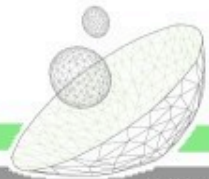
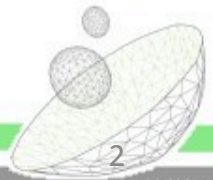


Performance Out of the Box On Multicore and Manycore Hardware with **Code Modernization**

Embrace Micro-Architecture Changes
Abstract-Out Instruction Set Variety
Achieve State-Of-The-Art Performance



PROCESSORS AND MOORE'S LAW

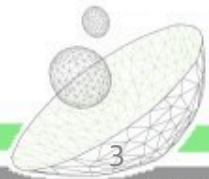


Processors Evolution

- Processors have changed

year	2000	2016	2016	2016	2014
processor	<i>Pentium 4</i>	Xeon E5-v4	KNL	Pascal	Power 8
core frequency (GHz)	3,8	2,2	1,5	1,478	4,116
vector unit size (DP)	1	4	8	32	2
pipelines / core	1	2	2	1	2
contexts	1	2	4	32	8
core count	1	22	72	56	12
FMA	1	2	2	2	2
Peak scalar GFLOPS	3,8	194	432	166	99
Peak GFLOPS (DP)	3,8	774	3456	5300	395
SIMD/SIMT ratio	1	4	8	32	4
Bandwidth (R/W)	4,26	76,8	512	720	128 / 64
flop / memop	7,14	80,67	54	58,9	24,7 / 49,4
Bandwidth / core	4,26	3,49	7,11	12,86	10,67 / 5,33

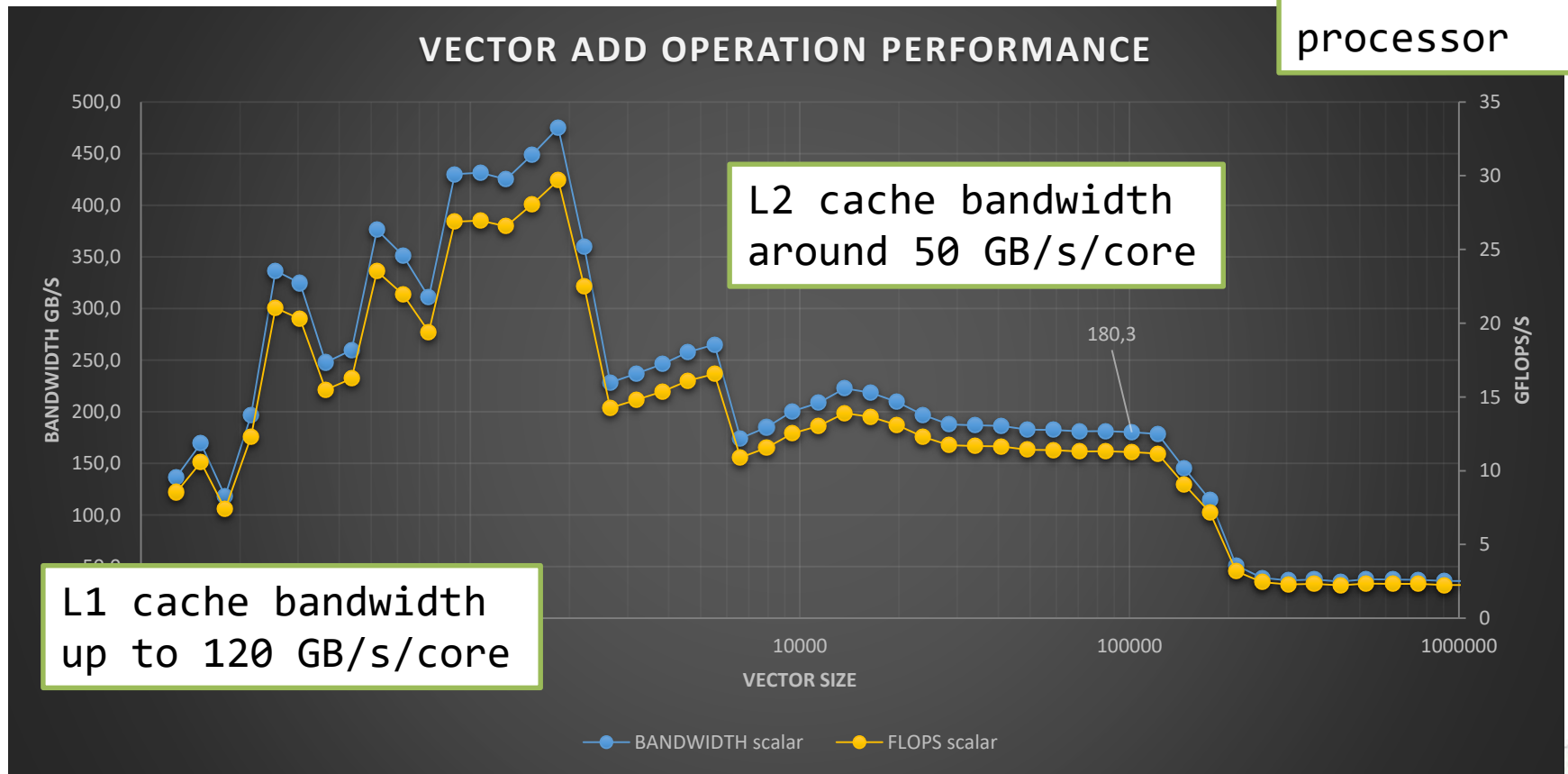
- Frequency drops, Core count / vector unit explodes
- Most problems get memory bound (flop / memop \sim 50)
- Multithreading is not the only issue (SIMD/SIMT ratio)
- Keeping-up with technology changes requires significant software development effort and training



Processor Evolution

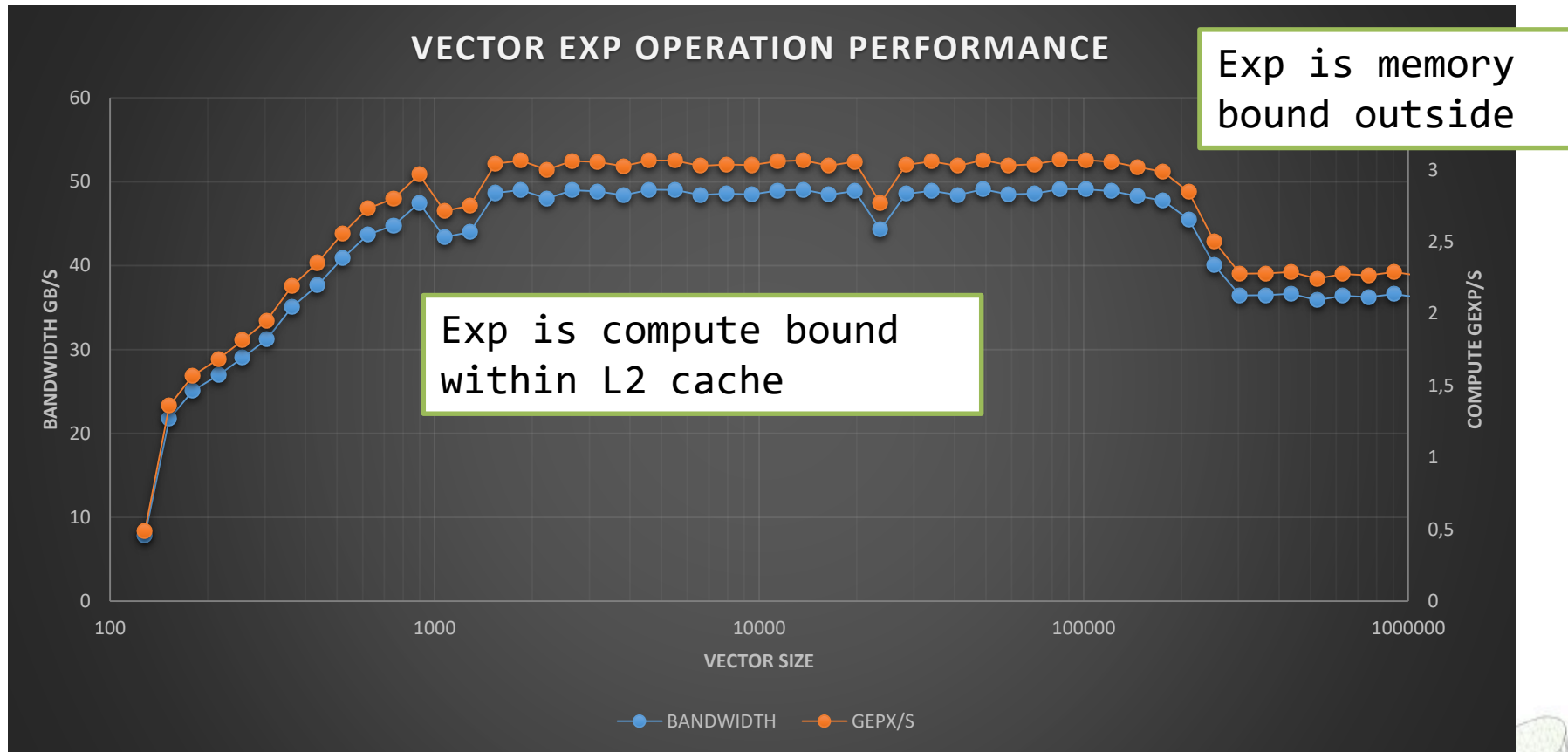
- Adding a constant to a vector of values

Peak is 112 for this processor



Processor Evolution

- Getting exp of a vector



Xeon E5-1620v4 – flop/memop = 23

Moore's Law FLOPS And BANDWIDTH

FLOPS double
every

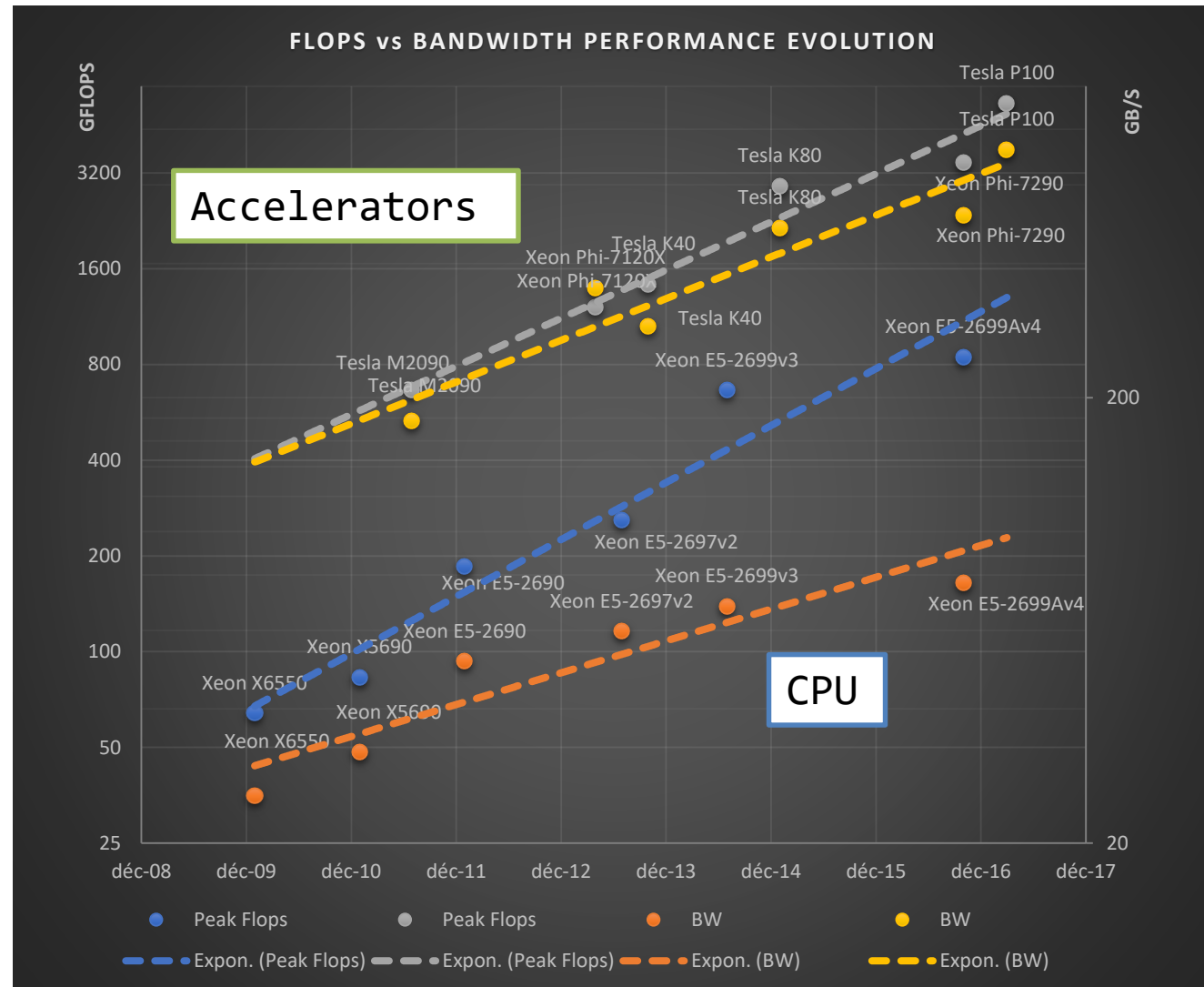
CPU: 1.8y

ACC: 1.9y

BANDWIDTH double
every

CPU: 4.3y

ACC: 2.8y



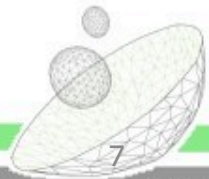
Moore's Law: Initial Understanding is Over

- Initial understanding:
 - **With no effort**, the performances of my software will **double every period**
 - As a result, I don't need to make optimization effort in my source code asset to get better performance with next generation processor

THIS IS OVER

- Moore's Law still plays:
 - Performance per dollar doubles every period
 - **BUT**: making use of performance requires significant effort
 - Improvements obtained with **vector units and multithreading**

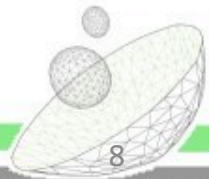
NEED SOFTWARE ASSET UPDATES FOR PERFORMANCE



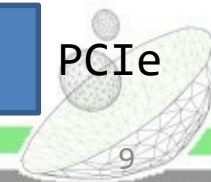
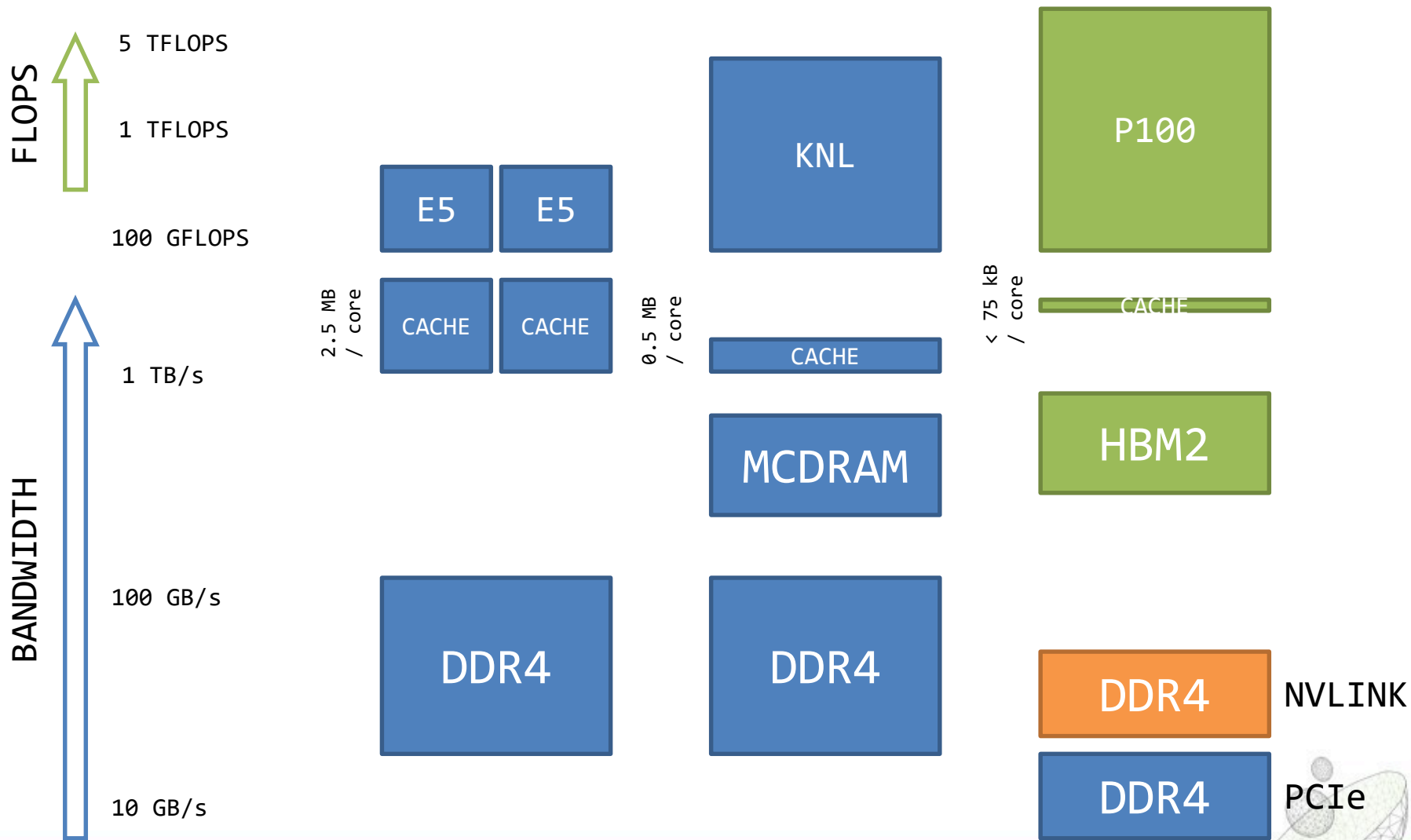
Programmers Perspective

- A processor has
 - Vector units
 - Simultaneous contexts
 - Memory layers

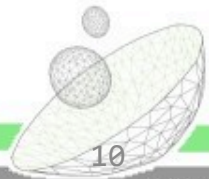
	2S Broadwell	Intel KNL	NVIDIA Pascal
vector unit size	4	8	32
registers	16	32	32 / 256
contexts per core	2	4	64 / 8
cores	44	72	56
L1 cache size (per core)	16 kB	16 kB	64 kB (++)
LLC cache size (per core)	2.5 MB	0.5 MB	0.073 MB
Bandwidth per core	3.5 GB/s	7.1 GB/s	12.9 GB/s



Programmers Perspective



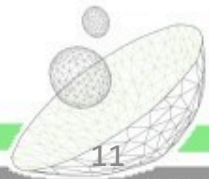
CODE MODERNIZATION



Compilers Need Help

- Pointer aliasing

```
void func (int n, double* a, double* b)
{
    for (int k = 0 ; k < n ; ++k)
    {
        a [k] += b[k] ;
    }
}
```

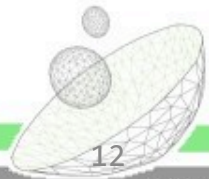


Compilers Need Help

- Pointer aliasing

```
void func (int n, double* a, double* b)
{
    for (int k = 0 ; k < n ; ++k)
    {
        a [k] += b[k] ;
    }
}
```

```
int main(...)
{
    func (n, a, a - 1);
}
```



Compilers Need Help

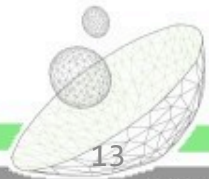
- Pointer aliasing

```
void func (int n, double* a, double* b)
{
    for (int k = 0 ; k < n ; ++k)
    {
        a [k] += b[k] ;
    }
}
```

```
int main(...)
{
    func (n, a, a - 1);
}
```

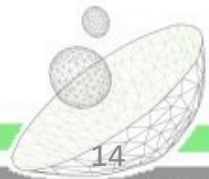
- Previous optimizations cause troubles

```
void func(int n, double* a, double* b)
{
    for (int k = 0; k < n; ++k)
    {
        *(a++) += *(b++);
    }
}
```



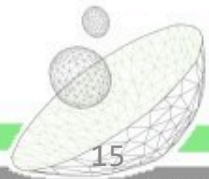
Compilers Need Help

- On Heuristics
 - Cannot always define what is behind a function call
 - Global / multifile optimization gets complicated
 - Heuristic calibrated on specific hardware
 - Cost of accessing memory cannot be defined at compile time
 - Accessing main memory is much larger than compute
- On Vectorization
 - Masked operations fairly recent
 - Compilers need updates to identify vectorization opportunities
 - Depending on input type (register spill or memory), vectorization opportunity might not be relevant



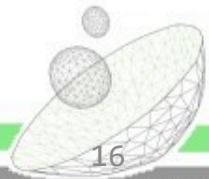
Be Explicit

- Two parallelization levels, very different
 - Vector-unit level
 - Small chunks, minimize branching, etc.
 - Thread level
 - Work will be distributed across Cores/threads
 - Join has a cost (microseconds)
- Loop ordering is important !
 - Two nested loops: vectorize inner.
 - If inner loop is not parallelizable, heuristic most often refuses to vectorize outer (see example).



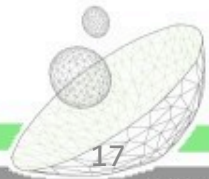
How to Express ?

- OpenMP
 - Directive-based – significant change in 4.x (target/SIMD)
- CUDA
 - NVIDIA environment to program GPUs – some x86 implementations
- HIP
 - AMD *clone* of CUDA
- Intel Threading Building Blocks™
 - Intel API to expose parallel and concurrent constructs
- OpenCL
 - Chronos group initiative – similar to CUDA in work distribution pattern
- OpenACC
 - Directive-based approach: similar yet different to OpenMP 4.x
- And more...



Decoupling language from performance

USING C# FOR HIGH THROUGHPUT COMPUTING



Language Compilers and Runtimes

Algorithmic optimization: e.g. difference between bubble and heap-sort

Languages

- allow developers to express algorithms to be run on computers

Compiler optimization: extract parallelism (with help)

Compilers

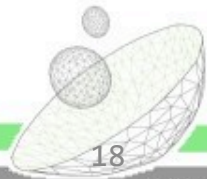
- transform source code expressed in a language into machine language

Runtime optimization: Use vendor-tuned libraries

Runtimes

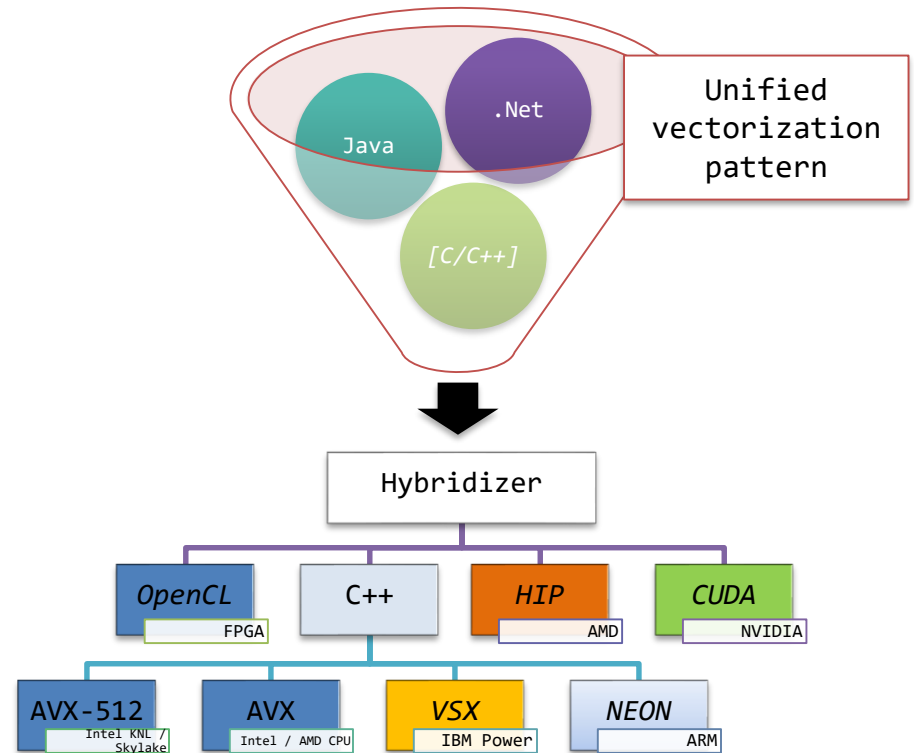
- are implementations of widely used methods and algorithms (e.g. BLAS)

Language choice should not impact performance

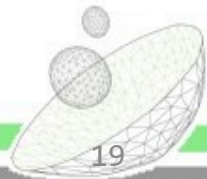


Hybridizer Solution

- Input
 - .Net
 - Java
 - [C/C++]
- Environments:
 - Windows / Linux
- Generate source code
 - **CUDA/C** for NVIDIA GPU
 - **C++ (vectorized)** for native platforms

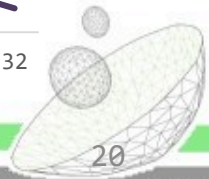
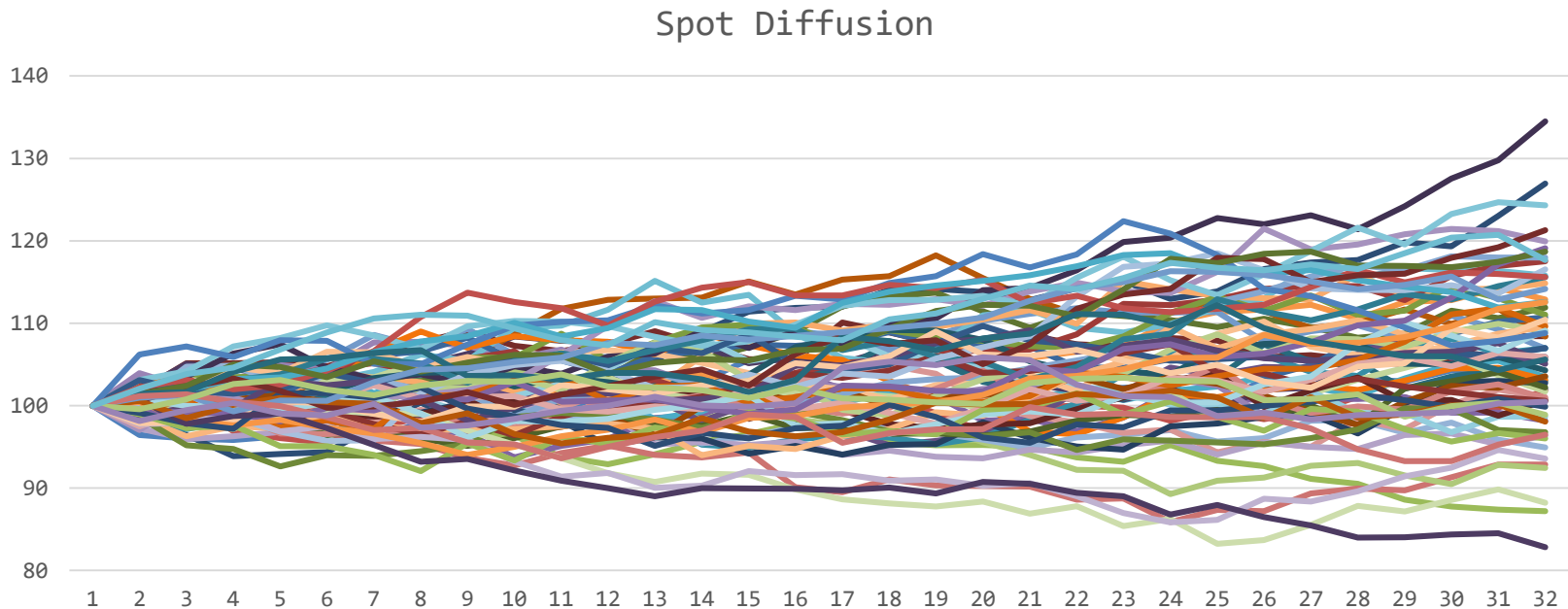


Compilation pipeline to replace dot net JIT



Example Walkthrough

- Financial Model Spot Diffusion
 - Fundamentals of Monte-Carlo simulations
 - Building block of complex models and products
 - Reference benchmark for FSI



Financial Model Spot Diffusion

- Implementation(s)
 - Two nested loops : over time and simulations

```
void diffusion (...)
```

```
{  
  for (int simId = 0 ; simId < simCount ; ++simId)  
    for (int time = 0; time < datesCount; ++time)  
    {  
      lnS[time+1, simId] = lnS[time, simId] + ...;  
    }  
}
```

Simulation Outer Loop

```
void diffusion (...)
```


```
{  
  for (int time = 0; time < datesCount; ++time)  
    for (int simId = 0 ; simId < simCount ; ++simId)  
    {  
      lnS[time+1, simId] = lnS[time, simId] + ...;  
    }  
}
```

Time Outer Loop

Financial Model Spot Diffusion

Dot net source code
Generic parameters for flexibility


```
[Kernel]
public void Diffusion(
    int simFrom, int simTo,
    int timeFrom, int timeTo,
    Volatility volatility,
    Rate rate,
    LogSpot logSpot,
    Brownian brownian,
    Schedule schedule)
{
    for (alignedindex simId = VectorUnit.ID + simFrom;
        simId < simTo; simId += VectorUnit.Count)
    {
        double lnSk = logSpot[simId, timeFrom];
        for (int t = timeFrom; t < timeTo; ++t)
        {
            double sigma = volatility[lnSk, simId, t];
            double sqrtdt = schedule.getSqrtDT(t);
            double dt = schedule.getDT(t);
            lnSk += (sigma * brownian[simId, t] * sqrtdt) +
                (rate[simId, t] - 0.5 * sigma * sigma) * dt;
            logSpot[simId, t + 1] = lnSk;
        }
    }
}
```



C++ source code with annotations
(two outer loop configurations)

```
void diffuse (int simCount, int datesCount,
    const double* __restrict dates,
    const double* __restrict DT,
    const double* __restrict sqrtDT,
    const double* __restrict brownian,
    double* __restrict logSpot,
    double sigma, double rate)
{
    #pragma omp parallel for
    #pragma simd
    #pragma ivdep
    for (int simId = 0 ; simId < simCount ; ++simId)
    {
        double lnS = logSpot [simId] ;
        for (int time = 0 ; time < datesCount ; ++time)
        {
            lnS += (sigma * brownian[time * simCount + simId] * sqrtDT[time]) +
                (rate - 0.5 * sigma * sigma) * DT[time];
            logSpot[(time+1) * simCount + simId] = lnS ;
        }
    }
}

#pragma omp parallel for
for (int th = 0 ; th < 8 ; ++th)
{
    int simFrom = th * simCount / 8 ;
    int simTo = (th+1) * simCount / 8 ;
    for (int time = 0 ; time < datesCount ; ++time)
    {
        double* lnS = logSpot + (simCount * time) ;
        const double* brow = brownian + (time * simCount) ;
        #pragma ivdep
        #pragma simd
        for (int simId = simFrom ; simId < simTo ; ++simId)
        {
            lnS[simId + simCount] = lnS [simId] + (sigma * brow[simId] * sqrtDT[time]) + (rate - 0.5 * sigma * sigma) * DT[time];
        }
    }
}
```



Black-Scholes-Merton Diffusion

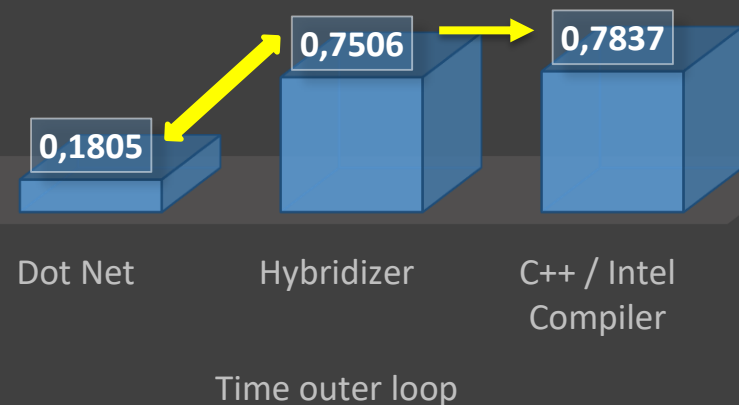
FINANCIAL MODEL SPOT DIFFUSION – GSTEPS/S

■ 16384 simulations (off-cache)

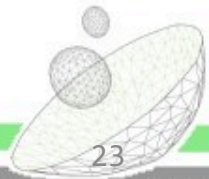
Memory-bound

Significant performance improvement over dotnet (x4)

Limited overhead compared to handwritten (4%)



- Comparing object-oriented code, with generics, processed by Hybridizer
- with hand-written optimized C++ code compiled with Intel Composer 2015

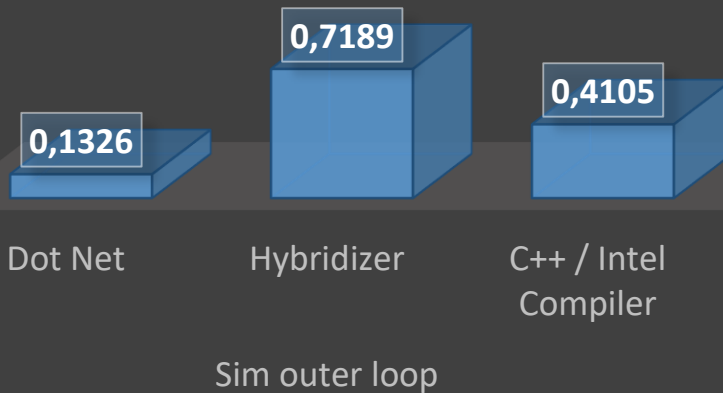


Black-Scholes-Merton Diffusion

FINANCIAL MODEL SPOT DIFFUSION – GSTEPS/S

■ 16384 simulations (off-cache)

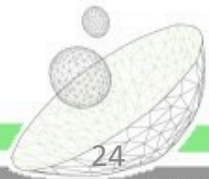
Memory-bound



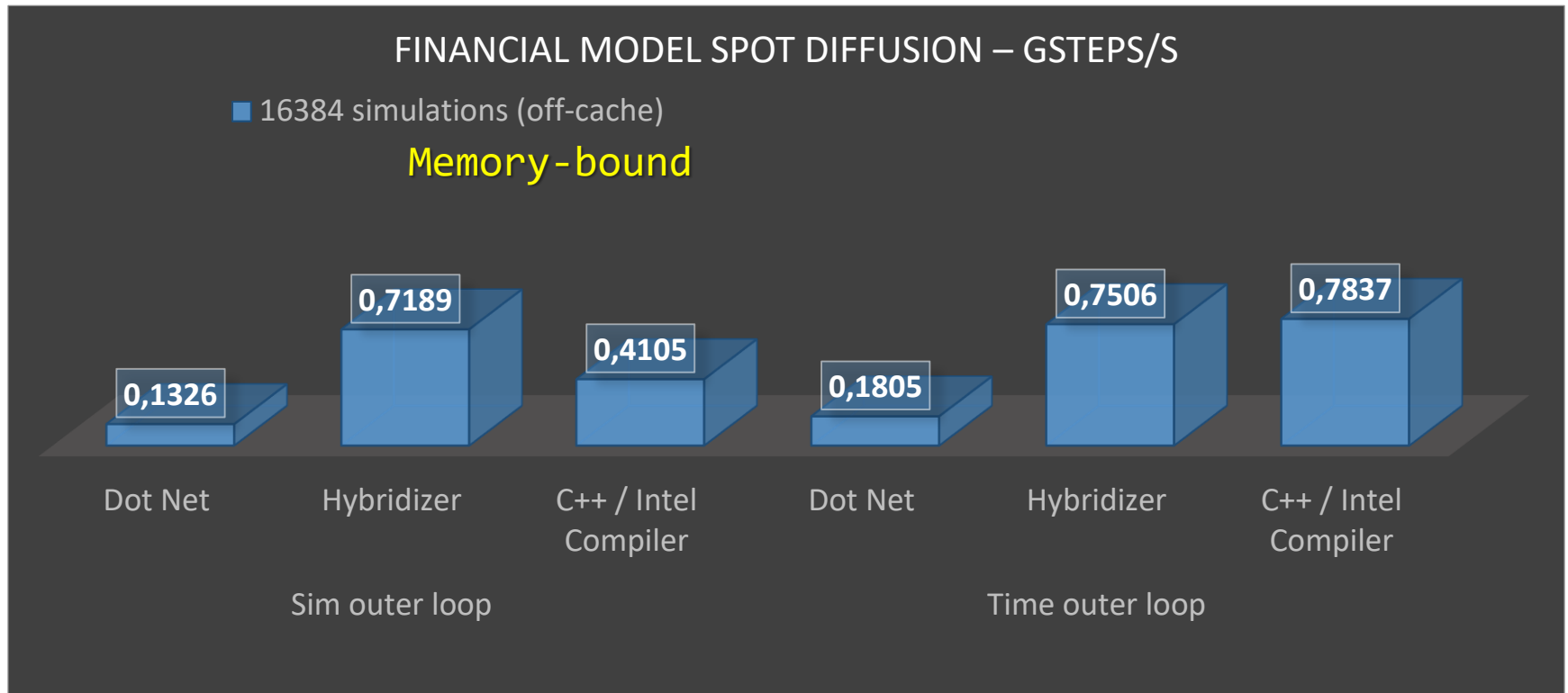
Loop ordering prevents some optimizations for Intel compiler:

Inner loop cannot be vectorized => none get

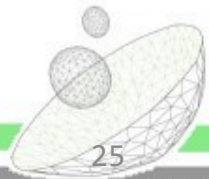
- Comparing object-oriented code, with generics, processed by Hybridizer
- with hand-written optimized C++ code compiled with Intel Composer 2015



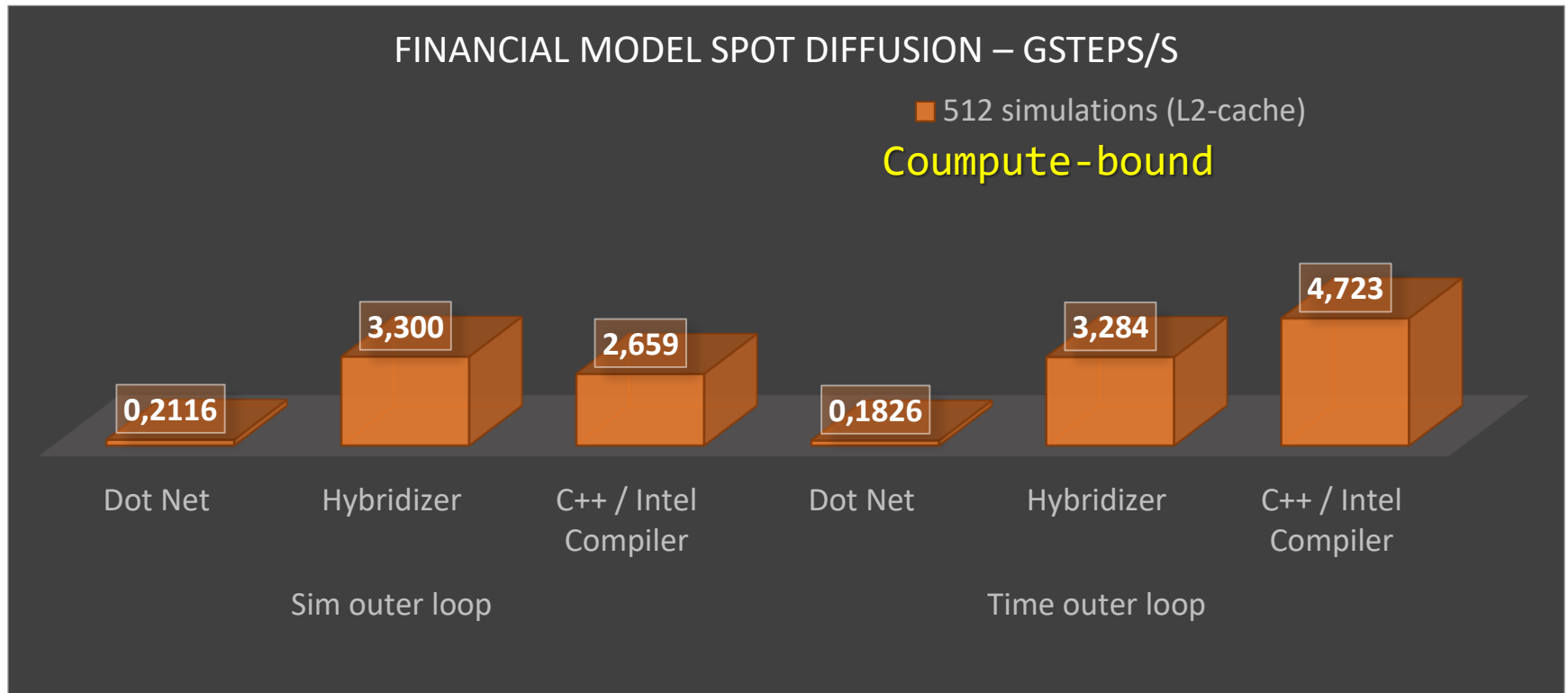
Black-Scholes-Merton Diffusion



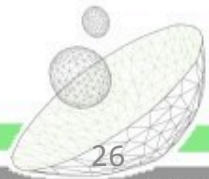
- Comparing object-oriented code, with generics, processed by Hybridizer
- with hand-written optimized C++ code compiled with Intel Composer 2015



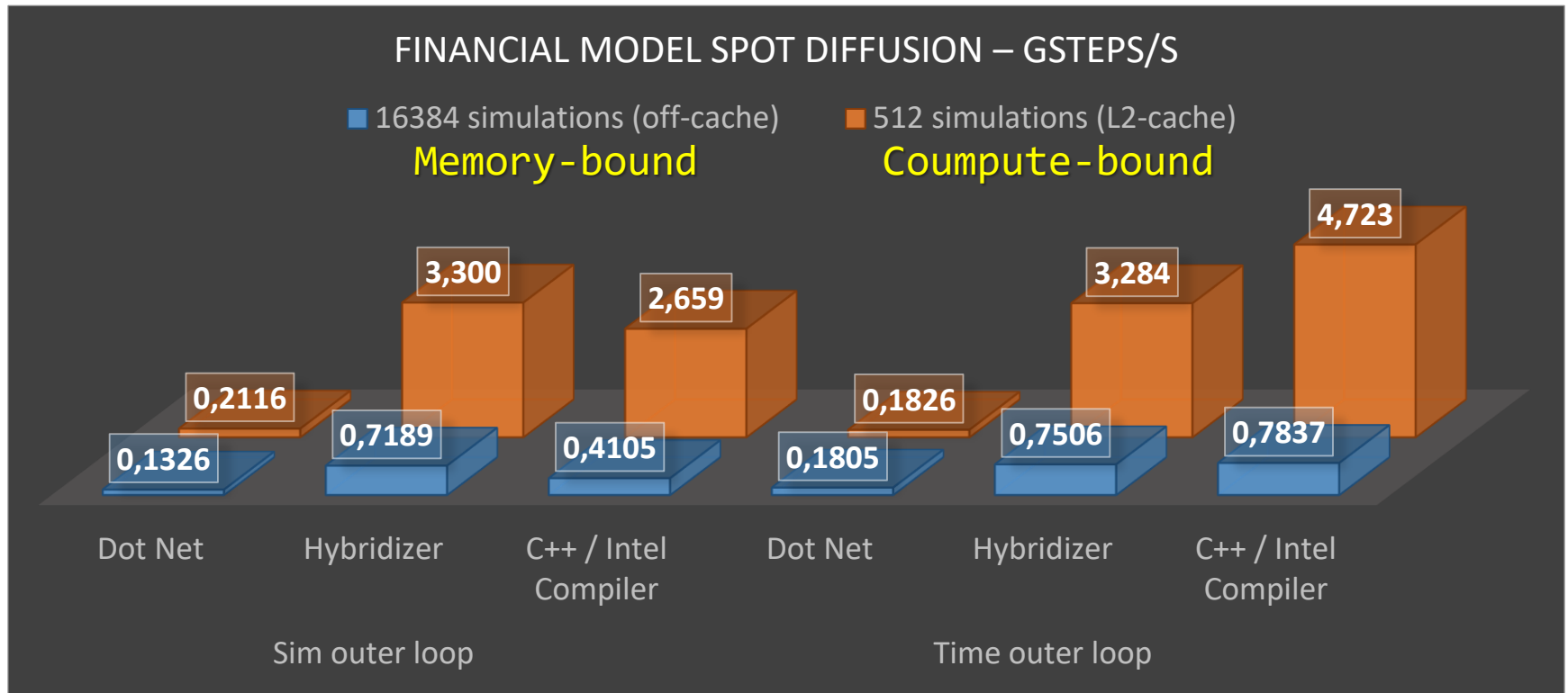
Black-Scholes-Merton Diffusion



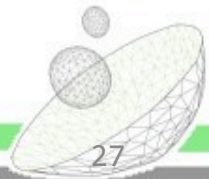
- Comparing object-oriented code, with generics, processed by Hybridizer
- with hand-written optimized C++ code compiled with Intel Composer 2015



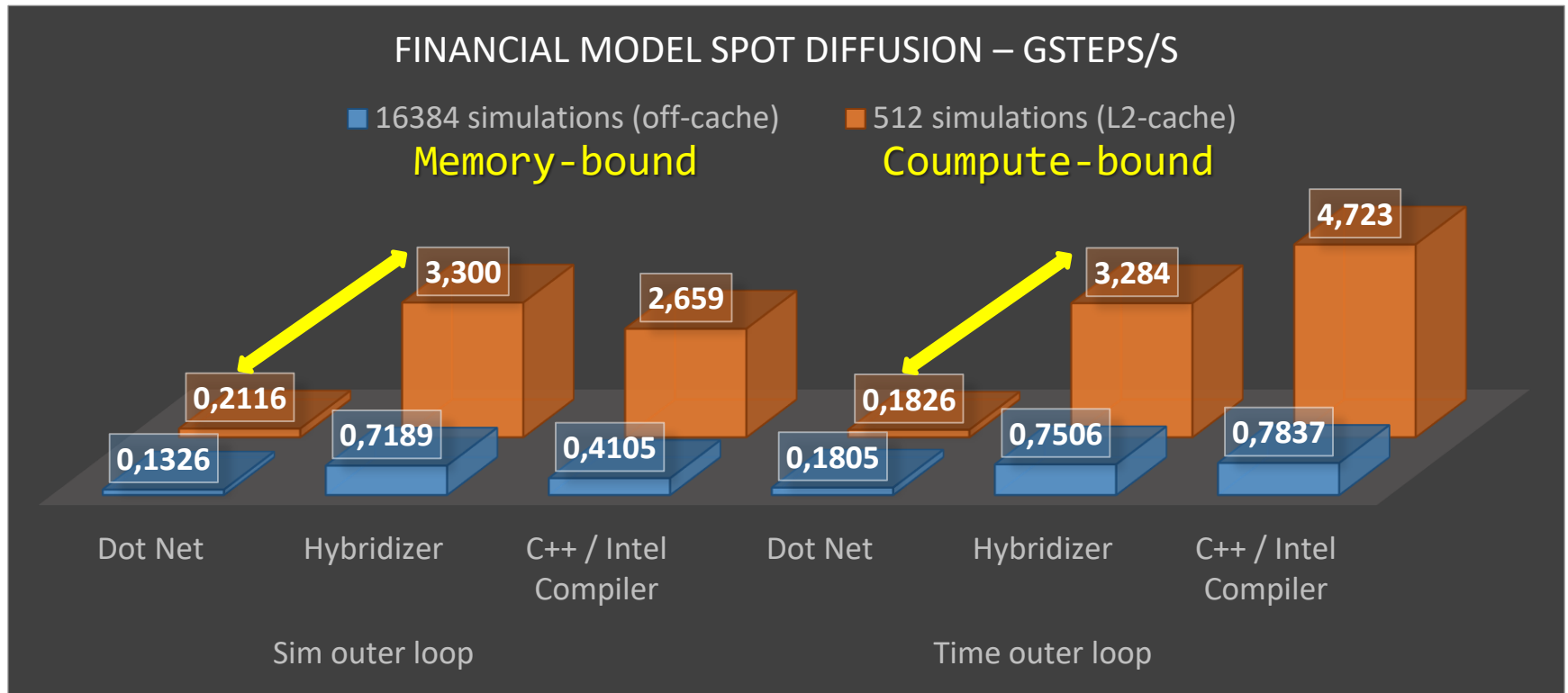
Black-Scholes-Merton Diffusion



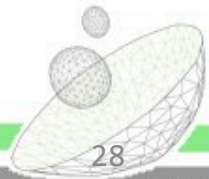
- Comparing object-oriented code, with generics, processed by Hybridizer
- with hand-written optimized C++ code compiled with Intel Composer 2015



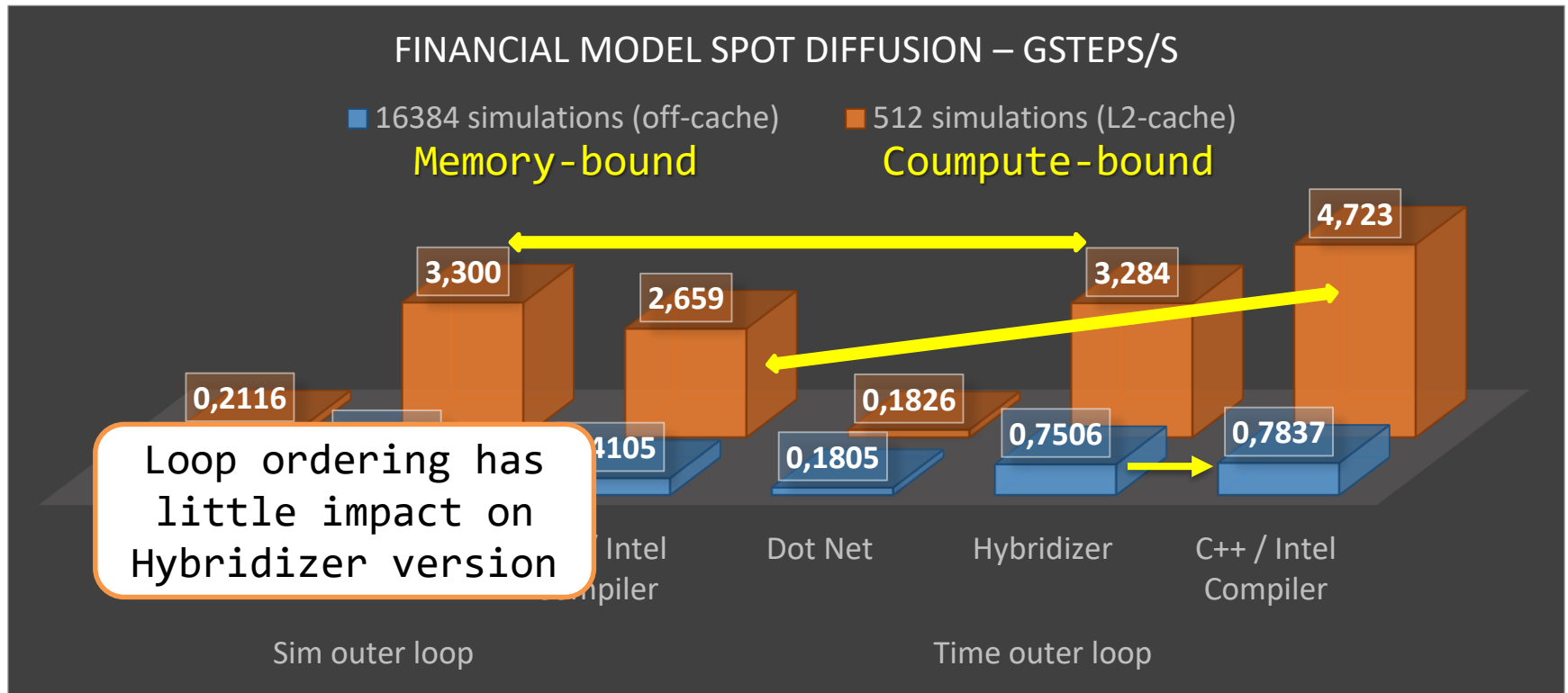
Black-Scholes-Merton Diffusion



- Hybridizer greatly improves dotnet performance: **5x to 18x**
- Object oriented programming preserved: single version of source code, reduces operational risk / testing costs.

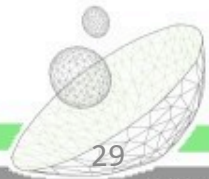


Black-Scholes-Merton Diffusion



- Hybridizer greatly improves dotnet performance: **5x to 18x**
- Object oriented programming preserved: single version of source code, reduces operational risk / testing costs.

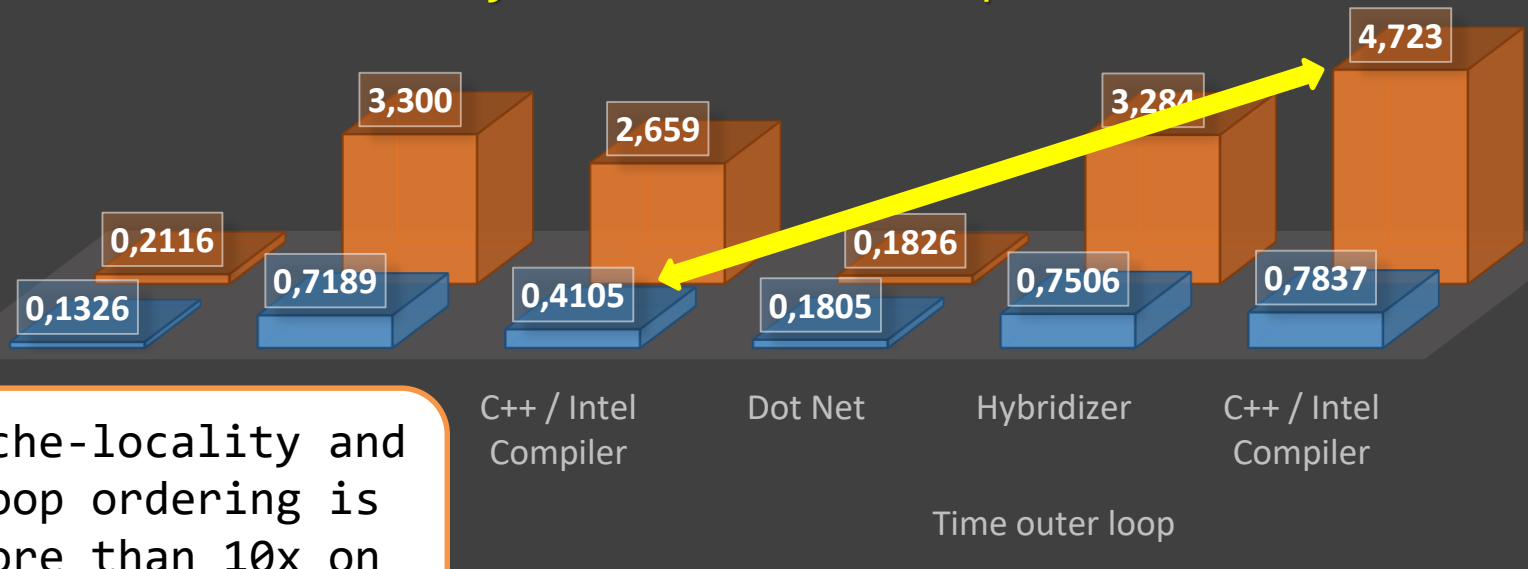
- Hybridizer provides benchmark-level performances (**96% of best performing off-cache**)
- Loop ordering has little impact for Hybridizer version (~4%) yet large impact for hand-written implementation (>45%)



Black-Scholes-Merton Diffusion

FINANCIAL MODEL SPOT DIFFUSION – GSTEPS/S

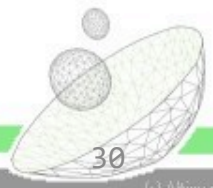
■ 16384 simulations (off-cache) **Memory-bound**
 ■ 512 simulations (L2-cache) **Compute-bound**



Cache-locality and Loop ordering is more than 10x on performance !

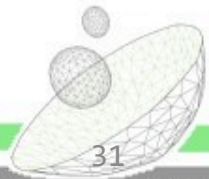
- Object oriented programming preserved: single version of source code, reduces operational risk / testing costs.

- Hybridizer provides benchmark-level performances (96% of best performing off-cache)
- Loop ordering has little impact for Hybridizer version (~4%) yet large impact for hand-written implementation (>45%)



Hybridizer Benefits

- **Single version of source code**
 - Express parallelism with a paradigm of choice (ParallelFor / iterators / custom indexing type)
 - Generates several flavors of source code
- **Execution on a variety of platforms**
 - Plain C, CUDA
 - Vector-units: AVX, AVX2, AVX-512
 - External libraries integration (e.g. MKL) and extensibility (hand-tuned micro-architecture specific codes)
- **Debugging / Profiling of output**
 - Code location is preserved on target platform
 - Integration in existing debugging / profiling tools
 - Generated source-code is readable for auditing



Integration with Intel Vtune Amplifier

Standard System.Math methods

```
public static double CND(double d)
{
    double K = 1.0 / (1.0 + 0.2316419 * Math.Abs(d));
    double cnd = RSQR2PI *
        Math.Exp(-0.5 * d * d)
        * (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));
    if (d > 0)
        cnd = 1.0 - cnd;
    return cnd;
}
```

Scalar C# source File

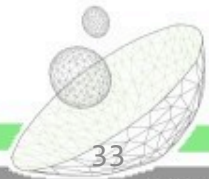
```
public static void CallPut(int index, double[] call, double[] put, double spot,
{
    double sigmaSqrtT = sigma * Math.Sqrt(maturity);
    double d1 = (Math.Log(spot / strike) + (rate + sigma * sigma / 2.0) * maturi
    double d2 = d1 - sigmaSqrtT;
    double kert = strike * Math.Exp(-rate * maturity);
    double CNDd1 = CND(d1);
    double CNDd2 = CND(d2);
    call = spot * CNDd1 - kert * CNDd2;
    put = kert * (1.0 - CNDd2) - spot * (1.0 - CNDd1);
}
```

Mapped on Intel SVML

AVX2 instructions

Address	Source	Assembly	CPU Time: Total	CPU Time: Self
0x18000591b	Block 46:			
0x18000591b	115	vmlpdd ymm4, ymm0, ymmword ptr [r13+0xc20]	0.002s	0.002s
0x180005924	115	vmlpdd ymm0, ymm7, ymmword ptr [r13+0xba0]	0.095s	0.095s
0x18000592d	115	vmovupd ymmword ptr [r13+0x500], ymm4	0.001s	0.001s
0x180005936	225	vmovupd ymmword ptr [r13+0x1c40], ymm4	0.028s	0.028s
0x18000593f	768	call 0x18000aff8 < svml_expd>	0.016s	0.016s
0x180005944	Block 47:			
0x180005944	115	vmlpdd ymm4, ymm0, ymmword ptr [r13+0xc60]	0.003s	0.003s
0x18000594d	115	vmovupd ymmword ptr [r13+0x4e0], ymm4	0.101s	0.101s
0x180005956	225	vmovupd ymmword ptr [r13+0x1c60], ymm4	0.017s	0.017s
0x18000595f	763	vandpd ymm5, ymm9, ymmword ptr [r13+0x740]	0.028s	0.028s
0x180005968	88	vfmadd213pd ymm5, ymm8, ymm12	0.003s	0.003s
0x18000596d		vdivpd ymm4, ymm12, ymm5		
0x180005971	225	vmovupd ymmword ptr [r13+0x1780], ymm4	0.554s	0.554s
0x18000597a	763	vandpd ymm4, ymm13, ymm9	0.036s	0.036s
0x18000597f		vfmadd213pd ymm4, ymm8, ymm12		
0x180005984		vdivpd ymm4, ymm12, ymm4		
0x180005988	225	vmovupd ymmword ptr [r13+0x17a0], ymm4	0.338s	0.338s
0x180005991	763	vandpd ymm4, ymm9, ymmword ptr [r13+0x6e0]	0.016s	0.016s
0x18000599a	88	vfmadd213pd ymm4, ymm8, ymm12	0.002s	0.002s
0x18000599f		vdivpd ymm4, ymm12, ymm4		
0x1800059a3	225	vmovupd ymmword ptr [r13+0x17c0], ymm4	0.597s	0.597s
0x1800059ac	763	vandpd ymm4, ymm15, ymm9	0.043s	0.043s
0x1800059b1	88	vfmadd213pd ymm4, ymm8, ymm12	0.001s	0.001s
0x1800059b6		vdivpd ymm4, ymm12, ymm4		
0x1800059ba	225	vmovupd ymmword ptr [r13+0x17e0], ymm4	0.650s	0.650s
0x1800059c3	763	vandpd ymm4, ymm9, ymmword ptr [r13+0x680]	0.049s	0.049s
0x1800059cc	88	vfmadd213pd ymm4, ymm8, ymm12	0.004s	0.004s
0x1800059d1		vdivpd ymm4, ymm12, ymm4		
0x1800059d5	225	vmovupd ymmword ptr [r13+0x1800], ymm4	0.598s	0.598s
0x1800059de	763	vandpd ymm4, ymm14, ymm9	0.037s	0.037s
0x1800059e3		vfmadd213pd ymm4, ymm8, ymm12		
0x1800059e8		vdivpd ymm4, ymm12, ymm4		
0x1800059ec	225	vmovupd ymmword ptr [r13+0x1820], ymm4	0.556s	0.556s
0x1800059f5	763	vandpd ymm4, ymm9, ymmword ptr [r13+0x620]	0.027s	0.027s
0x1800059fe		vfmadd213pd ymm4, ymm8, ymm12		
0x180005a03		vdivpd ymm4, ymm12, ymm4		
0x180005a07	225	vmovupd ymmword ptr [r13+0x1840], ymm4	0.586s	0.586s
0x180005a10	763	vandpd ymm4, ymm9, ymmword ptr [r13+0x5e0]	0.038s	0.038s
0x180005a19	88	vfmadd213pd ymm4, ymm8, ymm12	0.001s	0.001s
0x180005a1e	163	vdivpd ymm4, ymm12, ymm4	0.001s	0.001s
0x180005a22	225	vmovupd ymmword ptr [r13+0x1860], ymm4	0.566s	0.566s
0x180005a2b	223	xor al, al	0.041s	0.041s
0x180005a2d		xor r14d, r14d		
0x180005a30		vmovupd ymmword ptr [r13+0xd20], ymm7		
0x180005a39		vmovdqu ymmword ptr [r13+0xd40], ymm10		

ON VENDOR-TUNED LIBRARIES

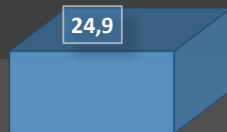


Matrix Multiply

Naive Matrix Multiply

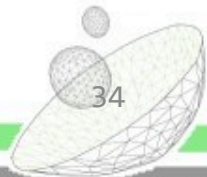
```
#pragma omp parallel for
for (int i = 0 ; i < rowsA ; ++i)
{
    for (int j = 0 ; j < colsB ; ++j)
    {
        double d = 0.0 ;
        for (int k = 0 ; k < colsA ; ++k)
        {
            d += A[i * colsA + k] * B[k * colsB + j] ;
        }
        C[i * colsB + j] = d ;
    }
} // */
```

MATRIX MULTIPLY (C++ / INTEL 15.0)



■ Naive

GFLOPS



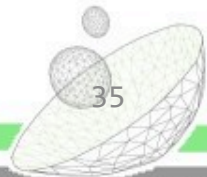
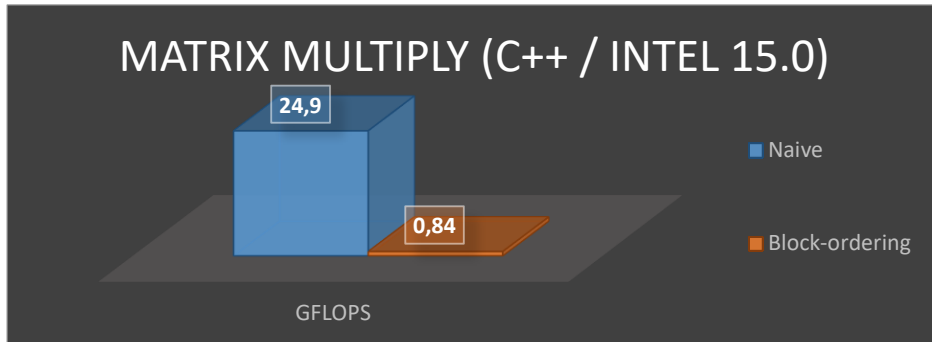
Matrix Multiply

Naive Matrix Multiply

```
#pragma omp parallel for
for (int i = 0 ; i < rowsA ; ++i)
{
    for (int j = 0 ; j < colsB ; ++j)
    {
        double d = 0.0 ;
        for (int k = 0 ; k < colsA ; ++k)
        {
            d += A[i * colsA + k] * B[k * colsB + j] ;
        }
        C[i * colsB + j] = d ;
    }
} // */
```

Splitting loops (better cache behavior?)

```
#pragma omp parallel for default(none) firstprivate(rowsA, colsA, A, colsB, B, C)
for(int ibk = 0 ; ibk < rowsA / SIZE ; ++ibk)
{
    for (int jbk = 0 ; jbk < colsB / SIZE ; ++jbk)
    {
        for (int i = ibk * SIZE ; i < (ibk + 1) * SIZE ; ++i)
        {
            for (int j = jbk * SIZE ; j < (jbk + 1) * SIZE ; ++j)
            {
                double d = 0.0 ;
                for (int k = 0 ; k < colsA ; ++k)
                {
                    d += A[i * colsA + k] * B[k * colsB + j] ;
                }
                C[i * colsB + j] = d ;
            }
        }
    }
} // */
```



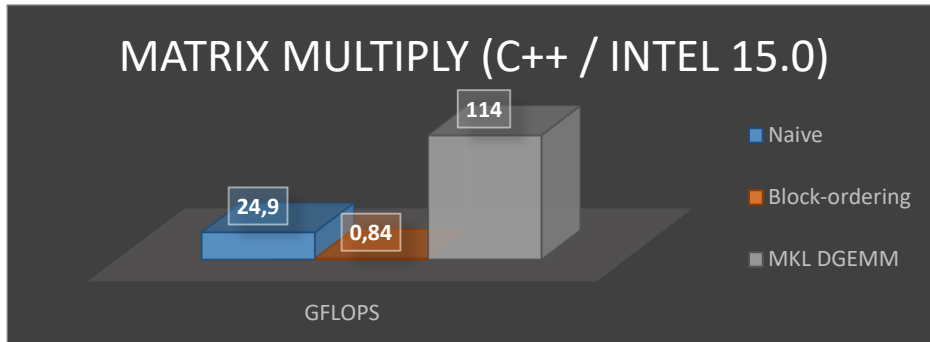
Matrix Multiply

Naive Matrix Multiply

```
#pragma omp parallel for
for (int i = 0 ; i < rowsA ; ++i)
{
    for (int j = 0 ; j < colsB ; ++j)
    {
        double d = 0.0 ;
        for (int k = 0 ; k < colsA ; ++k)
        {
            d += A[i * colsA + k] * B[k * colsB + j] ;
        }
        C[i * colsB + j] = d ;
    }
} // */
```

Splitting loops (better cache behavior?)

```
#pragma omp parallel for default(none) firstprivate(rowsA, colsA, A, colsB, B, C)
for(int ibk = 0 ; ibk < rowsA / SIZE ; ++ibk)
{
    for (int jbk = 0 ; jbk < colsB / SIZE ; ++jbk)
    {
        for (int i = ibk * SIZE ; i < (ibk + 1) * SIZE ; ++i)
        {
            for (int j = jbk * SIZE ; j < (jbk + 1) * SIZE ; ++j)
            {
                double d = 0.0 ;
                for (int k = 0 ; k < colsA ; ++k)
                {
                    d += A[i * colsA + k] * B[k * colsB + j] ;
                }
                C[i * colsB + j] = d ;
            }
        }
    }
} // */
```



Matrix-Multiply sounds simple, however it involves advanced features:

- Vector-unit operations
- Non-temporal write
- Several layers of memory prefetching
- Many corner cases for unaligned sizes, transposes, etc.

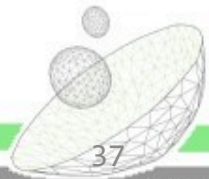
Prefer Vendor-Tuned Libraries



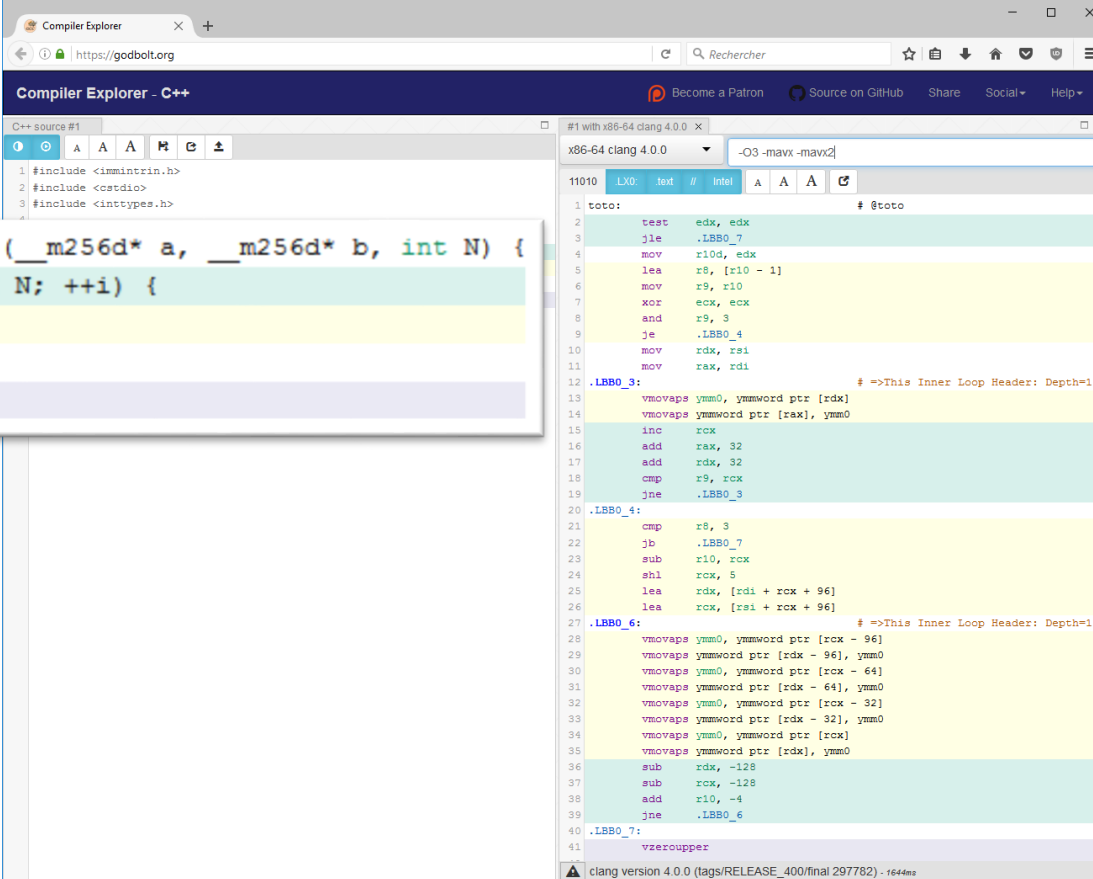
A Good Compiler Is Not Enough

Use Vendor-Tuned Libraries

- « What every programmer should know about memory », by *Ulrich Drepper*
 - It takes a lot to write (close to) optimal code
 - Understanding of core components of the system are necessary to get good performance (getting a compute-bound implementation of matrix multiply is **very hard**)
- Micro-architecture evolve
 - AVX means 256 bits operands => new instruction set wrt SSE
 - AVX-2 has more instructions => need to redefine some code (different latencies, fused multiply-add, integer operations, gather instruction)
 - AVX-512 is totally different, moreover flops/memops ratio evolves => need to rewrite
- Vendors provide optimized libraries (Intel MKL)
 - Prefer optimized libraries over hand-written versions
 - Most often better performance writing code to transition from custom data layout to optimized library's data layout
- **Hybridizer integrates these libraries** with Extensibility attributes
 - Available through wrapper methods (no overhead)
 - No overhead using these libraries
 - Same approach to integrate existing in-house developments



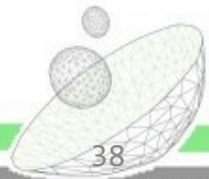
Want to do it anyway ?



The image shows a screenshot of the Compiler Explorer website. The left pane displays C++ source code for a function named `toto`. The right pane shows the corresponding assembly code generated by clang 4.0.0. A callout box highlights the C++ code.

```
extern "C" void toto(__m256d* a, __m256d* b, int N) {  
    for(int i = 0; i < N; ++i) {  
        a[i] = b[i];  
    }  
}
```

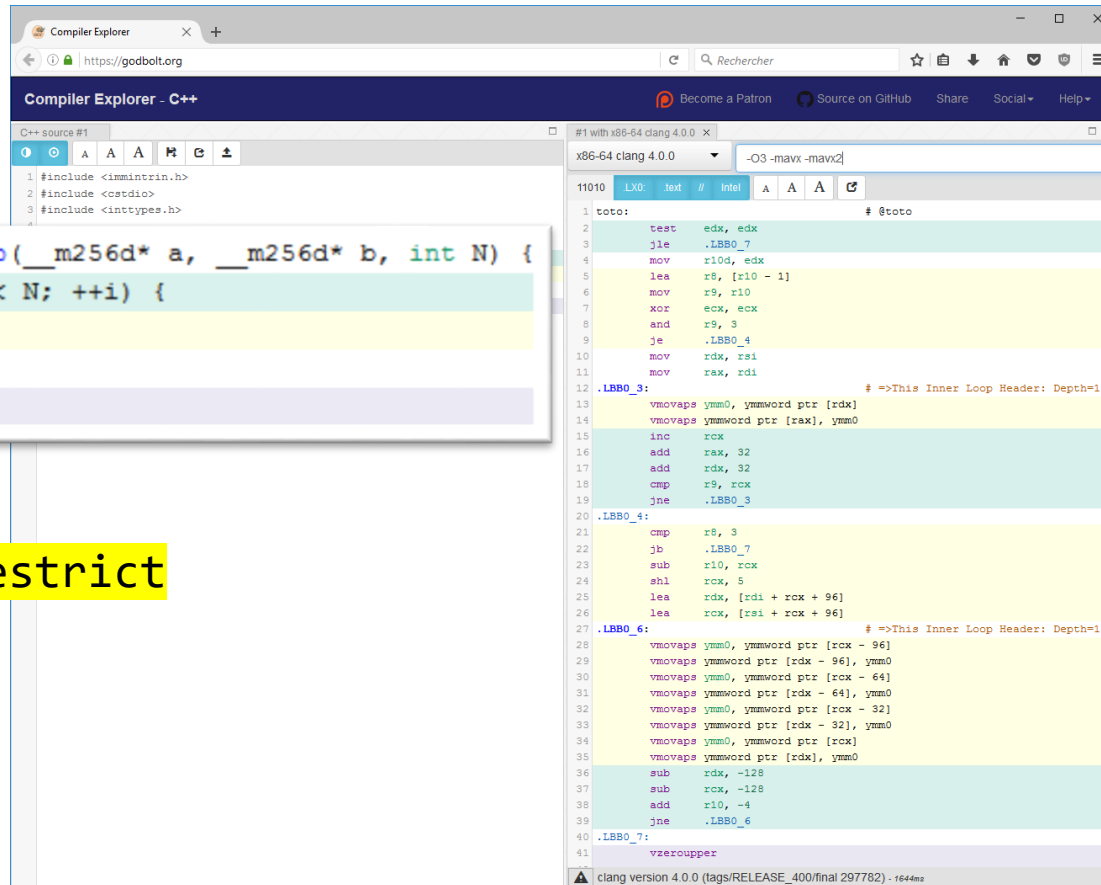
The assembly code shows the function `toto` starting at address 11010. It includes instructions for testing, moving, and comparing registers, as well as vector move instructions (`vmovaps`) for the main loop. The assembly is annotated with labels like `.LBB0_3`, `.LBB0_4`, and `.LBB0_6`.



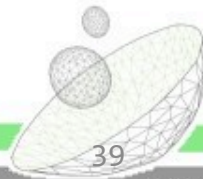
Want to do it anyway ?

```
extern "C" void toto(__m256d* a, __m256d* b, int N) {  
    for(int i = 0; i < N; ++i) {  
        a[i] = b[i];  
    }  
}
```

Forgot `__restrict`



The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code for a function named `toto` that takes two `__m256d` pointers and an integer `N`, and performs a loop copying elements from `b` to `a`. The right pane shows the assembly output for `x86-64 clang 4.0.0` with optimization flags `-O3 -mavx -mavx2`. The assembly includes instructions for AVX-512 vector operations, such as `vmovaps`, `vmovq`, `vmovd`, `vzeroupper`, and `vzeroupper`, along with control flow instructions like `test`, `jle`, `mov`, `lea`, `xor`, `and`, `je`, `mov`, `mov`, `inc`, `add`, `cmp`, `jne`, `cmp`, `jb`, `sub`, `shl`, `lea`, `lea`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `vmovaps`, `sub`, `sub`, `add`, `jne`, and `vzeroupper`.

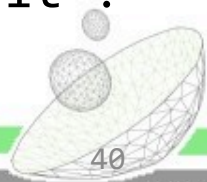


Want to do it anyway ?

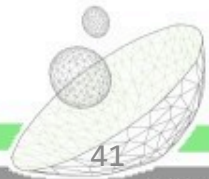
```
extern "C" void toto(__m256d* __restrict a, __m256d* b, int N) {
    for(int i = 0; i < N; ++i) {
        a[i] = b[i];
    }
}
```

```
toto:
1      test    edx, edx
2      jle    .LBB0_2
3
4      push   rax
5      dec    edx
6      shl   rdx, 5
7      add   rdx, 32
8      call  memcpy
9      add   rsp, 8
10
.LBB0_2:
11     ret
```

- Seems pointless anyway -> compiler will go back to it !



HYBRID VECTOR LIBRARY



Decorrelate Compute from Data

DATA



Data storage is implicit.
Enables full cache
awareness in
implementations

Developers express algorithm with
a few constraints that provide
compilers enough information for
optimization.

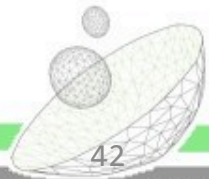


ALGORITHM



EXECUTION

Parallel constructs are expressed
by library so that compilers have
enough elements to optimize
execution



Example

- Discounting Cash Flows
 - Several cash flows (each have c, T)
 - Stochastic rate (may be interpolated)
 - Need 1st and 2nd order moments

$$DF(T, r, c) = c \cdot e^{-rT}$$

$$\Pi = \sum_{i=0}^{sim} DF(T, r_i, c)$$

```
double func(int nsimul, double T, double c)
{
    hybridvector* hv ;
    hv1_create(&hv, nsimul);           // allocate "vector"

    generate_rates(hv, nsimul);       // hv [i] <- r [i]

    hv1_mul_scalar(hv, -T);           // hv [i] <- hv [i] * (-T)
    hv1_apply_exp(hv);                // hv [i] <- exp (hv [i])
    hv1_mul_scalar(hv, c);            // hv [i] <- c * hv [i]

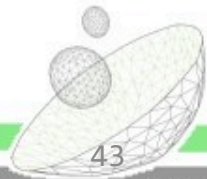
    frozenvector* fv;
    double PV;

    hv1_freeze(&fv, hv);              // compute moments
    hv1_frozen_avg(fv, &PV);          // PV <- Sum (hv [i])

    hv1_destroy(hv);                  // clean-up
    hv1_frozen_destroy(fv);

    return PV;
}
```

- No explicit memory allocation
- Focus on algorithm
- No explicit execution scheme (no loop)



Example

- Viewing algorithm as a graph
 - Rates: « data » node
 - Freeze: « execution » node
 - Runtime can re-arrange operations

```
double func(int nsimul, double T, double c)
{
    hybridvector* hv ;
    hv1_create(&hv, nsimul);           // allocate "vector"

    generate_rates(hv, nsimul);        // hv [i] <- r [i]

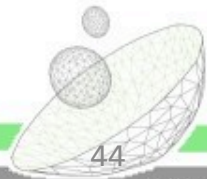
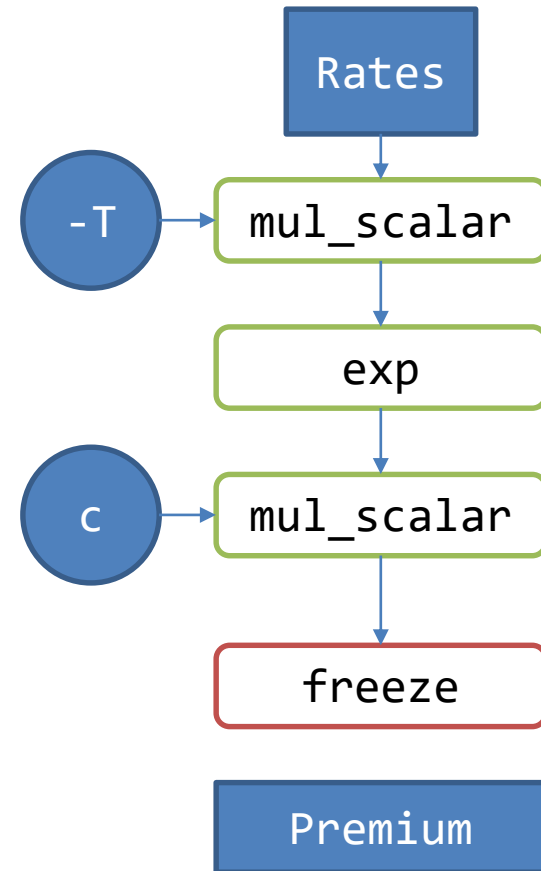
    hv1_mul_scalar(hv, -T);            // hv [i] <- hv [i] * (-T)
    hv1_apply_exp(hv);                // hv [i] <- exp (hv [i])
    hv1_mul_scalar(hv, c);             // hv [i] <- c * hv [i]

    frozenvector* fv;
    double PV;

    hv1_freeze(&fv, hv);              // compute moments
    hv1_frozen_avg(fv, &PV);          // PV <- Sum (hv [i])

    hv1_destroy(hv);                  // clean-up
    hv1_frozen_destroy(fv);

    return PV;
}
```



Example

- Lambdas !
 - Alternate approach
 - More readable
 - Easier to test

```
int func(int nsimul, double T, double c)
{
    hybridvector* hv ;
    hv1_create(&hv, nsimul);

    generate_rates(hv, nsimul);

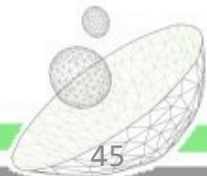
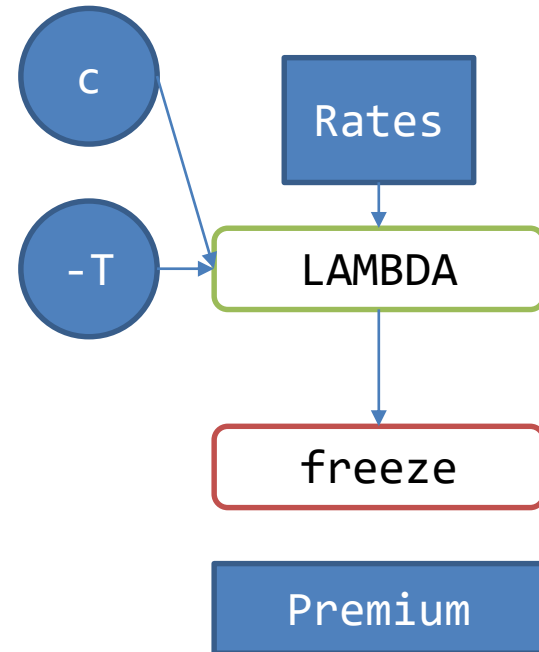
    hv1_apply_func <>(hv, [c,T](size_t k, double rate) -> double
    {
        return c * ::exp(-T * rate);
    });

    frozenvector* fv;
    double PV;

    hv1_freeze(&fv, hv);
    hv1_frozen_avg(fv, &PV);

    hv1_destroy(hv);
    hv1_frozen_destroy(fv);

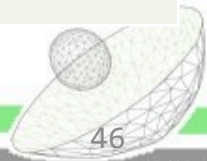
    return PV;
}
```



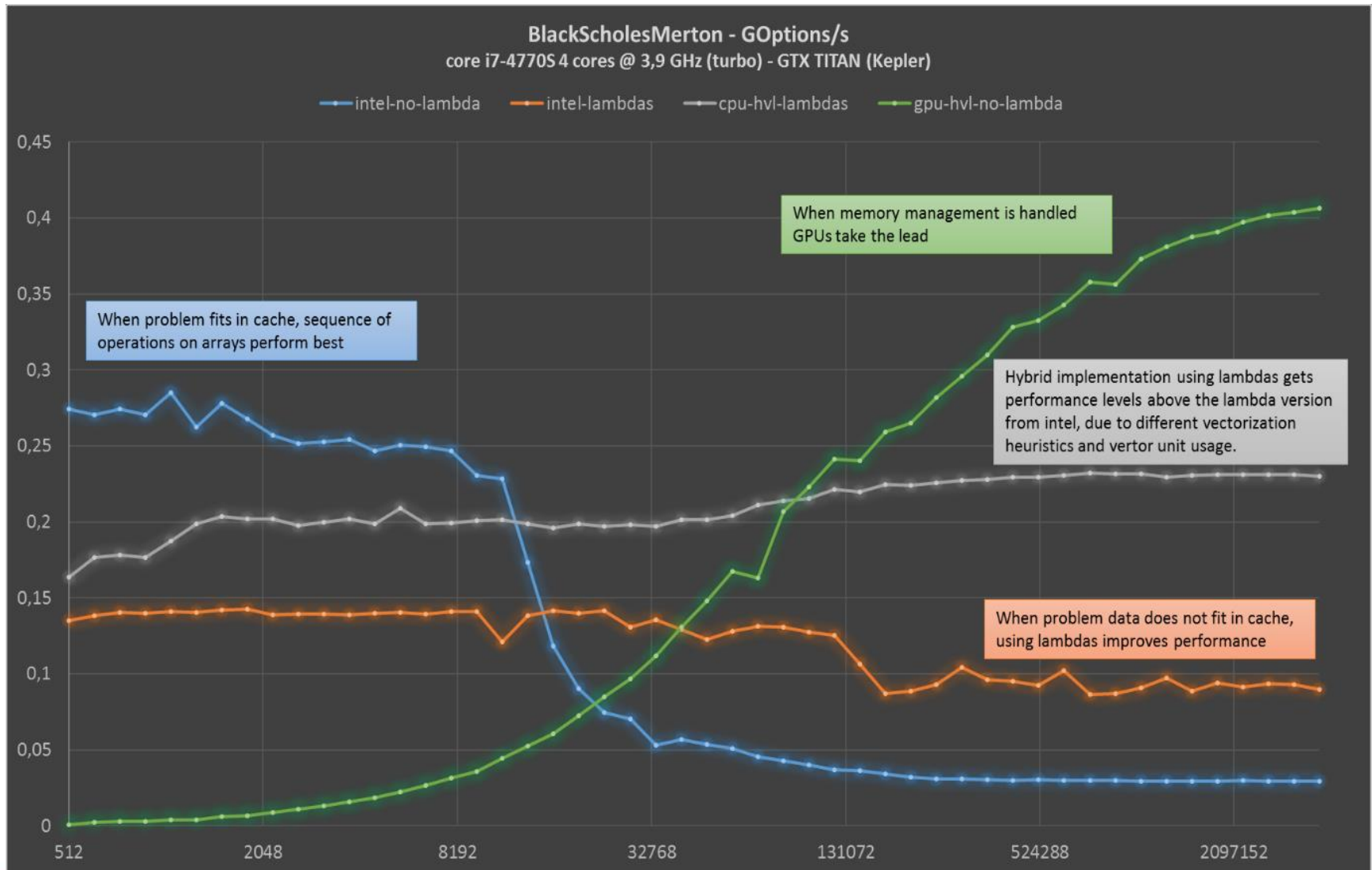
Variety of Implementations

- Using this API, algorithm and runtime implementation are decoupled
- Implicit data allocation allows alternate hardware with no code change
- Computations don't need to occur when scheduled: they are solely needed at freeze
 - DAG is constructed and evaluated at freeze

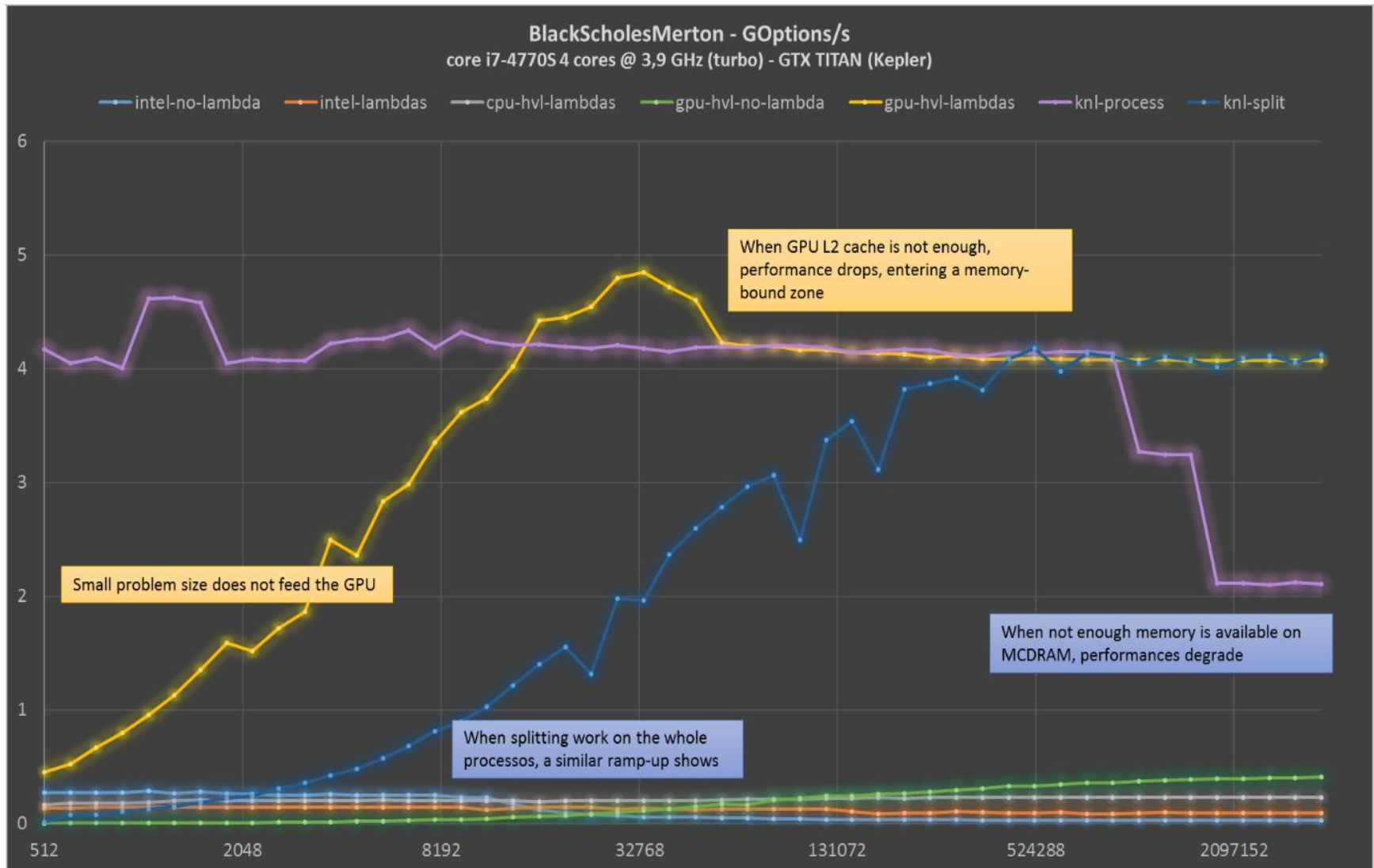
Implementation	lambda	platform	Notes
Intel-no-lambda		Intel X86 (AVX2)	Mapped on MKL
CPU-hv1-lambda	Y	Intel X86 (AVX2)	
GPU-hv1-no-lambda		Kepler GPU	Equivalent of MKL
GPU-hv1-lambda	Y	Kepler GPU	



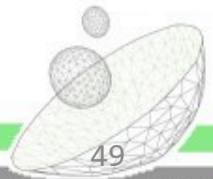
Performance On Black-Scholes Options



Performance On Black-Scholes Options

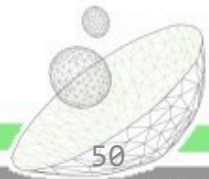


SUMMARY



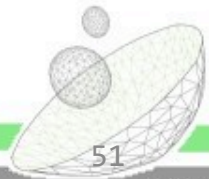
Take Home Message

- Modernize Code
 - Without code change, performance will stall or degrade on future hardware



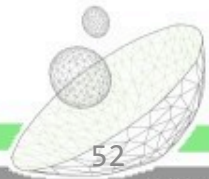
Take Home Message

- Modernize Code
 - Without code change, performance will stall or degrade on future hardware
- Modernize Code
 - Be explicit in the code to offer parallelization opportunities to the compiler



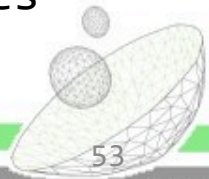
Take Home Message

- Modernize Code
 - Without code change, performance will stall or degrade on future hardware
- Modernize Code
 - Be explicit in the code to offer parallelization opportunities to the compiler
- Modernize Code
 - Memory access is the elephant in the room !

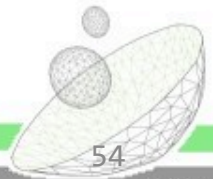


Take Home Message

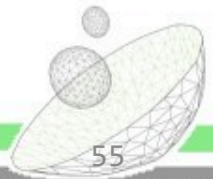
- Modernize Code
 - Without code change, performance will stall or degrade on future hardware
- Modernize Code
 - Be explicit in the code to offer parallelization opportunities to the compiler
- Modernize Code
 - Memory access is the elephant in the room !
- Modernize Code
 - Adapt source code to make use of vendor-tuned libraries



Thank you



Bonus/backup Slides



Benchmark-Level Performances

```

const double A1 = 0.31938153;
const double A2 = -0.356563782;
const double A3 = 1.781477937;
const double A4 = -1.821255978;
const double A5 = 1.330274429;
const double RSQRT2PI = 0.39894228040143267793994605993438;

public static double CND(double d)
{
    double K = 1.0 / (1.0 + 0.2316419 * Math.Abs(d));

    double cnd = RSQRT2PI *
        Math.Exp(-0.5 * d * d)
        *
        (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));

    if (d > 0)
        cnd = 1.0 - cnd;

    return cnd;
}

[Kernel]
public static void CallPut(out double call, out double put,
    double spot, double strike,
    double rate, double sigma, double maturity)
{
    double sigmaSqrtT = sigma * Math.Sqrt(maturity);
    double d1 = (Math.Log(spot / strike) + (rate +
        sigma * sigma / 2.0) * maturity) / sigmaSqrtT;
    double d2 = d1 - sigmaSqrtT;
    double kert = strike * Math.Exp(-rate * maturity);
    double CNDD1 = CND(d1);
    double CNDD2 = CND(d2);
    call = spot * CNDD1 - kert * CNDD2;
    put = kert * (1.0 - CNDD1);
}
    
```



```

static double cnd(double d)
{
    const double A1 = 0.31938153;
    const double A2 = -0.356563782;
    const double A3 = 1.781477937;
    const double A4 = -1.821255978;
    const double A5 = 1.330274429;
    const double RSQRT2PI = 0.39894228040143267793994605993438;

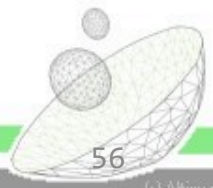
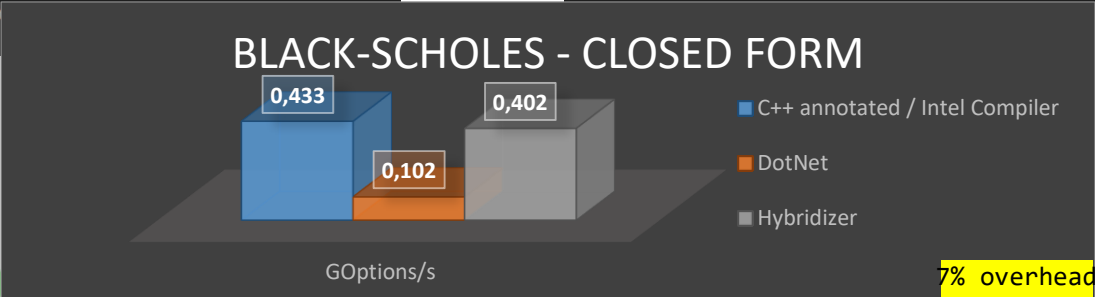
    double K = 1.0 / (1.0 + 0.2316419 * fabs(d));

    double cnd = RSQRT2PI * exp(- 0.5 * d * d) *
        (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));

    if (d > 0)
        cnd = 1.0 - cnd;

    return cnd;
}

static void bsm(double& call, double& put, double S0,
    double K, double r, double sigma, double T)
{
    double sigmaSqrtT = sigma * ::sqrt (T) ;
    double d1 = (::log(S0/K) + (r + sigma*sigma/2.0) * T) / sigmaSqrtT ;
    double d2 = d1 - sigmaSqrtT ;
    double kert = K * ::exp (-r * T) ;
    double CNDD1 = cnd(d1);
    double CNDD2 = cnd(d2);
    call = S0 * CNDD1 - kert * CNDD2;
    put = kert * (1.0 - CNDD1);
}
    
```



Extended features

Virtual Functions

- Interfaces / abstract classes and inheritance is supported
- Underlying implementation is a function-table

Object oriented programming productivity maintained ...

Generics

- Generics get mapped onto templates
- C++ template concepts are expressed by DotNet/Java generics constraints
- Restored performance

... And overhead can be removed

