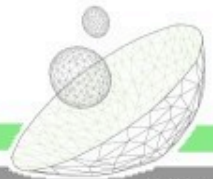


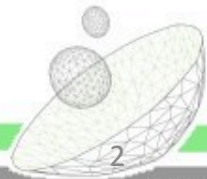
Altimesh Hybridizer™

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



Finance and Regulation

- Financial institutions are very creative
 - Derivative products ecosystem grows constantly
 - Some players introduce new product types to leverage corner unregulated financial traits [e.g. Subprimes]
- Every big financial event yields new regulations
 - More stress scenarios [Too big to fail]
 - More complex financial quantitative models [Liquidity]
 - Higher number of simulations [unlikely systemic events]
- Quant analysts need to (re-)design quant libraries constantly
 - New models need to be developed, tested and integrated in existing system
 - Performance is getting critical: from thousands to millions of simulations - same power envelope ?
 - Code optimization gets low priority: following changes implied by regulators is already a heavy burden

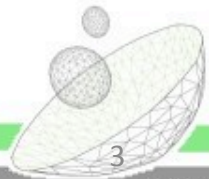


Processor Ecosystem

- Processors have changed

| year | 2000 | 2014 | 2013 | 2012 |
|------------------------|------------------|--------------|--------------|--------------|
| processor | <i>Pentium 4</i> | Xeon E5-v3 | Xeon PHI | Kepler |
| core frequency (GHz) | 3,8 | 2,3 | 1,24 | 0,745 |
| vector unit size (DP) | 1 | 4 | 8 | 32 |
| pipelines / core | 1 | 2 | 1 | 2 |
| contexts | 1 | 2 | 4 | 4 |
| core count | 1 | 18 | 61 | 15 |
| FMA | 1 | 2 | 2 | 2 |
| Peak scalar GFLOPS | 3,8 | 165,6 | 151,28 | 22,35 |
| Peak GFLOPS (DP) | 3,8 | 662,4 | 1210,24 | 1430,4 |
| SIMD/SIMT ratio | 1 | 4 | 8 | 64 |
| Bandwidth (R/W) | 4,26 | 68 | 352 | 288 |
| flop / memop | 7,14 | 77,93 | 27,51 | 39,73 |
| Bandwidth / core | 4,26 | 3,78 | 5,77 | 19,20 |

- Frequency drops, Core count / vector unit explodes
- Most problems get memory bound (flop / memop > 25)
- Multithreading is not the only issue (SIMD/SIMT ratio)
- Keeping-up with technology changes requires significant software development effort and training

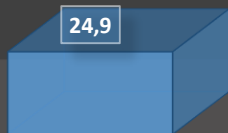


Matrix Multiply

Naive Matrix Multiply

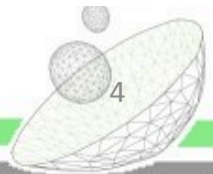
```
#pragma omp parallel for
for (int i = 0 ; i < rowsA ; ++i)
{
    for (int j = 0 ; j < colsB ; ++j)
    {
        double d = 0.0 ;
        for (int k = 0 ; k < colsA ; ++k)
        {
            d += A[i * colsA + k] * B[k * colsB + j] ;
        }
        C[i * colsB + j] = d ;
    }
} // */
```

MATRIX MULTIPLY (C++ / INTEL 15.0)



■ Naive

GFLOPS



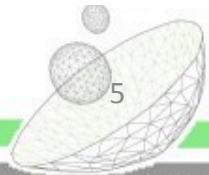
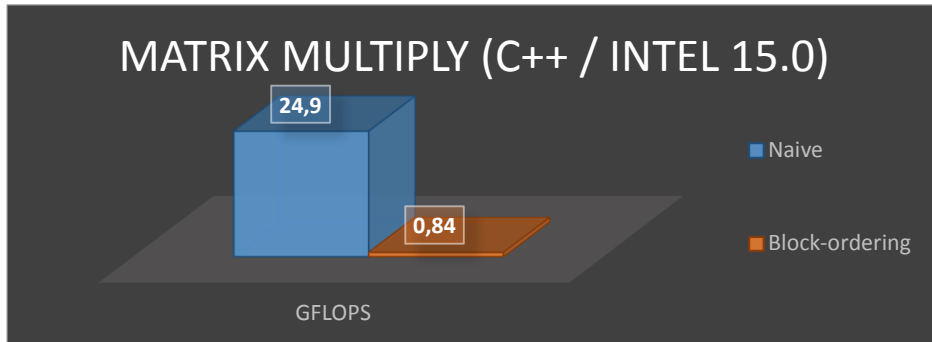
Matrix Multiply

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 ;
    }
} // */
```

Block-accumulation (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 ;
            }
        }
    }
} // */
```



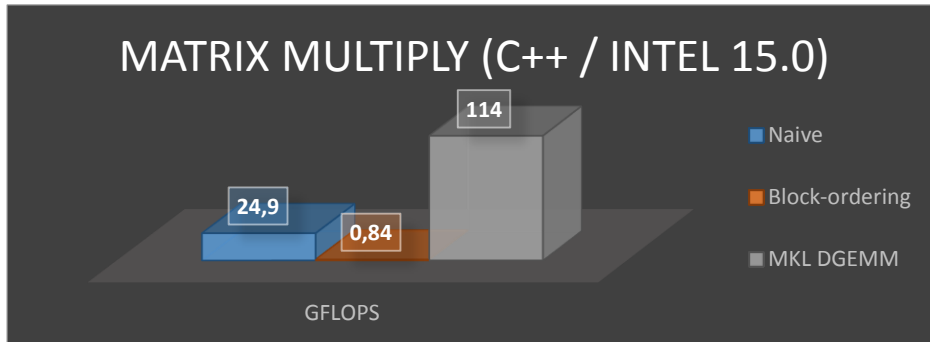
Matrix Multiply

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 ;
    }
} // */
```

Block-accumulation (better cache behavior?)

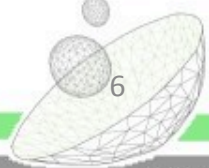
```
#pragma omp parallel for default(none) firstprivate(rowsA, colsA, A, colsB, B, C)
for(int ibk = 0 ; ibk < rowsA / SIZE ; ++ibk)
{
    for (int jbk = 0 ; jbk < colsB / SIZE ; ++jbk)
    {
        for (int i = ibk * SIZE ; i < (ibk + 1) * SIZE ; ++i)
        {
            for (int j = jbk * SIZE ;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

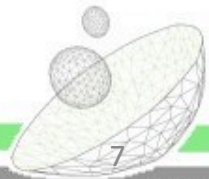
- Vector-unit operations
- Non-temporal write
- Several layers of memory prefetching
- Many corner cases for unaligned sizes, transposes, etc.

Prefer Vendor-Tuned Libraries



Use Vendor-Tuned Libraries

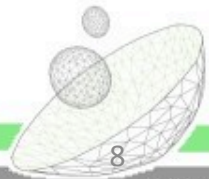
- « What every programmer should know about memory », by *Ulrich Drepper*
 - It takes a lot to write (close to) optimal code
 - Understanding of core components of the system are necessary to get good performance (getting a compute-bound implementation of matrix multiply is hard)
- Micro-architecture evolve
 - AVX means 256 bits operands => new instruction set wrt SSE
 - AVX-2 has more instructions => need to redefine some code (see 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
 - Sometimes better performance writing code to transition from custom code to optimized library
- 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



Key Changes to Embrace

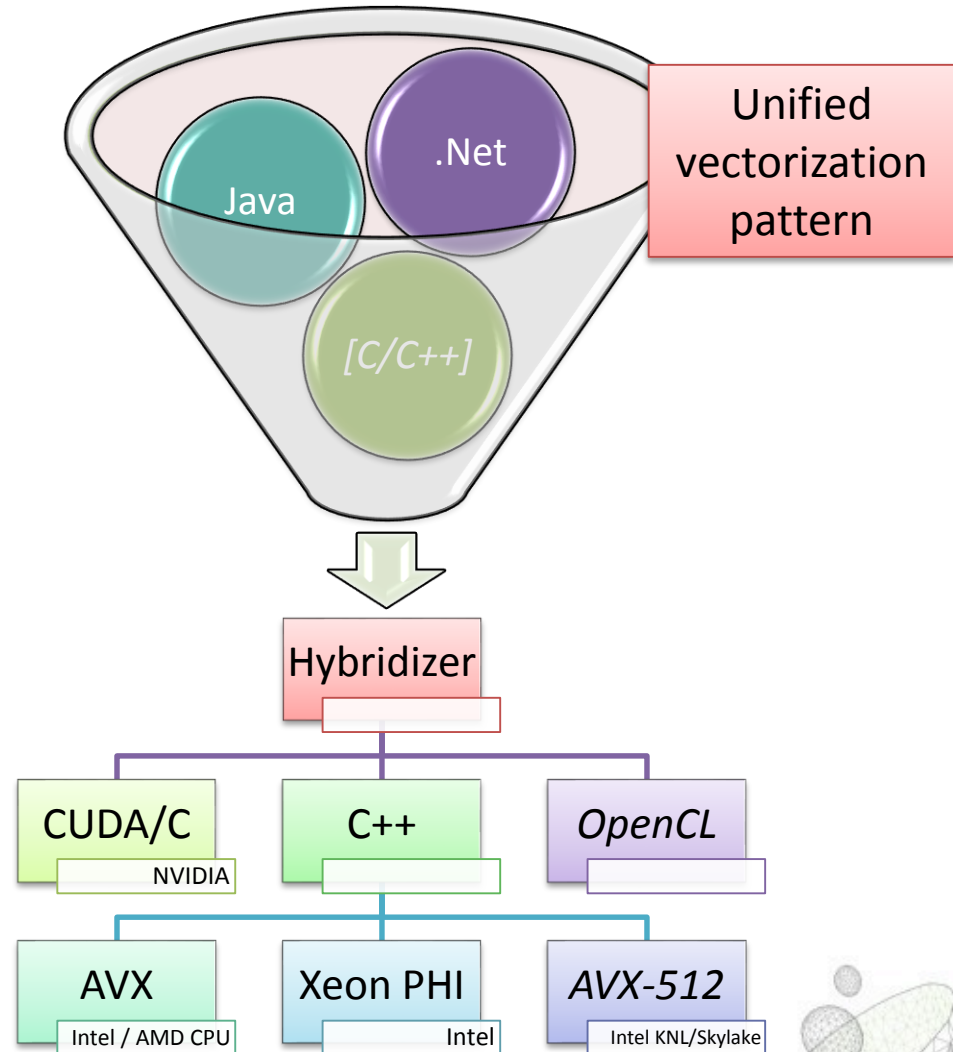
- **Multithread** : core count explode, and frequency stalls or decrease => **not using multithread** will lead to **performance decrease** in the future
- **Vectorize** : vector unit size grows. SIMD/SIMT ratio indicates the relative loss when not *vectorizing* code. AVX-512 will double the fall for Intel x86 architecture.
- **Cache-aware** : flop/memop increase (> 25). Operations need to occur in cache. Large vector operations are memory bound and should be replaced by small vector operations

Hybridizer aims at addressing these challenges with a unified approach



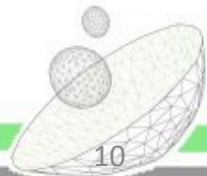
Hybridizer Solution

- Input
 - .Net
 - Java
 - C/C++ (*ongoing developments*)
- Environments:
 - Windows / Linux
- Generate source code
 - CUDA/C for NVIDIA GPU
 - C++ for native platforms
 - Open CL



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

Basic Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE 2015

Analysis Target Analysis Type Collection Log Summary Bottom-up Caller/Callee Top-down Tree Tasks and Frames BlackSch...

Source Assembly

Assembly grouping: Address

| So. | Source | Address | Sour... | Assembly | CPU Time: Total | CPU Time: Self |
|-----|---|-------------|-----------|--|-----------------|----------------|
| 54 | public static class BlackScholesMertonPrimitive | 0x18000591b | Block 46: | | | |
| 55 | | 0x18000591b | 115 | vmulpd ymm4, ymm0, ymmword ptr [r13+0xc20] | 0.002s | 0.002s |
| 56 | | 0x180005924 | 115 | vmulpd ymm0, ymm7, ymmword ptr [r13+0xba0] | 0.095s | 0.095s |
| 57 | | 0x18000592d | 115 | vmovupd ymmword ptr [r13+0x500], ymm4 | 0.000s | 0.000s |
| 58 | const double A3 = 1.71477937; | 0x180005936 | 225 | vmovupd ymmword ptr [r13+0x1c40], ymm4 | 0.028s | 0.028s |
| 59 | const double A4 = -1.82125978; | 0x18000593f | 768 | call 0x18000aff8 < svml_expd> | 0.016s | 0.016s |
| 60 | const double A5 = 1.330274429; | 0x180005944 | Block 47: | | | |
| 61 | const double RSQR2PI = 0.39894228040143267793994605993438; | 0x180005944 | 115 | vmulpd ymm4, ymm0, ymmword ptr [r13+0xc60] | 0.003s | 0.003s |
| 62 | | 0x18000594d | 115 | vmovupd ymmword ptr [r13+0x4e0], ymm4 | 0.101s | 0.101s |
| 63 | public static double CND(double d) | 0x180005956 | 225 | vmovupd ymmword ptr [r13+0x1c60], ymm4 | 0.017s | 0.017s |
| 64 | { | 0x18000595f | 763 | vandpd ymm5, ymm9, ymmword ptr [r13+0x740] | 0.028s | 0.028s |
| 65 | double K = 1.0 / (1.0 + 0.2316419 * Math.Abs(d)); | 0x180005968 | 88 | vfmadd213pd ymm5, ymm8, ymm12 | 0.003s | 0.003s |
| 66 | | 0x18000596d | | vdivpd ymm4, ymm12, ymm5 | | |
| 67 | double cnd = RSQR2PI * Math.Exp(-0.5 * d * d) | 0x180005971 | 225 | vmovupd ymmword ptr [r13+0x1780], ymm4 | 0.554s | 0.554s |
| 68 | * | 0x18000597a | 763 | vandpd ymm4, ymm13, ymm9 | 0.036s | 0.036s |
| 69 | (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))))); | 0x18000597f | | vfmadd213pd ymm4, ymm8, ymm12 | | |
| 70 | | 0x180005984 | | vdivpd ymm4, ymm12, ymm4 | | |
| 71 | | 0x180005988 | 225 | vmovupd ymmword ptr [r13+0x17a0], ymm4 | 0.339s | 0.339s |
| 72 | if (d > 0) | 0x180005991 | 763 | vandpd ymm4, ymm9, ymmword ptr [r13+0x6e0] | 0.020s | 0.020s |
| 73 | cnd = 1.0 - cnd; | 0x18000599a | 88 | vfmadd213pd ymm4, ymm8, ymm12 | 0.002s | 0.002s |
| 74 | | 0x18000599f | | vdivpd ymm4, ymm12, ymm4 | | |
| 75 | return cnd; | 0x1800059a3 | 225 | vmovupd ymmword ptr [r13+0x17c0], ymm4 | 0.597s | 0.597s |
| 76 | } | 0x1800059ac | 763 | vandpd ymm4, ymm15, ymm9 | 0.043s | 0.043s |
| 77 | | 0x1800059b1 | 88 | vfmadd213pd ymm4, ymm8, ymm12 | 0.001s | 0.001s |
| 78 | [Kernel] | 0x1800059b6 | | vdivpd ymm4, ymm12, ymm4 | | |
| 79 | public static void CallPut(out double call, double put, double spot, double | 0x1800059ba | 225 | vmovupd ymmword ptr [r13+0x17e0], ymm4 | 0.650s | 0.650s |
| 80 | double sigmaSqrtI = sigma * Math.Sqrt(maturity); | 0x1800059c3 | 763 | vandpd ymm4, ymm9, ymmword ptr [r13+0x680] | 0.049s | 0.049s |
| 81 | double d1 = (Math.Log(spot / strike) + (rate + sigma * sigma / 2.0) * maturi | 0x1800059cc | 88 | vfmadd213pd ymm4, ymm8, ymm12 | 0.004s | 0.004s |
| 82 | double d2 = d1 - sigmaSqrtI; | 0x1800059d1 | | vdivpd ymm4, ymm12, ymm4 | | |
| 83 | double kert = strike * Math.Exp(-rate * maturity); | 0x1800059d5 | 225 | vmovupd ymmword ptr [r13+0x1800], ymm4 | 0.598s | 0.598s |
| 84 | double CNDd1 = CND(d1); | 0x1800059de | 763 | vandpd ymm4, ymm14, ymm9 | 0.037s | 0.037s |
| 85 | double CNDd2 = CND(d2); | 0x1800059e3 | | vfmadd213pd ymm4, ymm8, ymm12 | | |
| 86 | call = spot * CNDd1 - kert * CNDd2; | 0x1800059e8 | | vdivpd ymm4, ymm12, ymm4 | | |
| 87 | put = kert * (1.0 - CNDd2) - spot * (1.0 - CNDd1); | 0x1800059ec | 225 | vmovupd ymmword ptr [r13+0x1820], ymm4 | 0.556s | 0.556s |
| 88 | | 0x1800059f5 | 763 | vandpd ymm4, ymm9, ymmword ptr [r13+0x620] | 0.027s | 0.027s |
| 89 | } | 0x1800059fe | | vfmadd213pd ymm4, ymm8, ymm12 | | |
| 90 | | 0x180005a03 | | vdivpd ymm4, ymm12, ymm4 | | |
| 91 | [Kernel] | 0x180005a07 | 225 | vmovupd ymmword ptr [r13+0x1840], ymm4 | 0.586s | 0.586s |
| 92 | public static void CallPut(int index, double[] call, double[] put, double spot, | 0x180005a10 | 763 | vandpd ymm4, ymm9, ymmword ptr [r13+0x5e0] | 0.038s | 0.038s |
| 93 | { | 0x180005a19 | 88 | vfmadd213pd ymm4, ymm8, ymm12 | 0.001s | 0.001s |
| 94 | double sigmaSqrtI = sigma * Math.Sqrt(maturity); | 0x180005a1e | 163 | vdivpd ymm4, ymm12, ymm4 | 0.001s | 0.001s |
| 95 | double d1 = (Math.Log(spot / strike) + (rate + sigma * sigma / 2.0) * maturi | 0x180005a22 | 225 | vmovupd ymmword ptr [r13+0x1860], ymm4 | 0.566s | 0.566s |
| 96 | double d2 = d1 - sigmaSqrtI; | 0x180005a2b | 223 | xor al, al | 0.041s | 0.041s |
| 97 | double kert = strike * Math.Exp(-rate * maturity); | 0x180005a2d | | xor r14d, r14d | | |
| 98 | double CNDd1 = CND(d1); | 0x180005a30 | | vmovupd ymmword ptr [r13+0xd20], ymm7 | | |
| 99 | double CNDd2 = CND(d2); | 0x180005a39 | | vmovdqu ymmword ptr [r13+0xd40], ymm10 | | |

Standard System.Math methods

Mapped on Intel SVML

AVX2 instructions

Scalar C# source File

Selected 14 row(s):

Highlighted 0 row(s):

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;

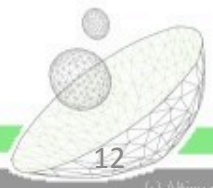
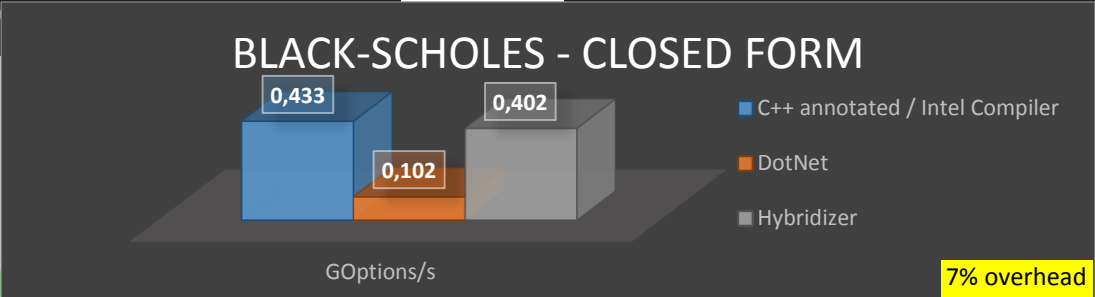
    double K = 1.0 / (1.0 + 0.2316419 * fabs(d));

    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

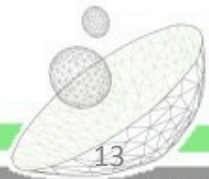
- Interfaces / abstract classes and inheritance is supported
- Underlying implementation is a function-table

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



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;
        }
    }
}
```



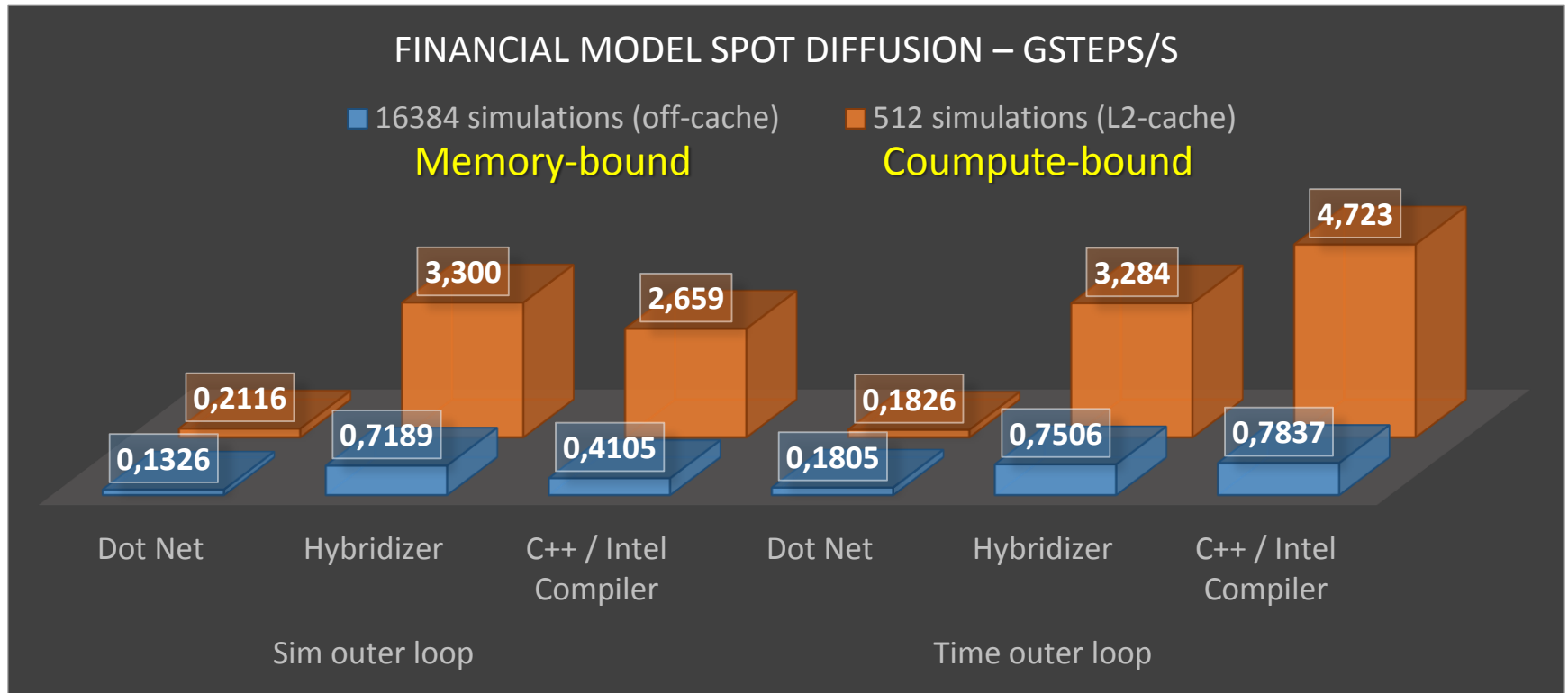
C++ source code with annotations
(two outer loop configurations)

```
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 ;
        }
    }
}

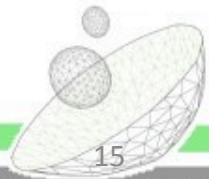
#pragma omp parallel for
for (int th = 0 ; th < 8 ; ++th)
{
    int simFrom = th * simCount / 8 ;
    int simTo = (th+1) * simCount / 8 ;
    for (int time = 0 ; time < datesCount ; ++time)
    {
        double* lnS = logSpot + (simCount * time) ;
        const double* brow = brownian + (time * simCount)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



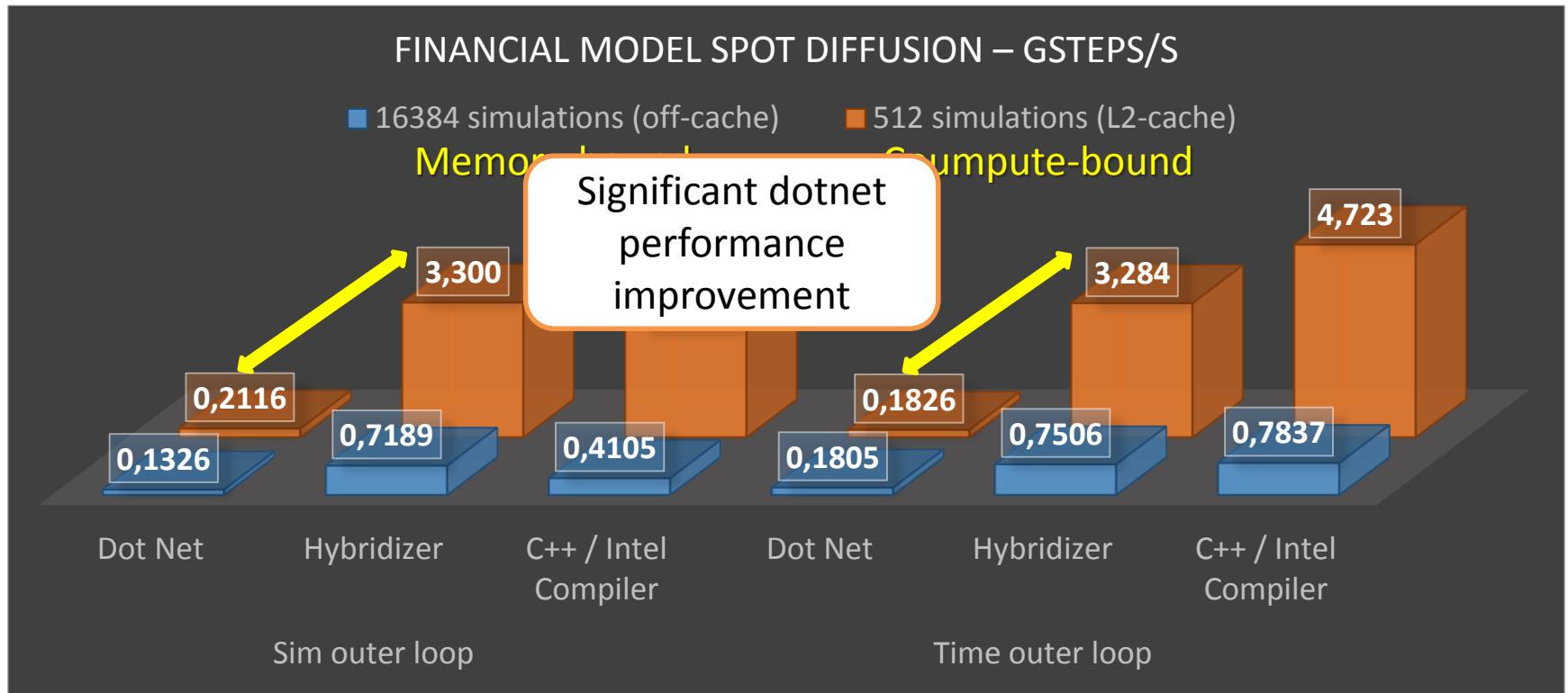
Black-Scholes-Merton Diffusion



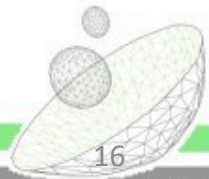
- Comparing object-oriented code, with generics, processed by Hybridizer
- with hand-written optimized C++ code compiled with Intel Composer 2015



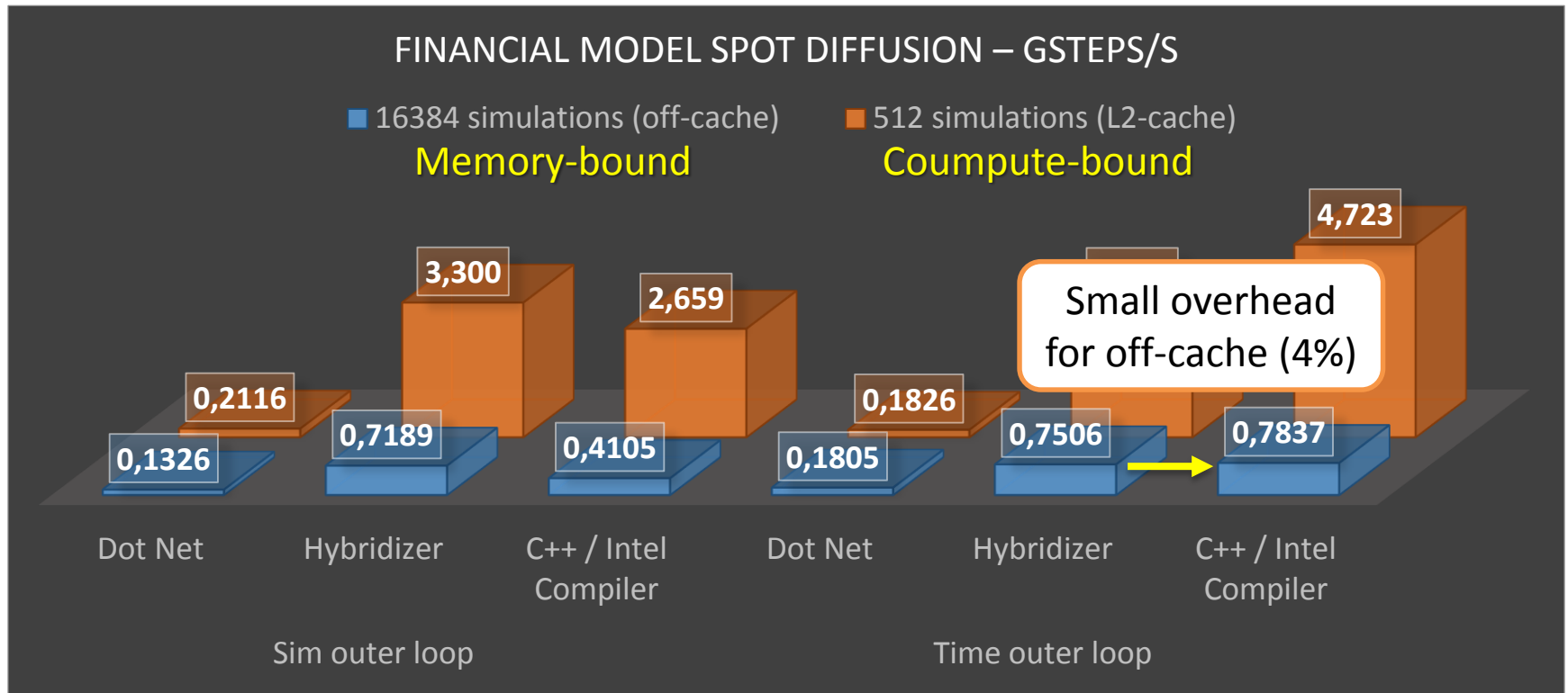
Black-Scholes-Merton Diffusion



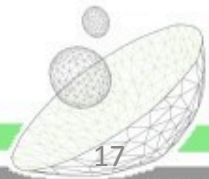
- Hybridizer greatly improves dotnet performance: **5x to 18x**
- Object oriented programming preserved: single version of source code, reduces operational risk / testing costs.



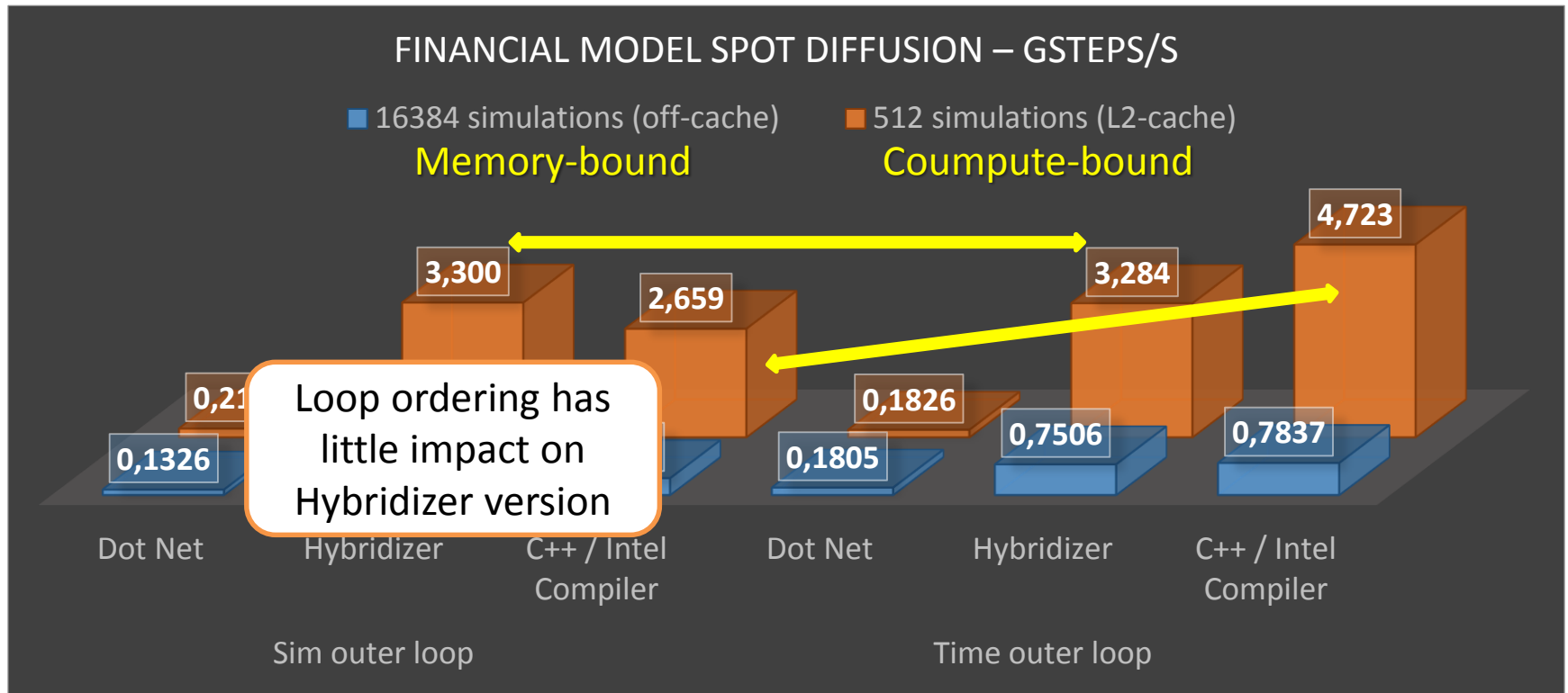
Black-Scholes-Merton Diffusion



- 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**)

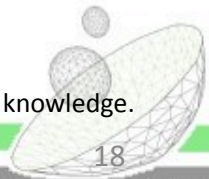


Black-Scholes-Merton Diffusion



- 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%)

NOTE: cache-locality and outer-loop selection has a **10x** impact on performance. Writing optimized C++ code requires significant effort and knowledge.



Conclusions

- Shortened development cycles
 - Single version of source code – with « managed » languages
 - Integrates with Debuggers and Profilers
- State-of-the art performances
 - Software development flexibility without performance costs
 - Close to Benchmark (>90%) for compute and memory bound problems
- Embrace micro-architecture changes
 - Hybridizer is AVX-512 ready – simply recompile ?

<http://www.altimesh.com>

