient

Try to vectorize the innermost loop(s). Ensure minimal control flow divergence and memory access uniformity

If vectorization of innermost loop is not profitable try to vectorize an outer loop

A shallow hierarchy may result in a loop that has to be both parallelized and vectorized. In that case, it needs to both provide sufficient amount of work and uniform control flow and memory access

Vectorize Innermost, Parallelize Outermost (VIPO)

Optimization Notice

intel®

experience
what's inside™

Configuration

Hardware configuration

- Processor:
  - Intel(R) Xeon(R) CPU E5-2697 @ 2.7 GHz (IVT)
  - 2 sockets/24 cores/48 Threads; Turbo Off
- Memory:
  - 128GB @ 1600 MHz

Software Configuration

- OS: Linux: RHEL 6.1
- Compiler:
  - Gcc 4.8
  - Intel Composer XE 2013 Sp1

Swap Pricer:

- Default : Number of scenarios 100
- Number of Swaps 100,000: parallelize at this level
- 26 time steps

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.

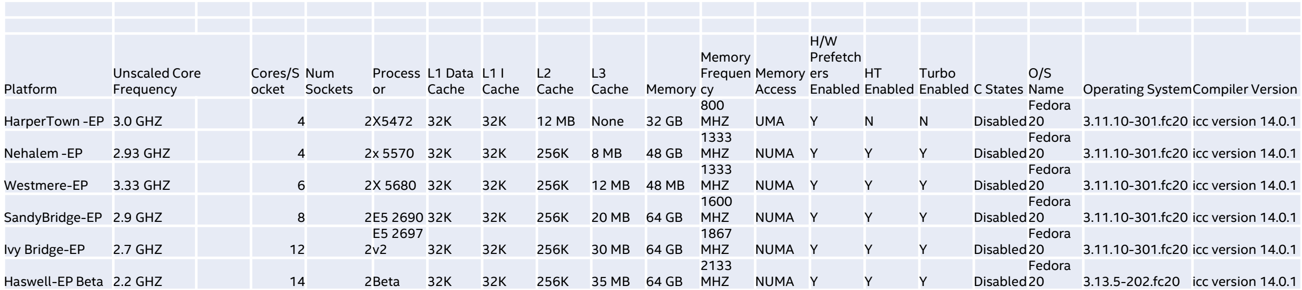
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Configuration for parallel speed-up

Platform Hardware and Software Configuration



Platform	Unscaled Core Frequency		Cores/Socket	Num Sockets	Processor	L1 Data Cache	L1 I Cache	L2 Cache	L3 Cache	Memory	Memory Frequency	Memory Access	H/W Prefetchers Enabled	HT Enabled	Turbo Enabled	C States	O/S Name	Operating System	Compiler Version
HarperTown -EP	3.0 GHZ		4	2	X5472	32K	32K	12 MB	None	32 GB	800 MHZ	UMA	Y	N	N	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Nehalem -EP	2.93 GHZ		4	2	x 5570	32K	32K	256K	8 MB	48 GB	1333 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Westmere-EP	3.33 GHZ		6	2	X 5680	32K	32K	256K	12 MB	48 MB	1333 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
SandyBridge-EP	2.9 GHZ		8	2	E5 2690	32K	32K	256K	20 MB	64 GB	1600 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Ivy Bridge-EP	2.7 GHZ		12	2	E5 2697 v2	32K	32K	256K	30 MB	64 GB	1867 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Haswell-EP Beta	2.2 GHZ		14	2	Beta	32K	32K	256K	35 MB	64 GB	2133 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.13.5-202.fc20	icc version 14.0.1

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®