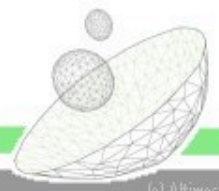
Performance Out of the Box On Multicore and Manycore Hardware with **Code Modernization**

Embrace Micro-Architecture Changes

Abstract-Out Instruction Set Variety

Achieve State-Of-The-Art Performance



PROCESSORS AND MOORE'S LAW

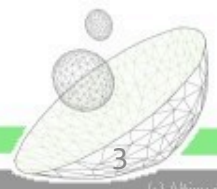
Processors Evolution

- Processors have changed

year	*2000*	2016	2016	2016	2014
processor	*Pentium 4*	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 ~> 50)
- Multithreading is not the only issue (SIMD/SIMT ratio)

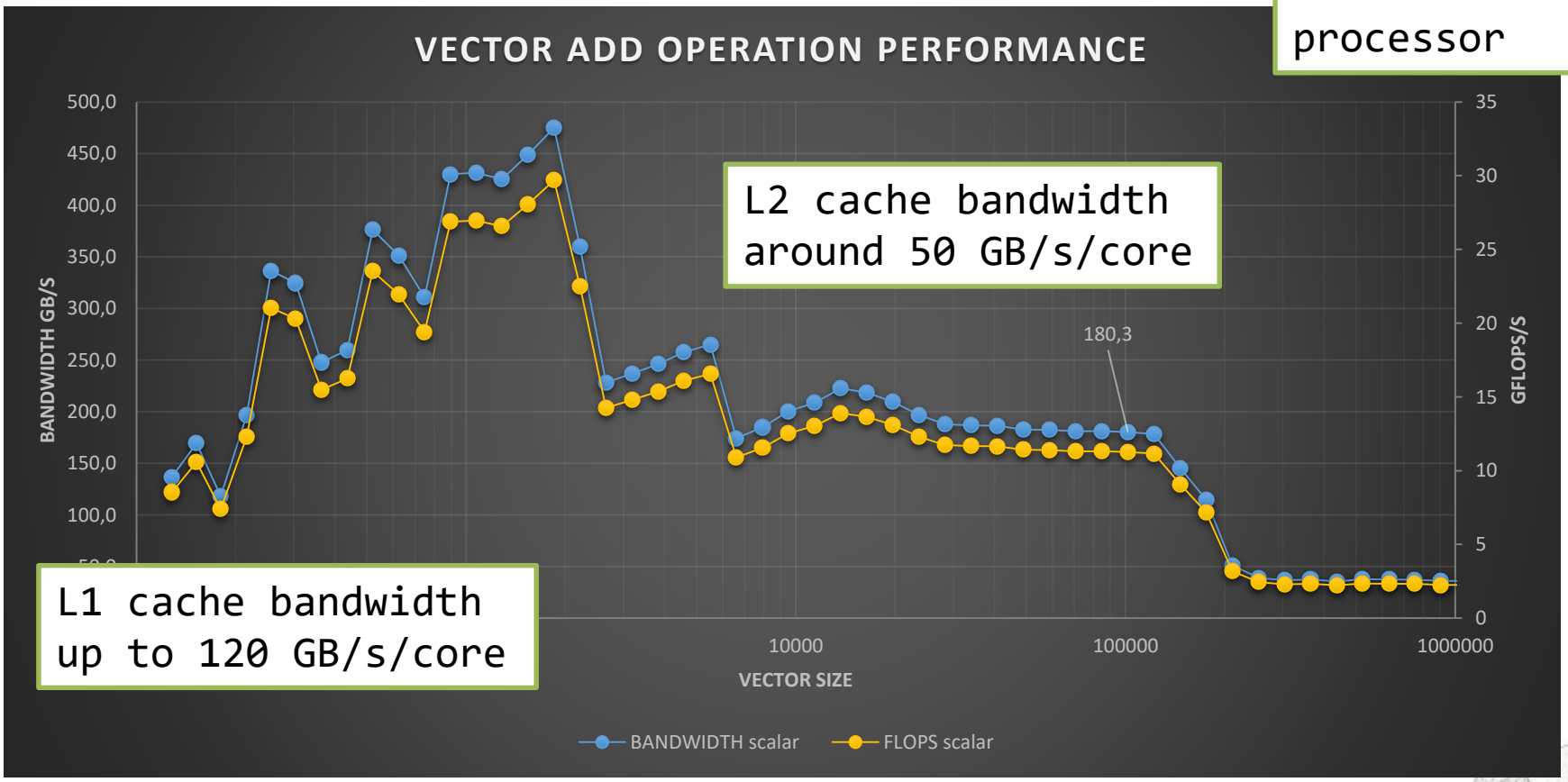
- Keeping-up with technology changes requires significant software development effort and training



Processor Evolution

- Adding a constant to a vector of values

Peak is 112 for this processor



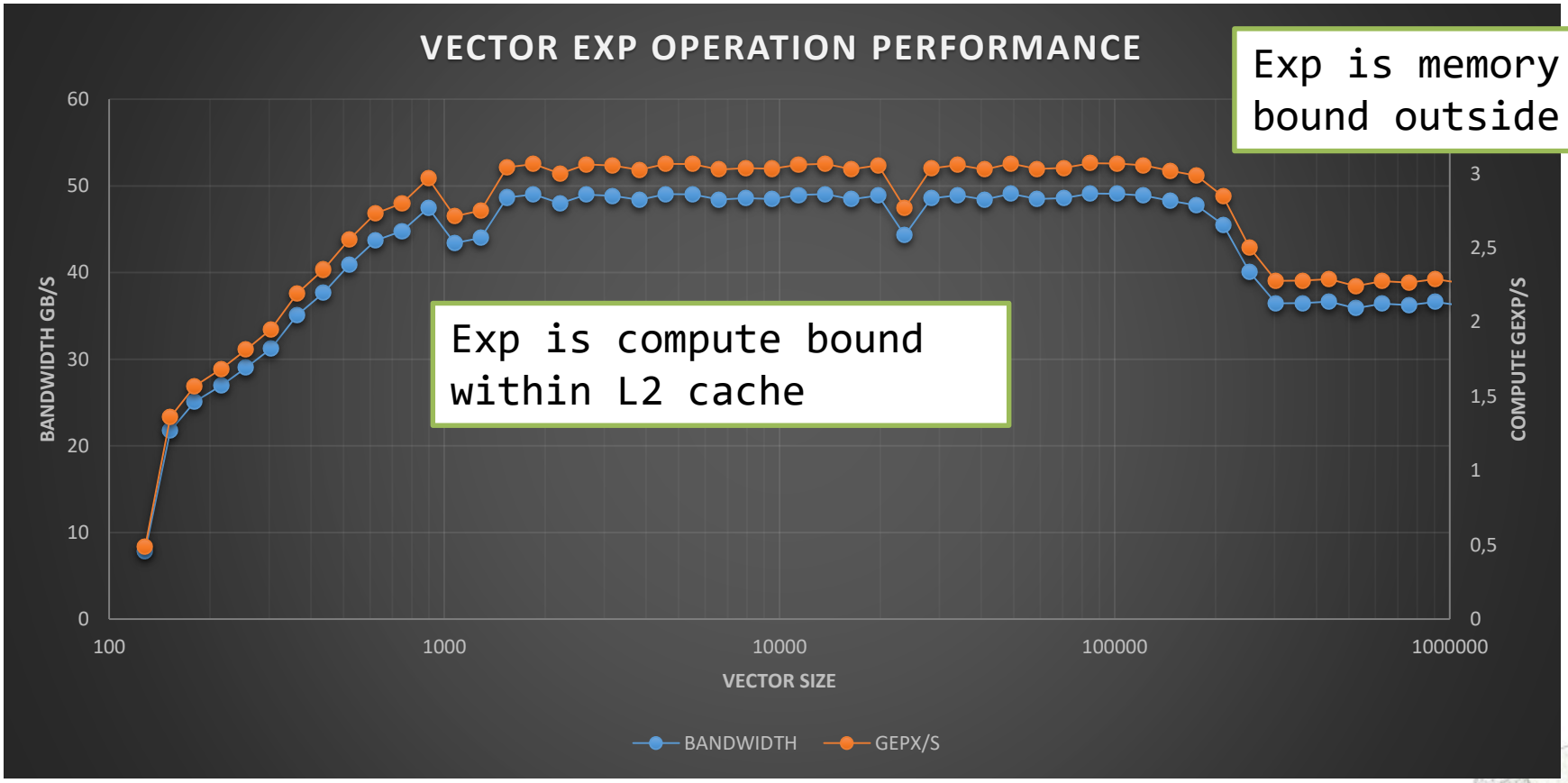
L2 cache bandwidth around 50 GB/s/core

L1 cache bandwidth up to 120 GB/s/core

Xeon E5-1620v4 – flop/memop = 23

Processor Evolution

- Getting exp of a vector



Xeon E5-1620v4 – flop/memop = 23

Moore's Law FLOPS And BANDWIDTH

FLOPS double every

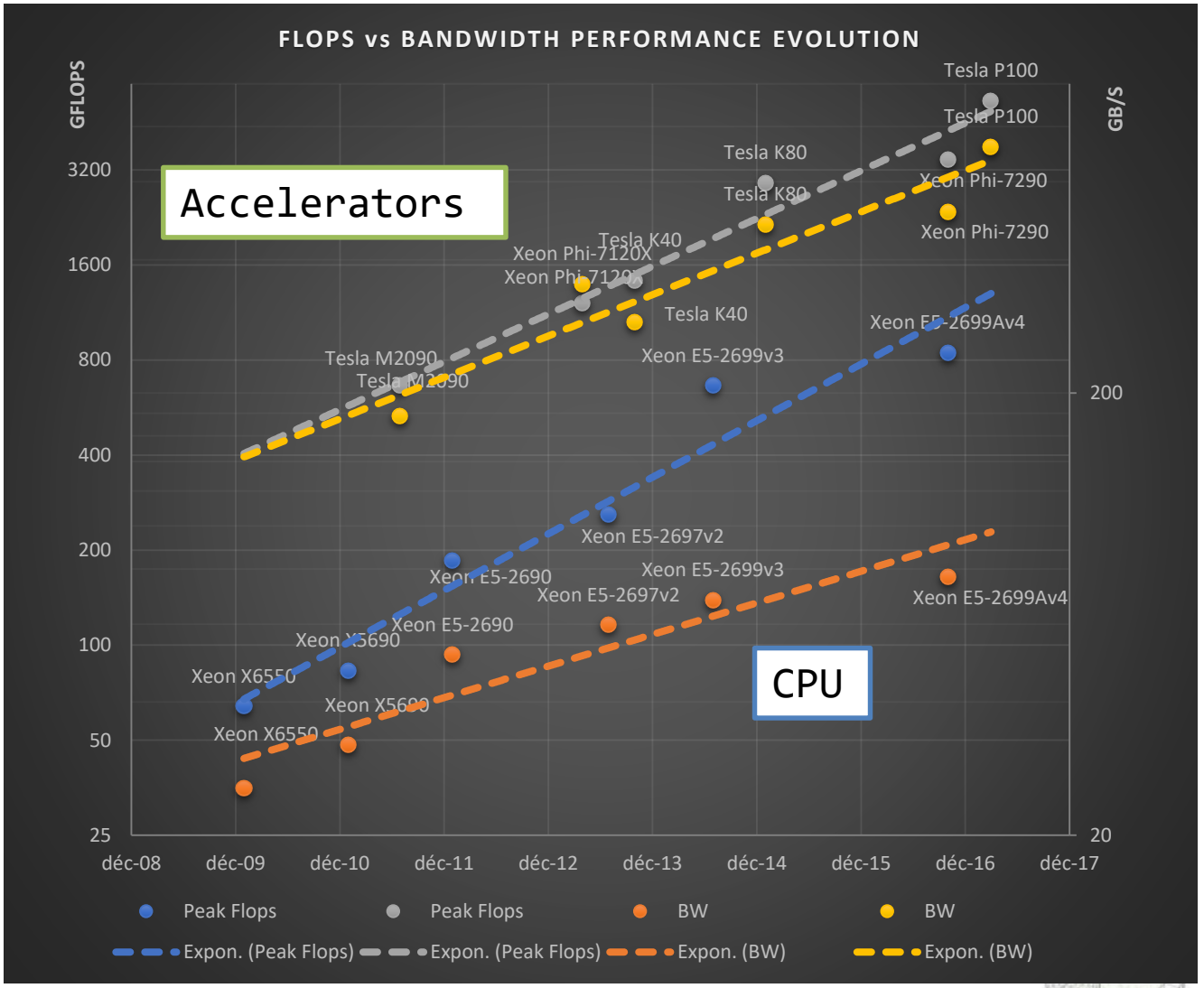
CPU: 1.8y

ACC: 1.9y

BANDWIDTH double every

CPU: 4.3y

ACC: 2.8y



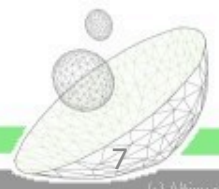
Moore's Law: Initial Understanding is Over

- 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

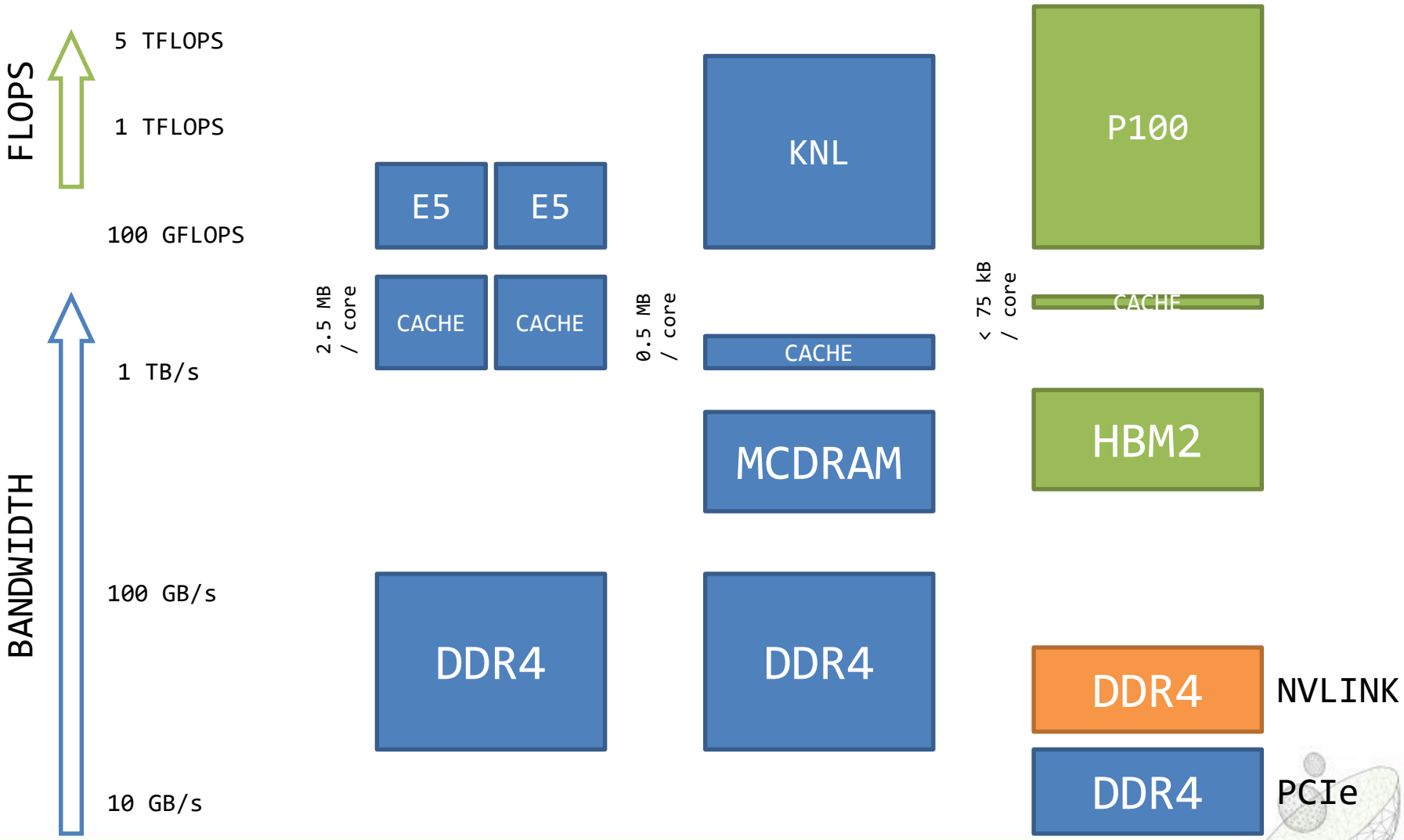


Programmers Perspective

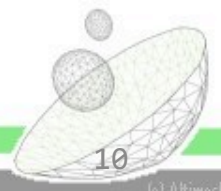
- A processor has
  - Vector units
  - Simultaneous contexts
  - Memory layers

	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

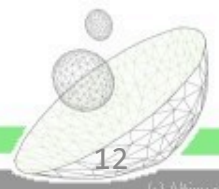
```
void func (int n, double* a, double* b)
{
    for (int k = 0 ; k < n ; ++k)
    {
        a [k] += b[k] ;
    }
}
```

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

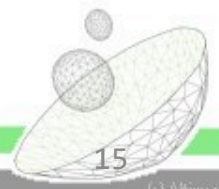
```
void func(int n, double* a, double* b)
{
    for (int k = 0; k < n; ++k)
    {
        *(a++) += *(b++);
    }
}
```

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

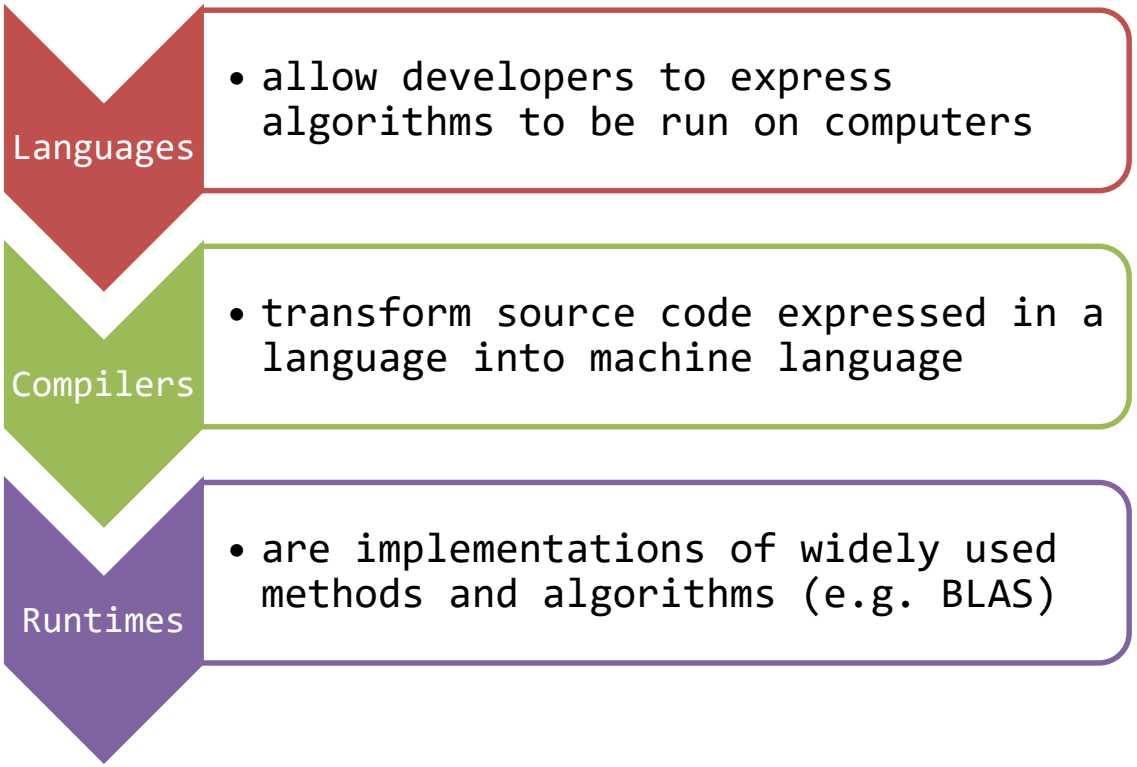
USING C# FOR HIGH THROUGHPUT COMPUTING

Language Compilers and Runtimes

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

Compiler optimization: extract parallelism (with help)

Runtime optimization: Use vendor-tuned libraries

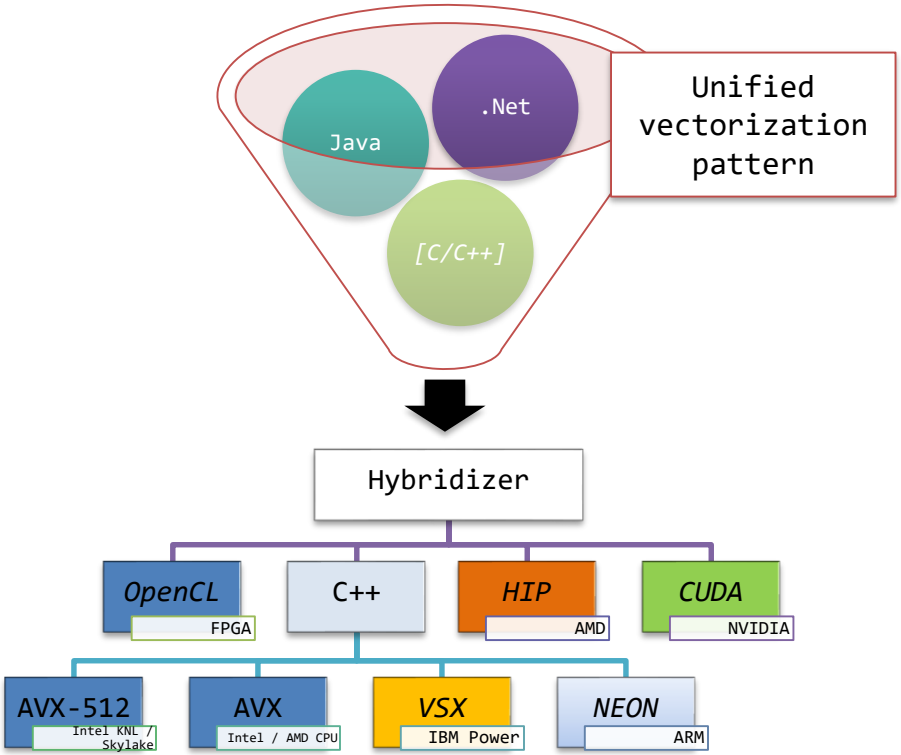


Language choice should not impact performance

Hybridizer Solution

- Input
  - .Net
  - Java
  - *[C/C++]*
- Environments:
  - Windows / Linux

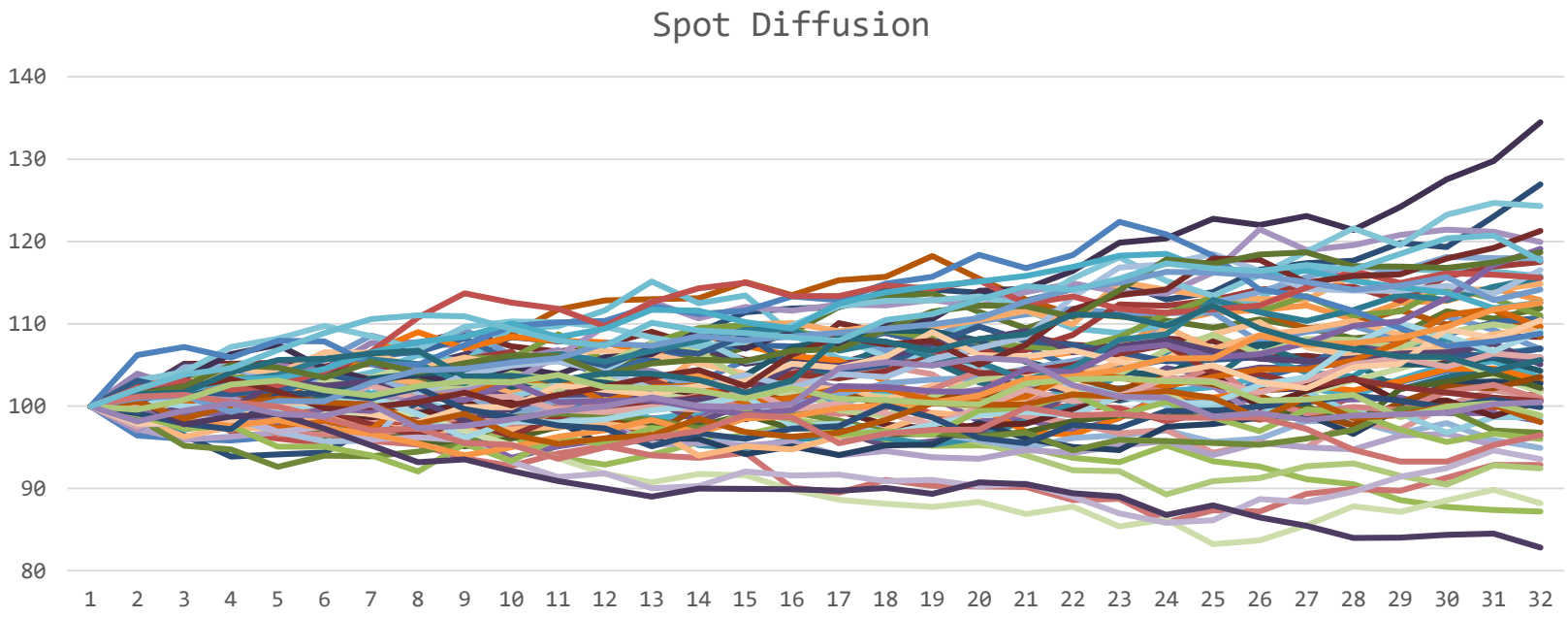
- Generate source code
  - **CUDA**/C for NVIDIA GPU
  - C++ (**vectorized**) for native platforms



Compilation pipeline to replace dot net JIT

Example Walkthrough

- Financial Model Spot Diffusion
  - Fundaments 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

Simulation Outer Loop

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

Time 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] + ...;
    }
}
```

Financial Model Spot Diffusion

Dot net source code
Generic parameters for flexibility

C#

```
[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;
        }
    }
}
```

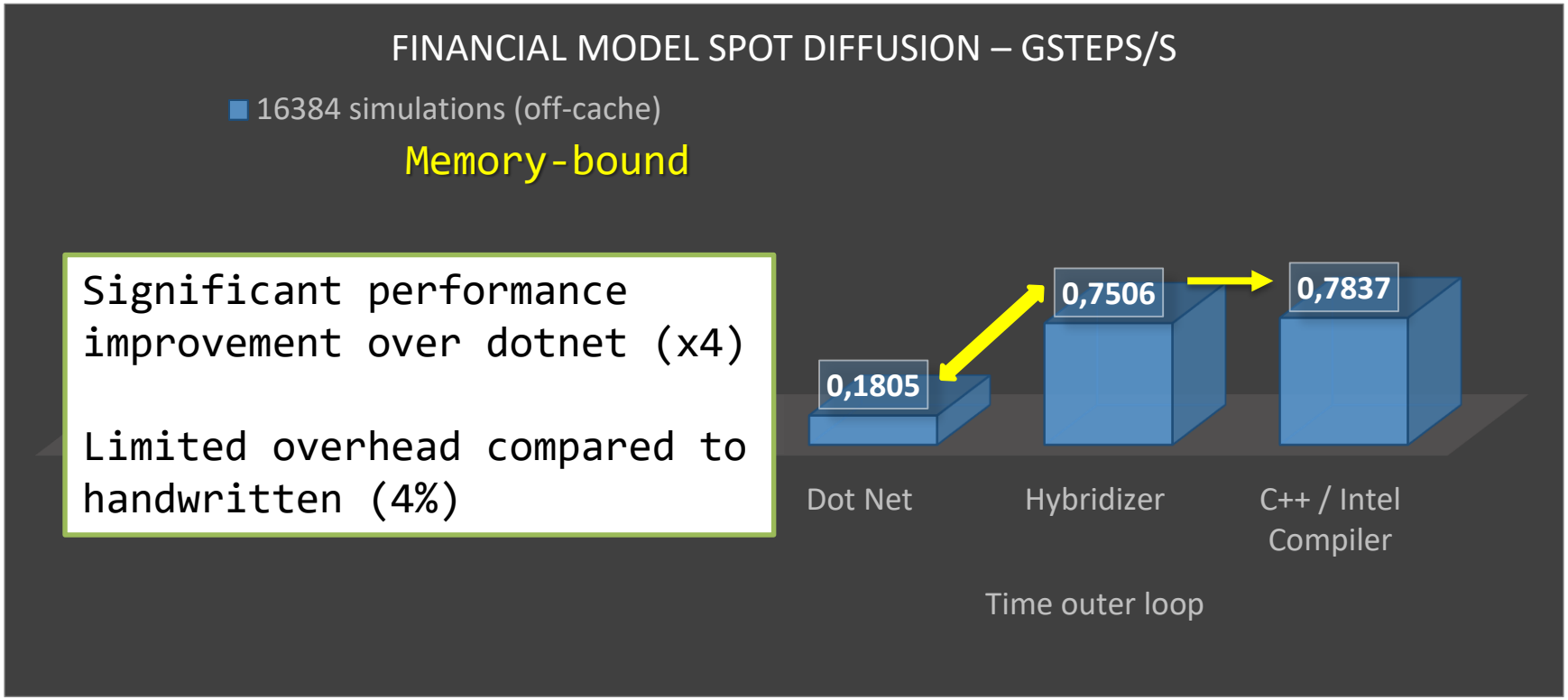
C++ source code with annotations
(two outer loop configurations)

C++

```
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 ;
        }
    }
}
```

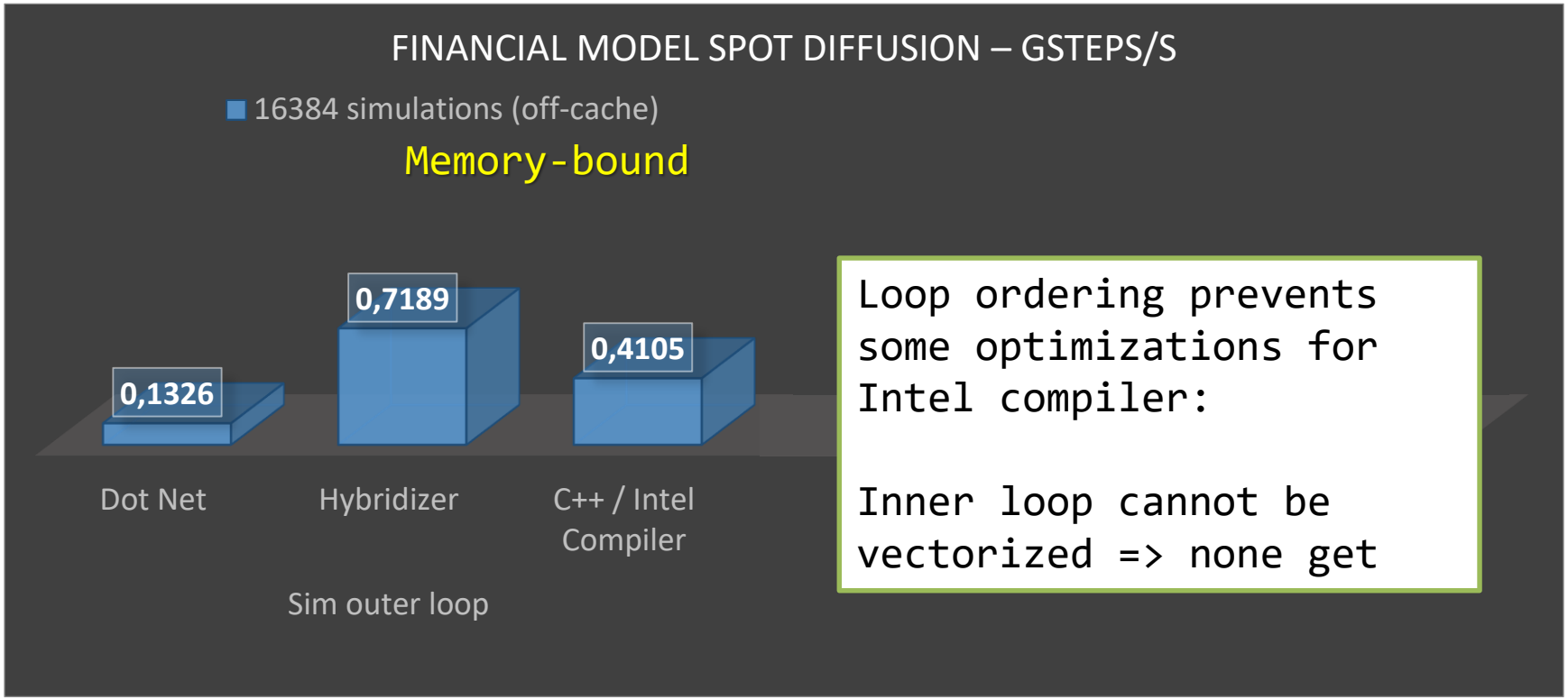
```
#pragma omp parallel for
for (int th = 0 ; th < 8 ; ++th)
{
    int simFrom = th * simCount / 8 ;
    int simTo = (th+1) * simCount / 8 ;
    for (int time = 0 ; time < datesCount ; ++time)
    {
        double* lnS = logSpot + (simCount * time) ;
        const double* brow = brownian + (time * simCount) ;
        #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



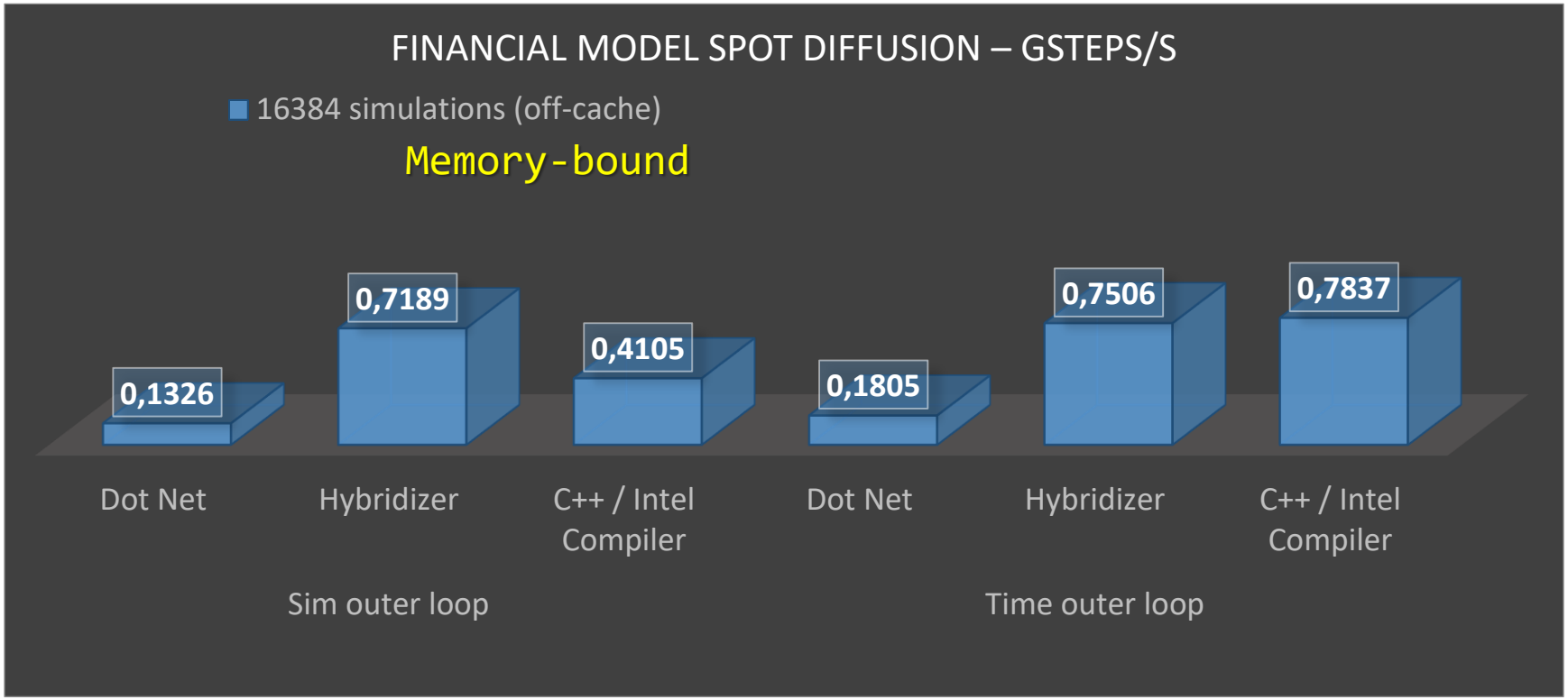
- **Comparing object-oriented code, with generics, processed by Hybridizer**
- **with hand-written optimized C++ code compiled with Intel Composer 2015**

Black-Scholes-Merton Diffusion



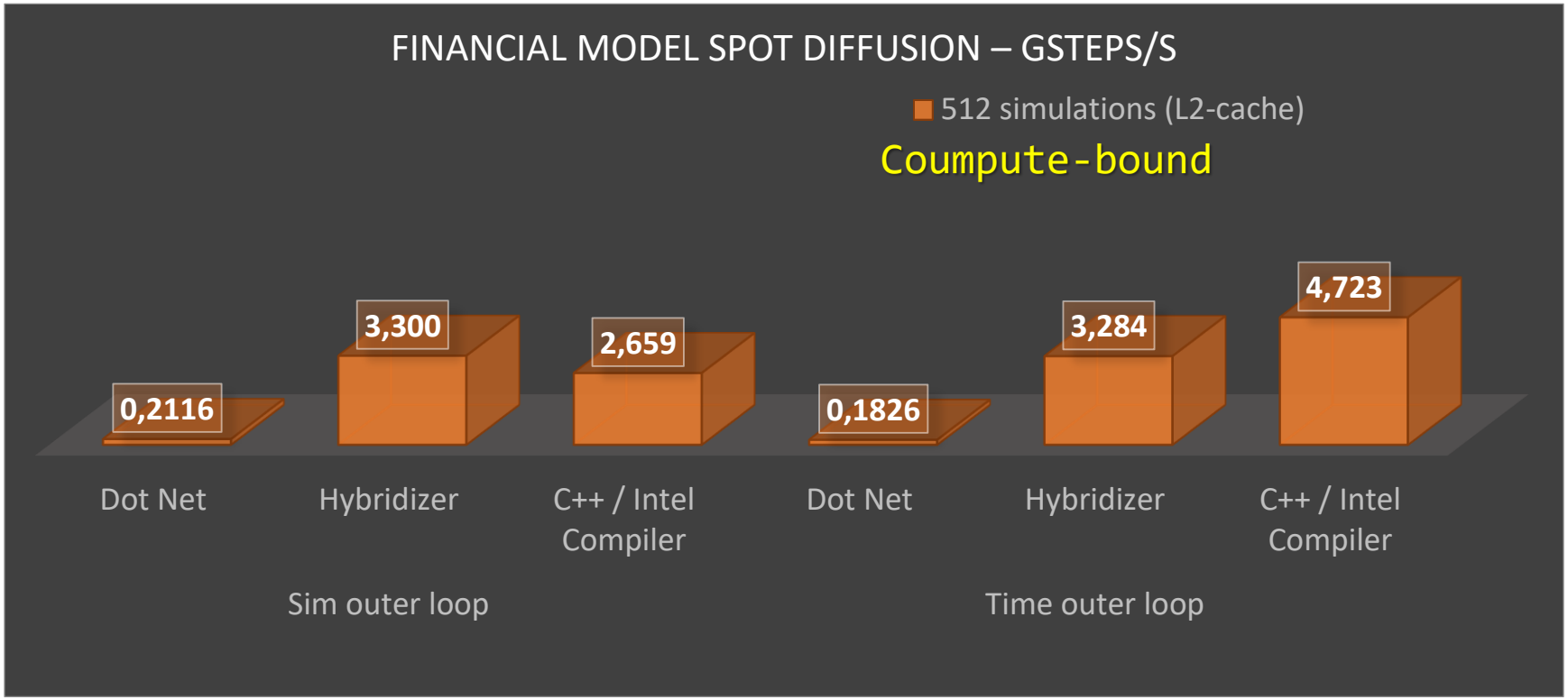
- **Comparing object-oriented code, with generics, processed by Hybridizer**
- **with hand-written optimized C++ code compiled with Intel Composer 2015**

Black-Scholes-Merton Diffusion



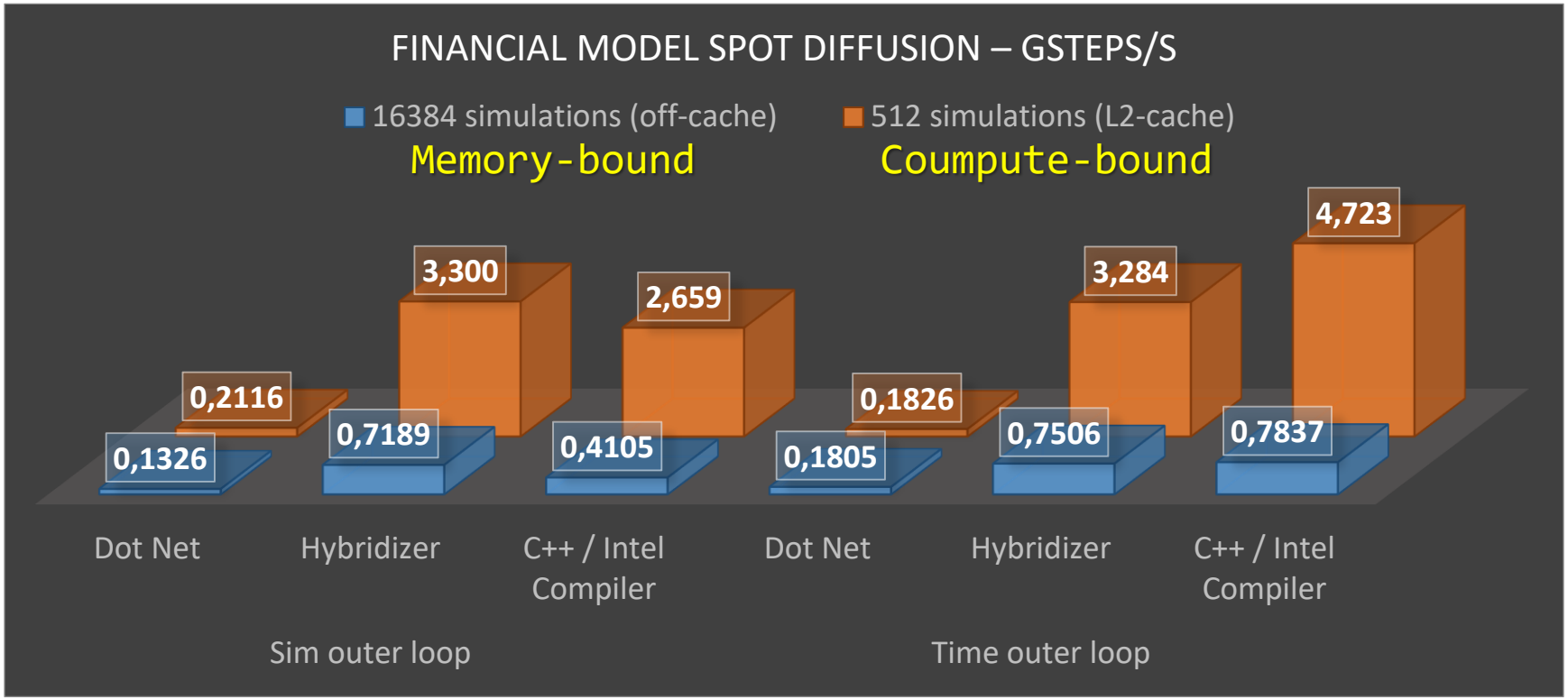
- **Comparing object-oriented code, with generics, processed by Hybridizer**
- **with hand-written optimized C++ code compiled with Intel Composer 2015**

Black-Scholes-Merton Diffusion



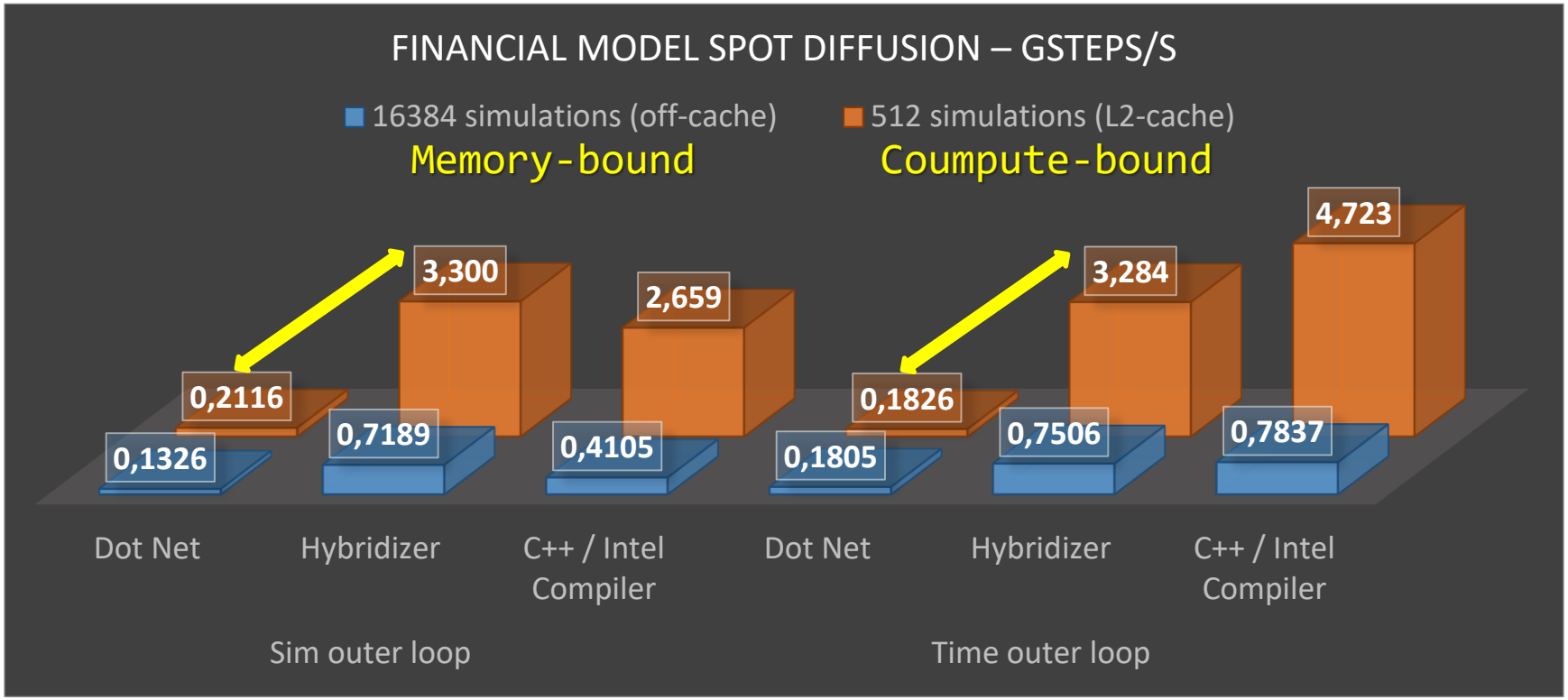
- **Comparing object-oriented code, with generics, processed by Hybridizer**
- **with hand-written optimized C++ code compiled with Intel Composer 2015**

Black-Scholes-Merton Diffusion



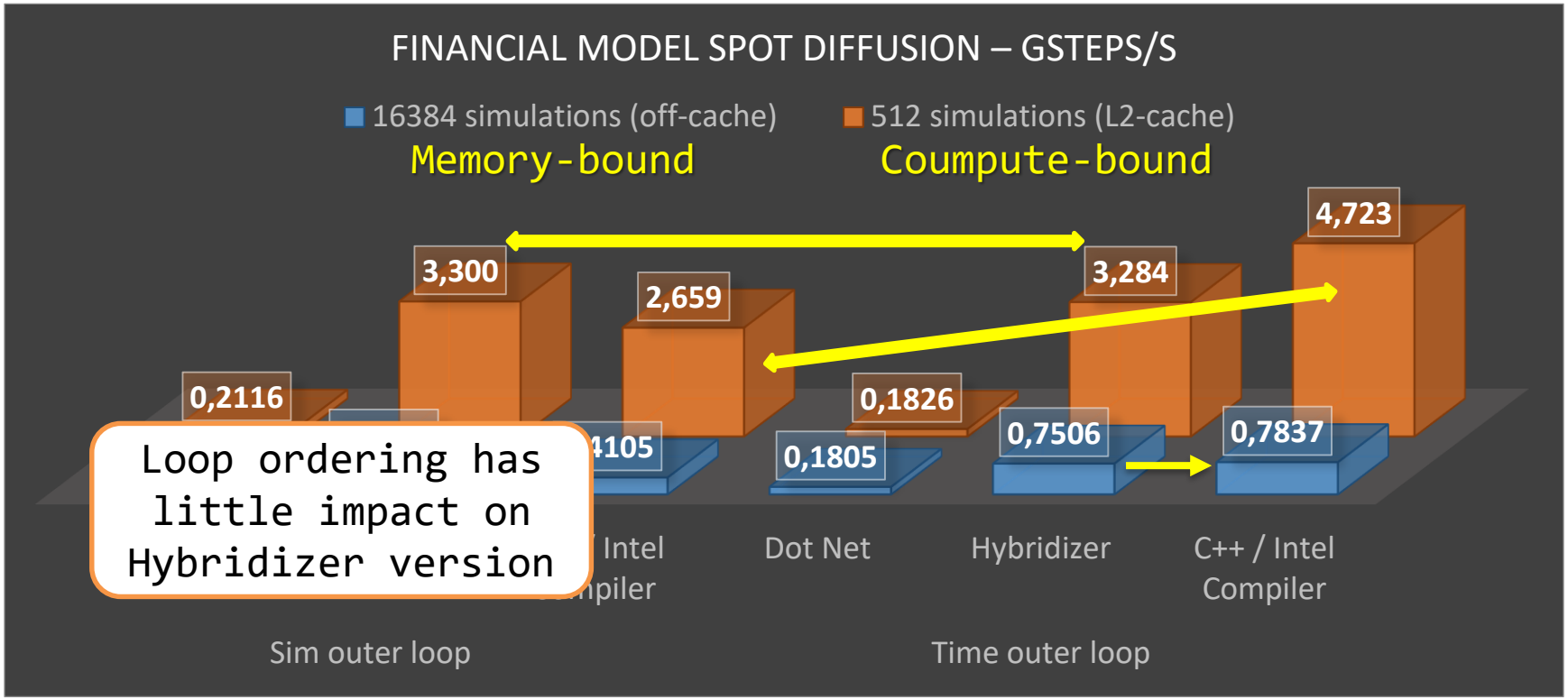
- **Comparing object-oriented code, with generics, processed by Hybridizer**
- **with hand-written optimized C++ code compiled with Intel Composer 2015**

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.**

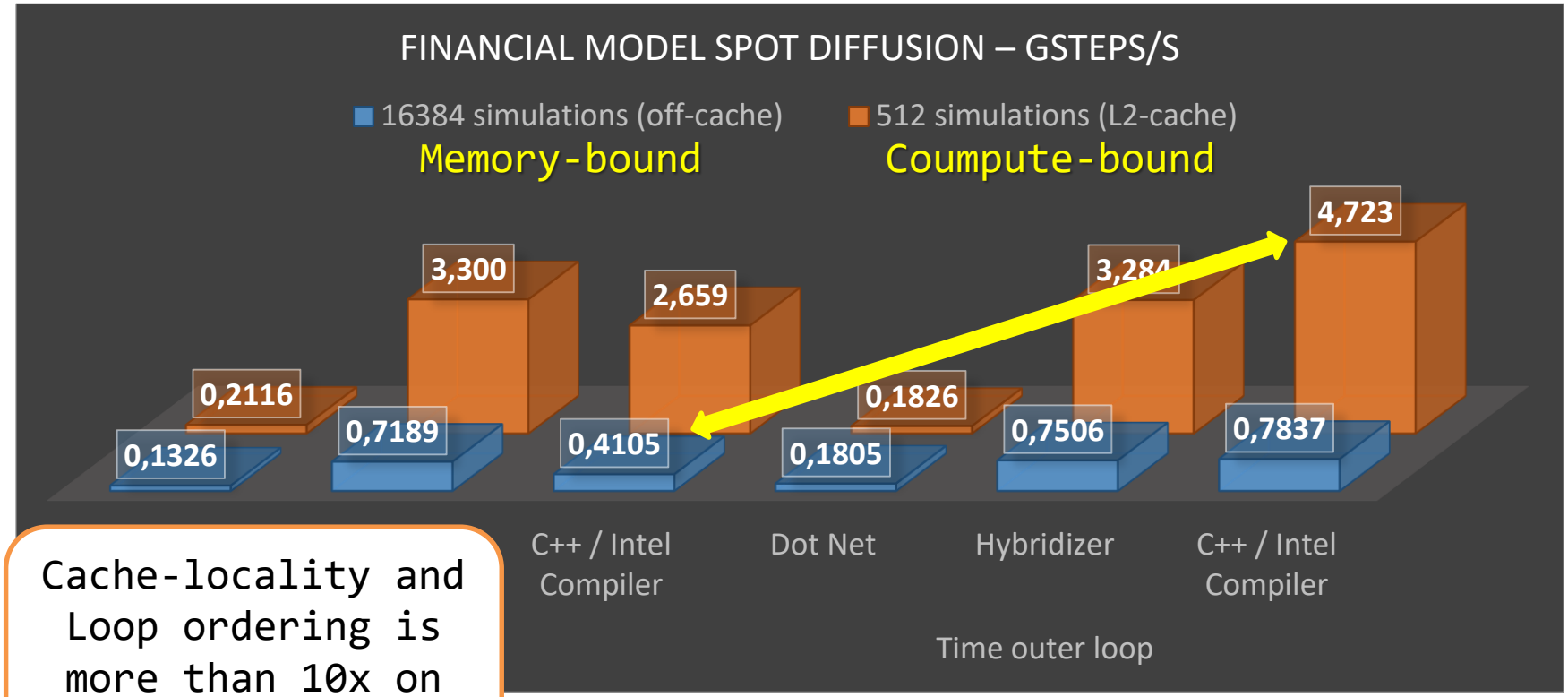
Black-Scholes-Merton Diffusion



- 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



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

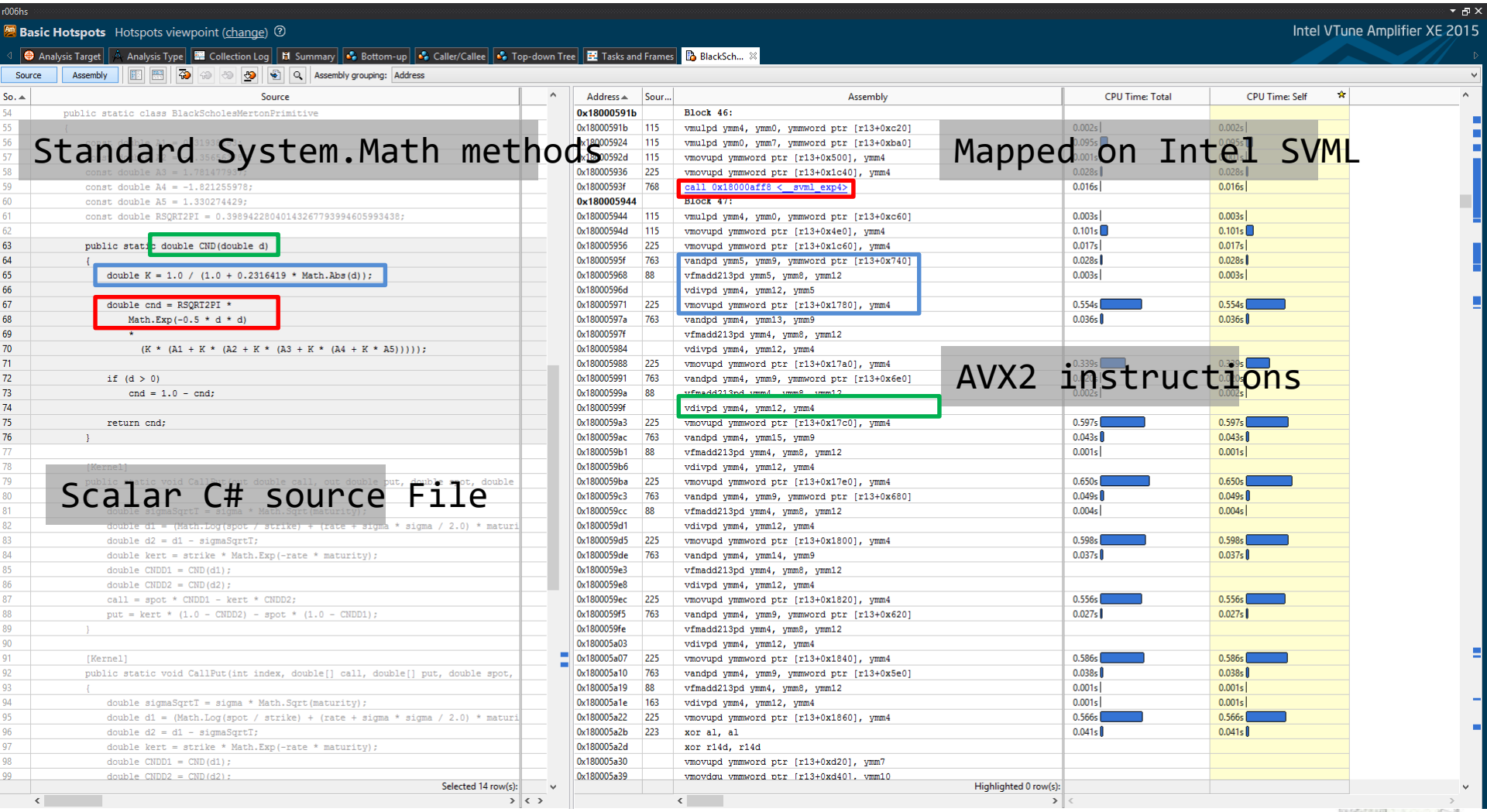
- roves dotnet
- Object oriented programming preserved: single version of source code, reduces operational risk / testing costs.

- 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

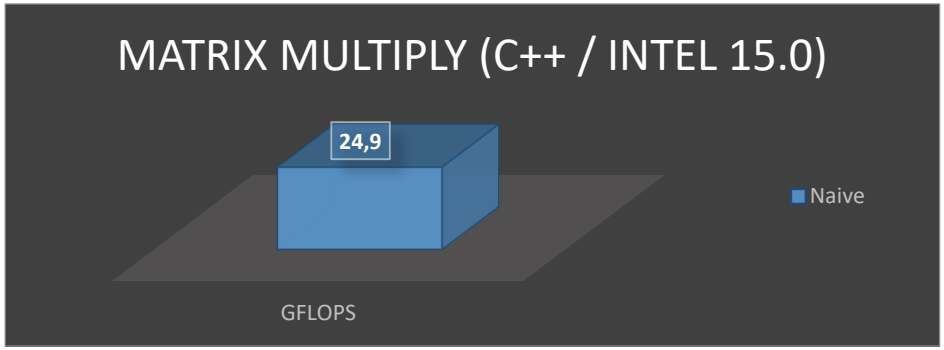


ON VENDOR-TUNED LIBRARIES

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 ;
    }
} // */
```



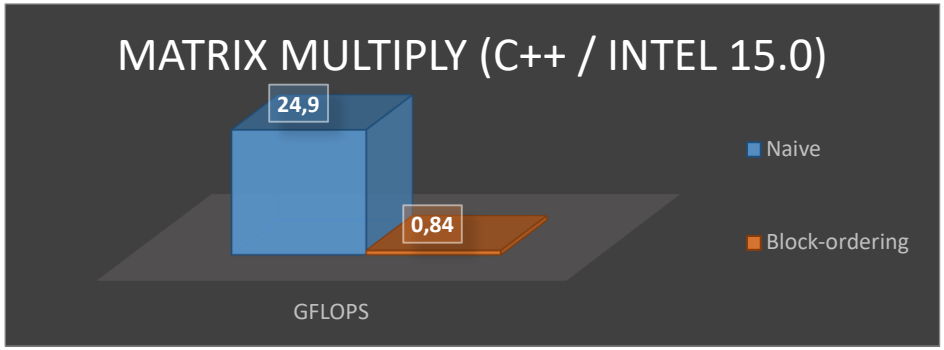
Matrix Multiply

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 ;
            }
        }
    }
} // */
```



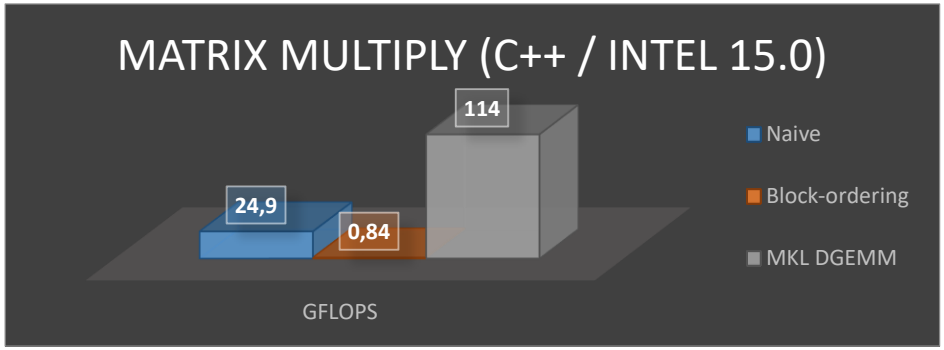
Matrix Multiply

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.

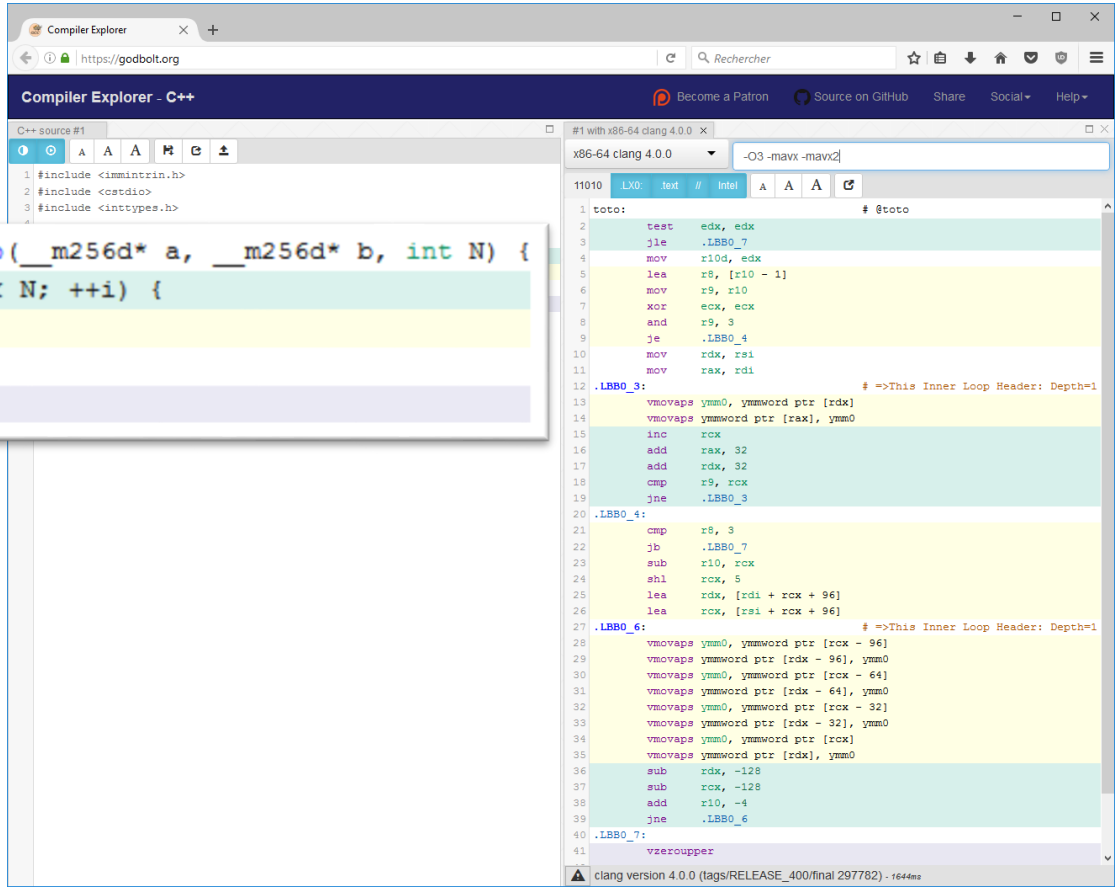
Prefer Vendor-Tuned Libraries

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 ?

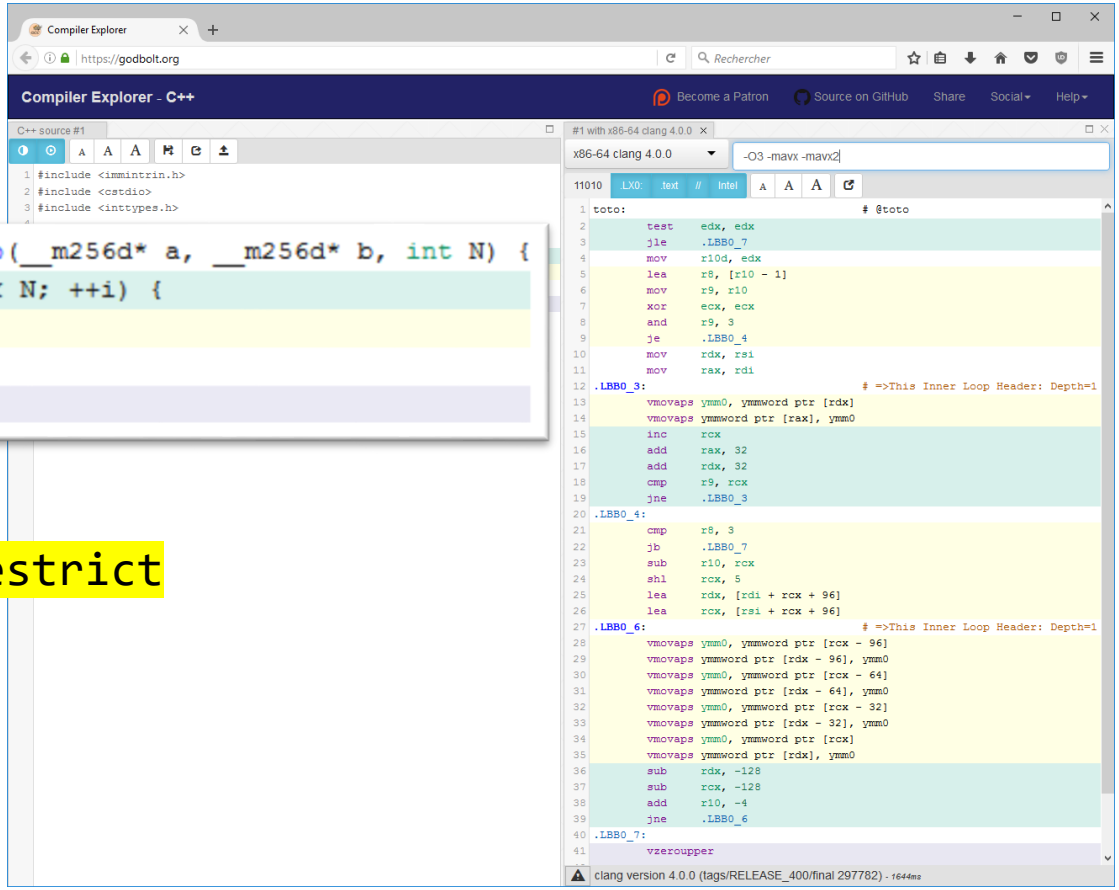
```
extern "C" void toto(__m256d* a, __m256d* b, int N) {
  for(int i = 0; i < N; ++i) {
    a[i] = b[i];
  }
}
```



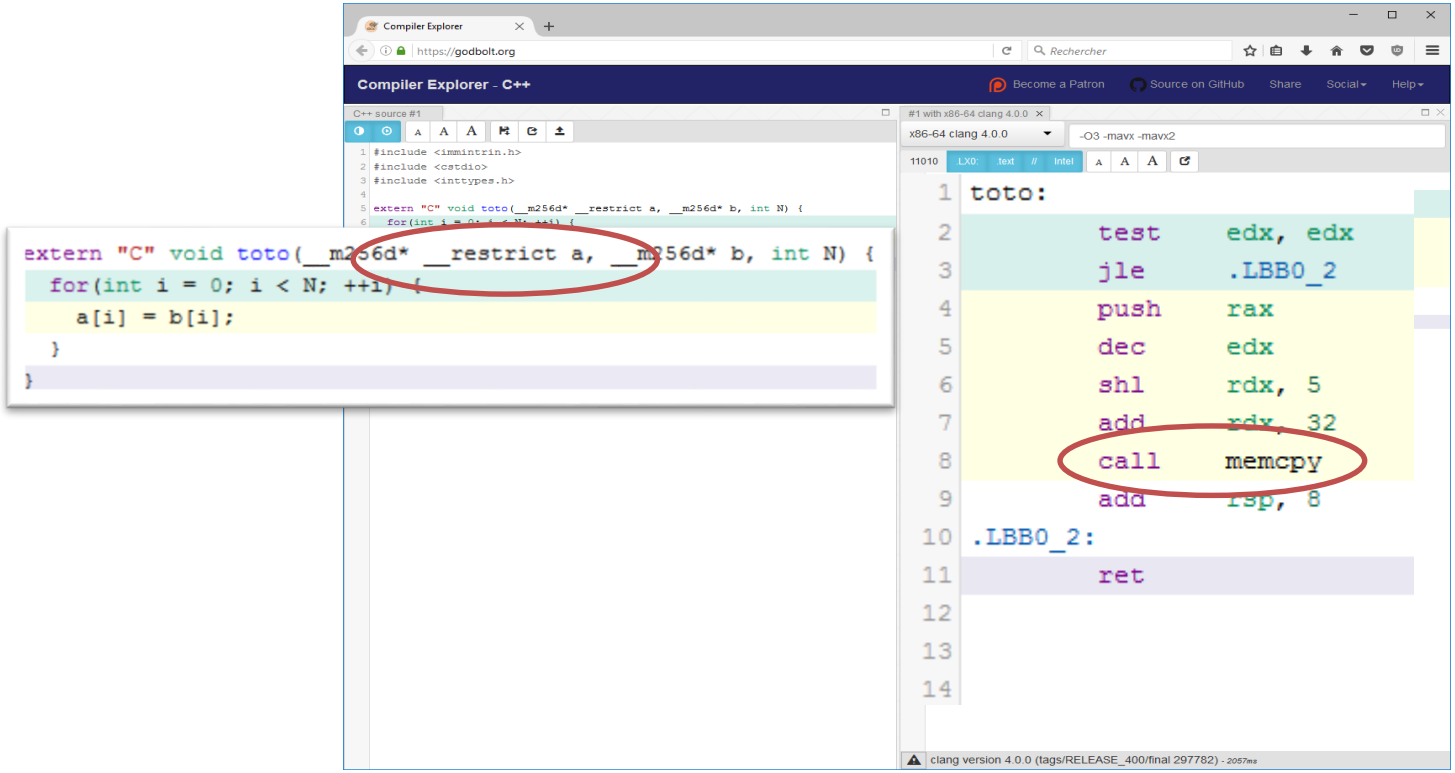
Want to do it anyway ?

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

Forgot __restrict

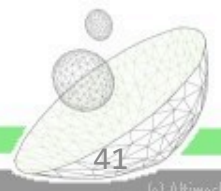


Want to do it anyway ?



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

HYBRID VECTOR LIBRARY



Decorrelate Compute from Data

DATA

Data storage is implicit. Enables full cache awarenes in implementations

Developers express algorithm with a few constaints 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

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

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

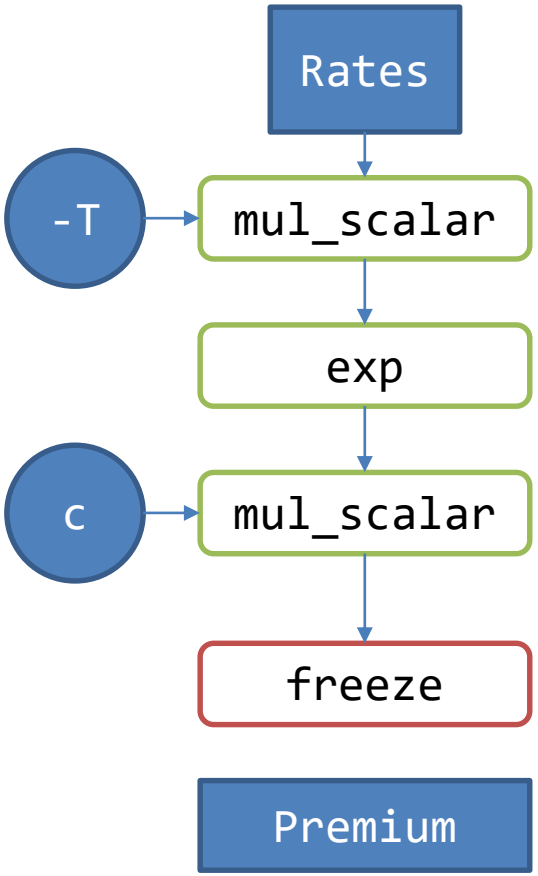
```
double func(int nsimul, double T, double c)
{
  hybridvector* hv ;
  hvl_create(&hv, nsimul);        // allocate "vector"

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

  hvl_mul_scalar(hv, -T);         // hv [i] <- hv [i] * (-T)
  hvl_apply_exp(hv);              // hv [i] <- exp (hv [i])
  hvl_mul_scalar(hv, c);          // hv [i] <- c * hv [i]

  frozenvector* fv;
  double PV;

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

  hvl_destroy(hv);                // clean-up
  hvl_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

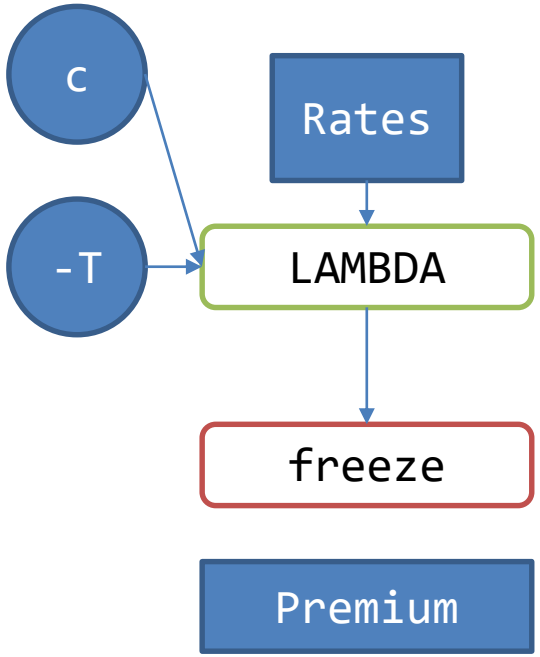
```
double func(int nsimul, double T, double c)
{
  hybridvector* hv ;
  hvl_create(&hv, nsimul);        // allocate "vector"

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

  hvl_mul_scalar(hv, -T);         // hv [i] <- hv [i] * (-T)
  hvl_apply_exp(hv);              // hv [i] <- exp (hv [i])
  hvl_mul_scalar(hv, c);          // hv [i] <- c * hv [i]

  frozenvector* fv;
  double PV;

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

  hvl_destroy(hv);                // clean-up
  hvl_frozen_destroy(fv);

  return PV;
}
```



Example

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

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

  generate_rates(hv, nsimul);

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

  frozenvector* fv;
  double PV;

  hvl_freeze(&fv, hv);
  hvl_frozen_avg(fv, &PV);

  hvl_destroy(hv);
  hvl_frozen_destroy(fv);

  return PV;
}
```

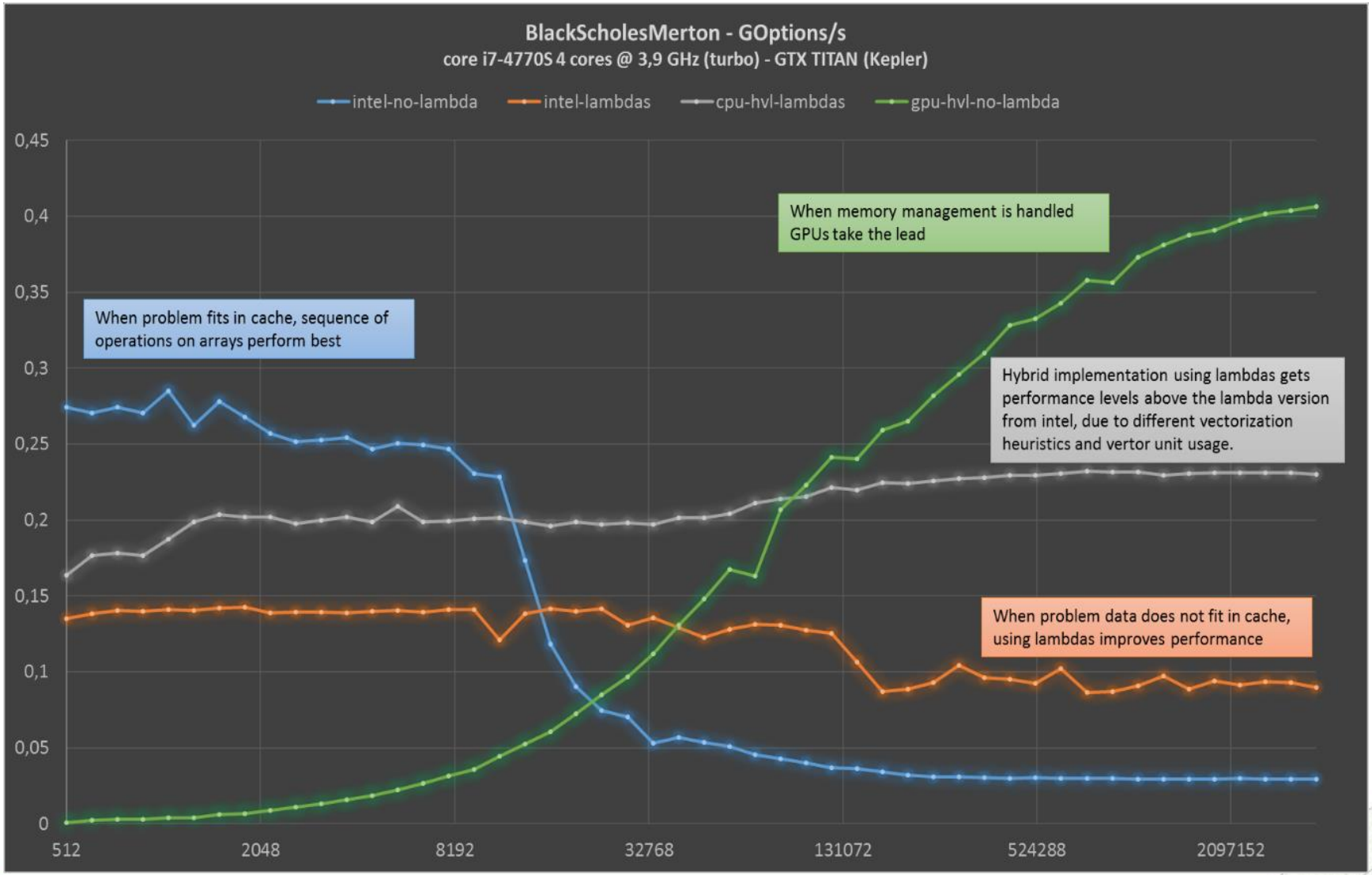


Variety of Implementations

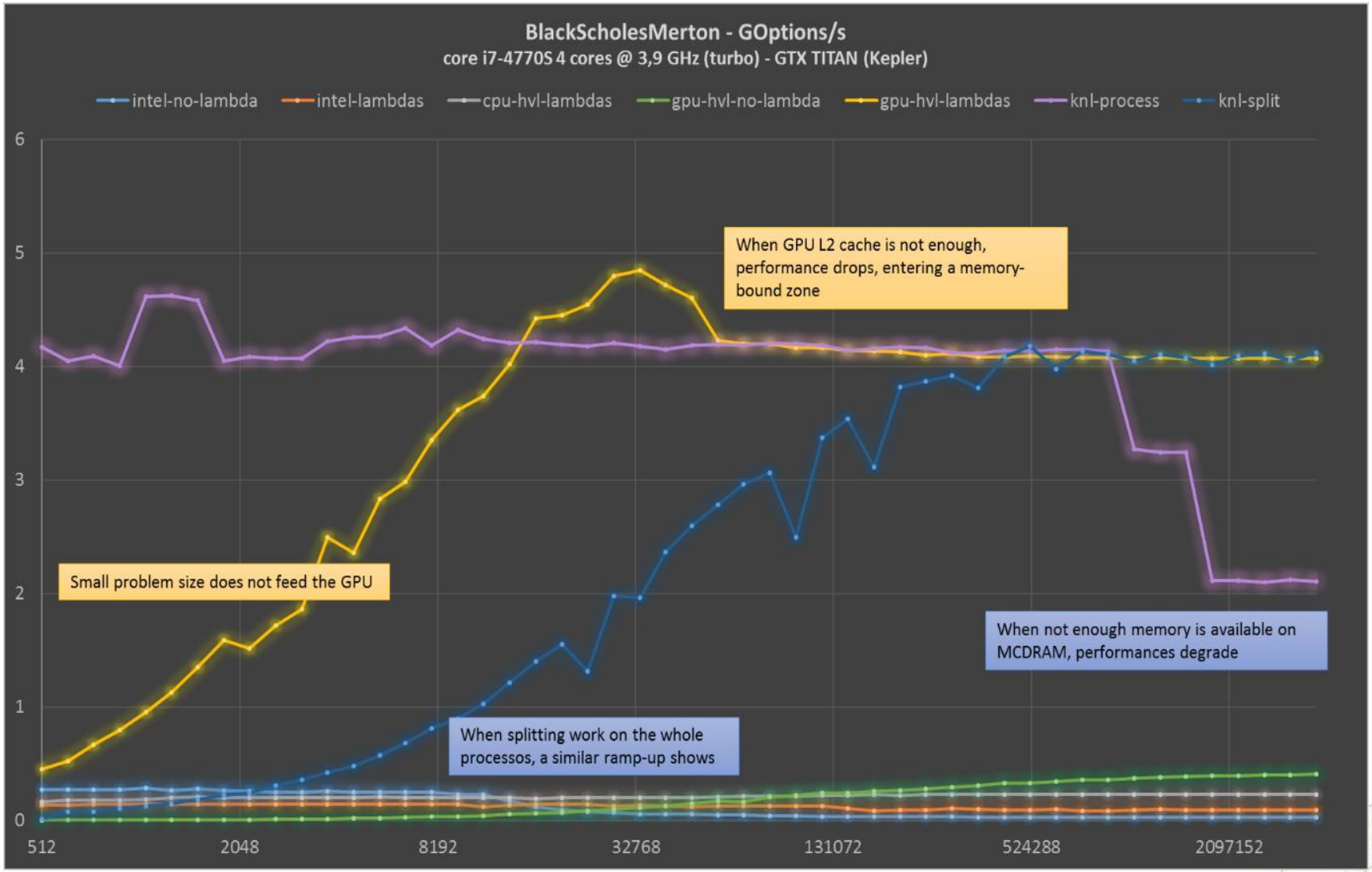
- Using this API, algorithm and runtime implementation are decoupled
- Implicit data allocation allows alternate hardware with no code change
- Computations don’t need to occur when scheduled: they are solely needed at freeze
  - DAG is constructed and evaluated at freeze

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

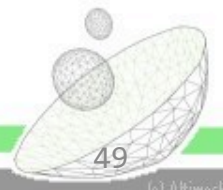
Performance On Black-Scholes Options



Performance On Black-Scholes Options

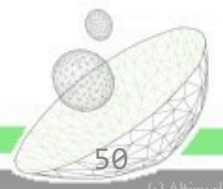


SUMMARY



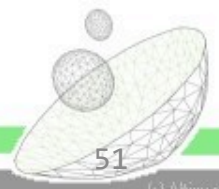
Take Home Message

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



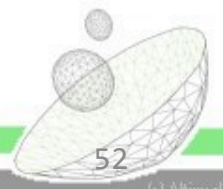
Take Home Message

- 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



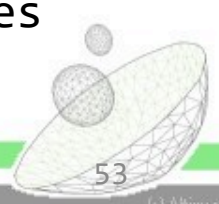
Take Home Message

- 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 !

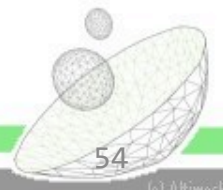


Take Home Message

- 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

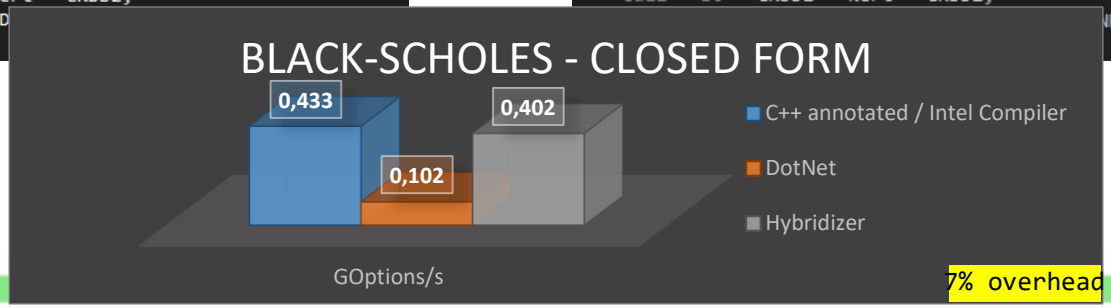
Benchmark-Level Performances

C#

```
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 - CNDD
}
```

C++

```
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;
                                    NDD1) ;
```



Extended features

Virtual Functions

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

Generics

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

Object oriented programming productivity maintained …

… And overhead can be removed