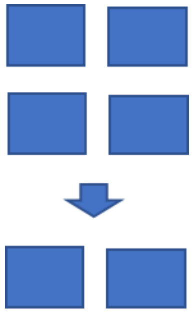
Fast C++ by using SIMD Types with Generic Lambdas and Filters

ANDREW DRAKEFORD

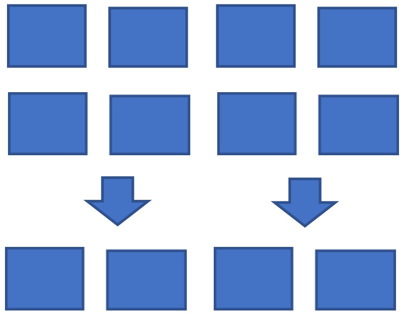
Cppcon
The C++ Conference
2022
September 12th-16th

Single Instruction Multiple Data



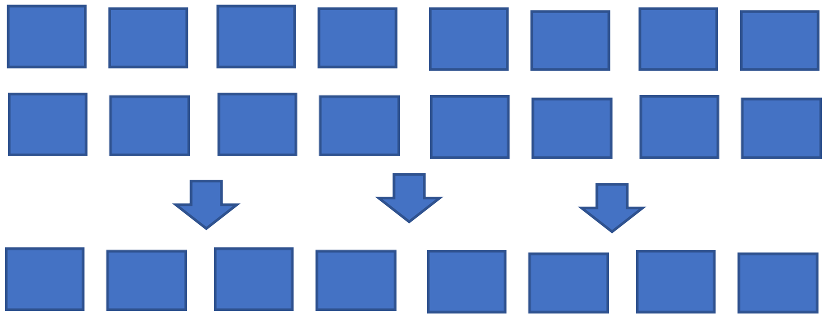
XMM register

SSE2



YMM register

AVX2



ZMM register

AVX512

Intrinsics

Intrinsic data types and functions store and operate on data meant for vectorised registers

_m256d (256 bits holds 4 doubles)

_mm256_add_pd(__m256d a, __m256d b) element-wise add two _mm256

SIMD Wrapper

```
Vec4f a(0.0f, 0.5f, 1.0f, 1.5f); // define vector
Vec4f b = sin(a); // sin  functiom
// b = ( 0.0000f , 0.4794f , 0.8415f , 0.9975f )
```

- SIMD wrapper hides complex intrinsic functions and data types under common names and operators and provides math operators and functions
- boost simd .. Eve
- std::simd
- VCL Vector Class Library

DR^3 Basics

DR^3 Basics

- Large Vector Type
- Lambda utilities
- Filters & Views

Vector VecXX

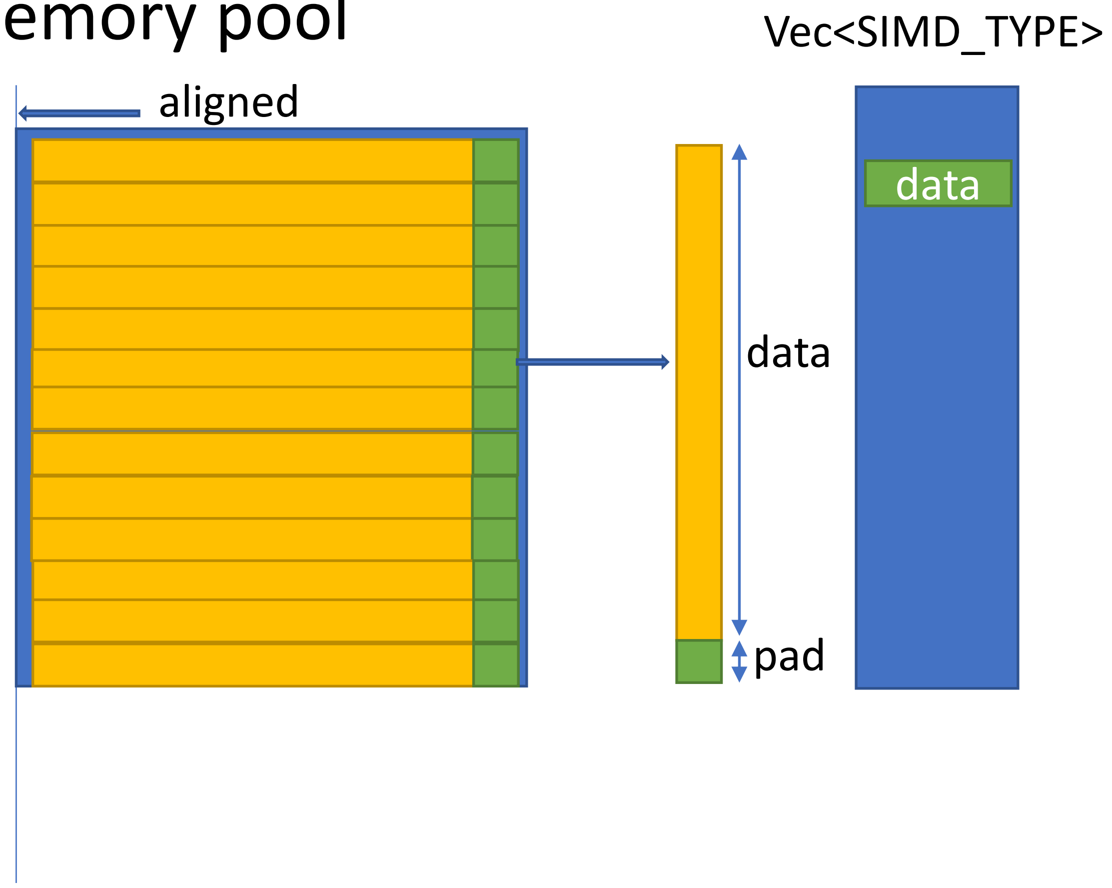
VecXX

- Memory managed vector type
- Supports math functions and operations
- Contiguous, aligned and padded
- Can change the scalar type and instruction set
- Substitutable for scalar type so we drop into existing code to make it vectorised

VecXX Utility

Vec

Memory pool



Math Operators and Functions

vec_A = vec_B+ vec_C

vec_A >= vec_B

vec_A = sin(vec_C)

Black-Scholes Example

Video

$$
\begin{aligned}
C(S_t, t) &= N(d_1)S_t - N(d_2)Ke^{-r(T-t)} \\
d_1 &= \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right] \\
d_2 &= d_1 - \sigma\sqrt{T-t}
\end{aligned}
$$

Pros & Cons of VecXX

What's Good?

- Easy to use.
- Very good memory layout, contiguous, aligned, predictable
- Memory allocation fast (custom allocation)

What's bad ?

- Traverses lots of memory to perform simple actions
- Need for custom re-writes of critical areas, or existing 3rd party library functions

The Problem : low intensity operations define the interface

We need to do more work but keep it open. A kind of user definable callable that’s going to be easy to use and agnostic of the wrapper type.

The Problem : low intensity operations define the interface

We need to do more work but keep it open. A kind of user definable callable that's going to be easy to use and agnostic of the wrapper type.

```
auto theAnswer  = []( auto x) { return expression_goes_here;};
```

When generic lambda functions are instantiated with a good SIMD wrapper

They have brutal performance characteristics.

DR^3 Lambda functions are generic

When we instantiate the lambda with a SIMD wrapper, we generate code that uses vectorised instructions.

A lambda function will only be instantiated

if the SIMD wrapper supports the function names and operations used.

To change instruction sets, the generic lambda function doesn't need to change we just change what type its instantiated with.

When generic lambda functions are instantiated with a good SIMD wrapper have brutal performance characteristics.

Essential Implementation Utilities take generic lambda’s and vecxx’s as arguments They apply operations specified by lambda over a vector.

Load	load data from memory into a register
Apply	apply the lambda to the register
Store	store the results back to memory

DR^3 Utilities

Transform

Branching

Filters and Views (contiguous, aligned, indexed and by value)

Performance
&
Simplicity

DR^3 Lambda Functions

- Transform
- Reduce/accumulate
- TransformReduce

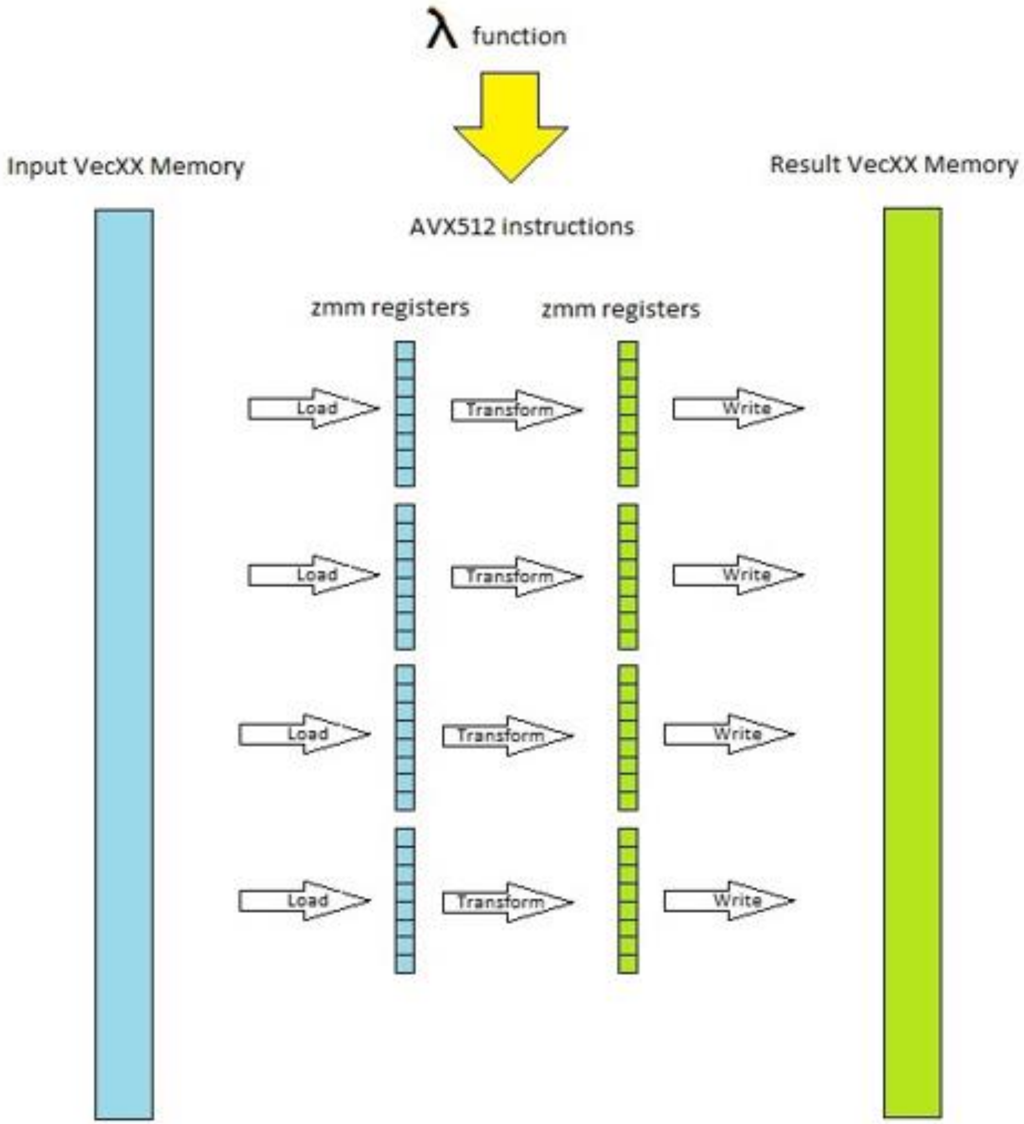
Approach

Write simple functions and compare with STL equivalents

- transform — memcpy
- reduce — max element
- transformReduce — inner product

- All compiled in same cpp file. So same project settings applied .

transform


λ function
Input VecXX Memory
Result VecXX Memory
AVX512 instructions
zmm registers
zmm registers
Load
Transform
Write
Load
Transform
Write
Load
Transform
Write
Load
Transform
Write

```cpp
std::vector<FloatType>  c(SZ, VecXX::SCALA_TYPE(0.5));
VecXX test(v);
auto cpyLambda = [](const auto& rhs) { return rhs; };

for (long i = 0; i < loop; ++i)
{
    auto res = transform( cpyLambda, test);
}
```

transform

Moving packed doubles via zmm register

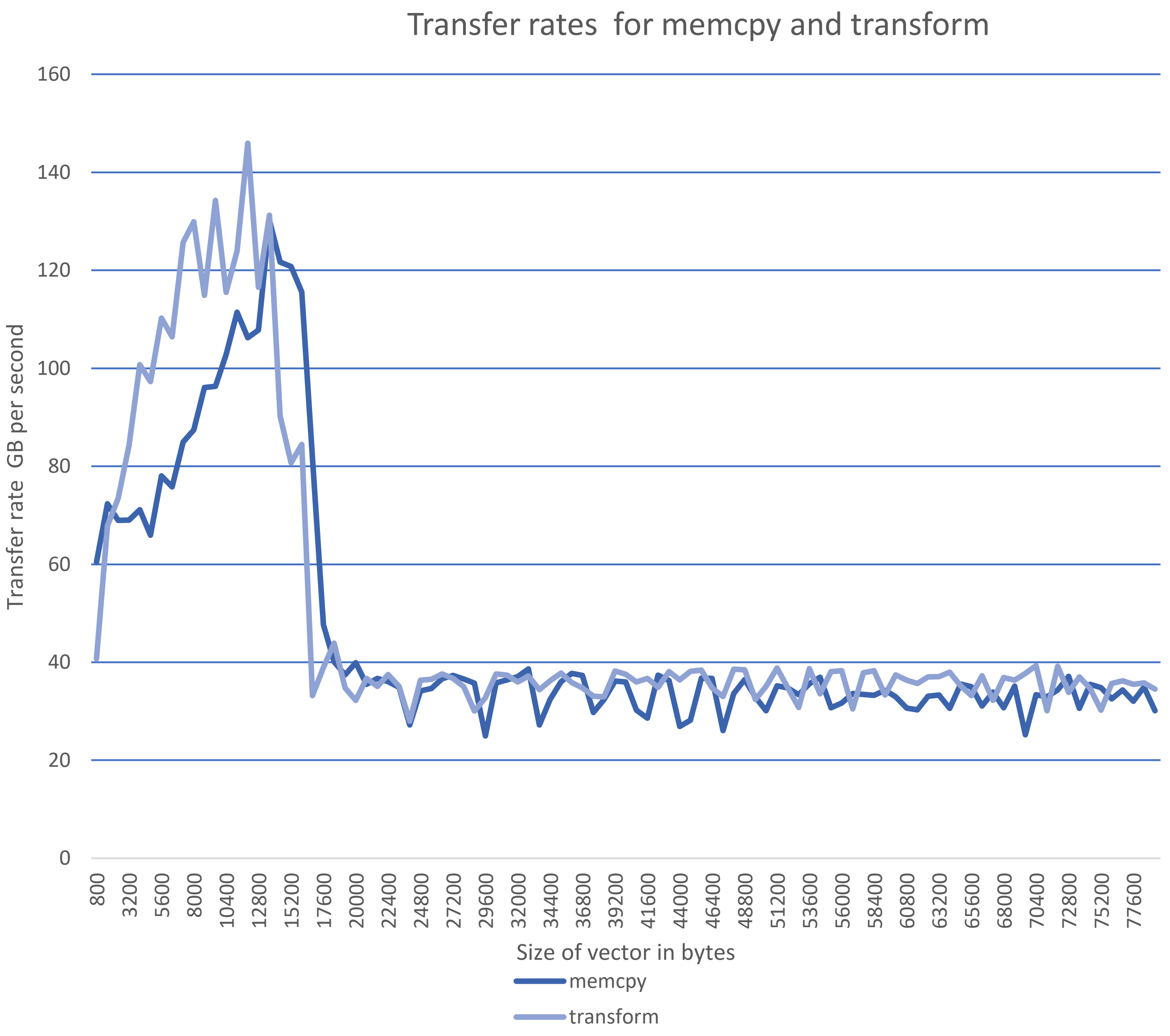
```asm
vmovupd      zmm0,zmmword ptr [rdx+rcx-40h]
vmovupd      zmmword ptr [rcx-40h],zmm0
vmovupd      zmm1,zmmword ptr [rdx+rcx]
vmovupd      zmmword ptr [rcx],zmm1
vmovupd      zmm0,zmmword ptr [rdx+rcx+40h]
vmovupd      zmmword ptr [rcx+40h],zmm0
vmovupd      zmm1,zmmword ptr [rdx+rcx+80h]
vmovupd      zmmword ptr [rcx+80h],zmm1
lea          rcx,[rcx+100h]
sub          r9,1
jne          ApplyTransformUR_X<Vec8d,
 <lambda_37c86e8c431697a3720d3d806be773b5> >+0C0h (07FF770A71F20h)
```

memcpy

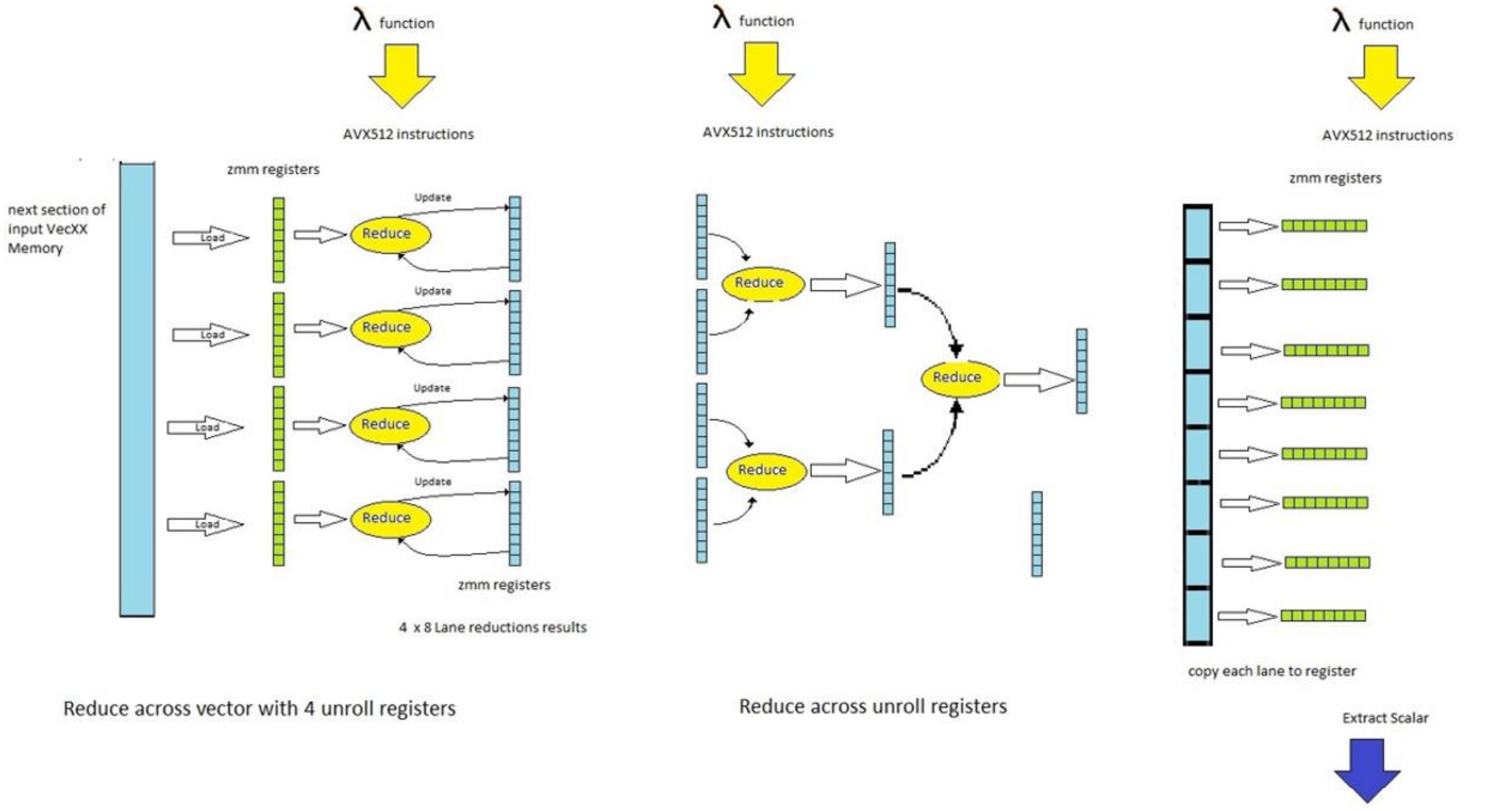
```
vmovdqu     ymm1,ymmword ptr [rdx]     ≤ 1ms elapsed
vmovdqu     ymm2,ymmword ptr [rdx+20h]
vmovdqu     ymm3,ymmword ptr [rdx+40h]
vmovdqu     ymm4,ymmword ptr [rdx+60h]
vmovdqa     ymmword ptr [rcx],ymm1
vmovdqa     ymmword ptr [rcx+20h],ymm2
vmovdqa     ymmword ptr [rcx+40h],ymm3
vmovdqa     ymmword ptr [rcx+60h],ymm4
vmovdqu     ymm1,ymmword ptr [rdx+80h]
vmovdqu     ymm2,ymmword ptr [rdx+0A0h]
vmovdqu     ymm3,ymmword ptr [rdx+0C0h]
vmovdqu     ymm4,ymmword ptr [rdx+0E0h]
vmovdqa     ymmword ptr [rcx+80h],ymm1
vmovdqa     ymmword ptr [rcx+0A0h],ymm2
vmovdqa     ymmword ptr [rcx+0C0h],ymm3
vmovdqa     ymmword ptr [rcx+0E0h],ymm4
add         rcx,100h
add         rdx,100h
sub         r8,100h
cmp         r8,100h
jae         00007FFFD47B14C0
```

```
for (long i = 0; i < loop; ++i)
{
    std::memcpy(cp, vp, bufferSz);
}
```

Performance



reduce



λ function
AVX512 instructions
zmm registers
next section of input VecXX Memory
Load
Reduce
Update
zmm registers
4 x 8 Lane reductions results
Reduce across vector with 4 unroll registers
λ function
AVX512 instructions
Reduce
Reduce across unroll registers
λ function
AVX512 instructions
zmm registers
copy each lane to register
Extract Scalar

max element reduce

```
auto mxDbl = [](auto lhs, auto rhs) { return iff(lhs > rhs, lhs, rhs); };
for (long l = 0; l < TEST_LOOP_SZ; l++)
{
    res = reduce(test,mxDbl );
}
```

```
00007FF62EF02740  vmovupd   zmm0,zmmword ptr [r11]
00007FF62EF02746  lea       r11,[r11+100h]
00007FF62EF0274D  lea       r10,[r10+100h]
00007FF62EF02754  vcmppd    k1,zmm1,zmm0,6
00007FF62EF0275B  vmovapd   zmm0{k1},zmm1
00007FF62EF02761  vmovupd   zmm1,zmm0
00007FF62EF02767  vmovupd   zmm0,zmmword ptr [r10-140h]
00007FF62EF0276E  vcmppd    k1,zmm2,zmm0,6
00007FF62EF02775  vmovapd   zmm0{k1},zmm2
00007FF62EF0277B  vmovupd   zmm2,zmm0
00007FF62EF02781  vmovupd   zmm0,zmmword ptr [r10-100h]
00007FF62EF02788  vcmppd    k1,zmm4,zmm0,6
00007FF62EF0278F  vmovapd   zmm0{k1},zmm4
00007FF62EF02795  vmovupd   zmm4,zmm0
00007FF62EF0279B  vmovupd   zmm0,zmmword ptr [r10-0C0h]
00007FF62EF027A2  vcmppd    k1,zmm3,zmm0,6
00007FF62EF027A9  vmovapd   zmm0{k1},zmm3
00007FF62EF027AF  vmovupd   zmm3,zmm0
00007FF62EF027B5  sub       rcx,1
00007FF62EF027B9  jne       ApplyAccumulate2UR_X<Vec8d,
```

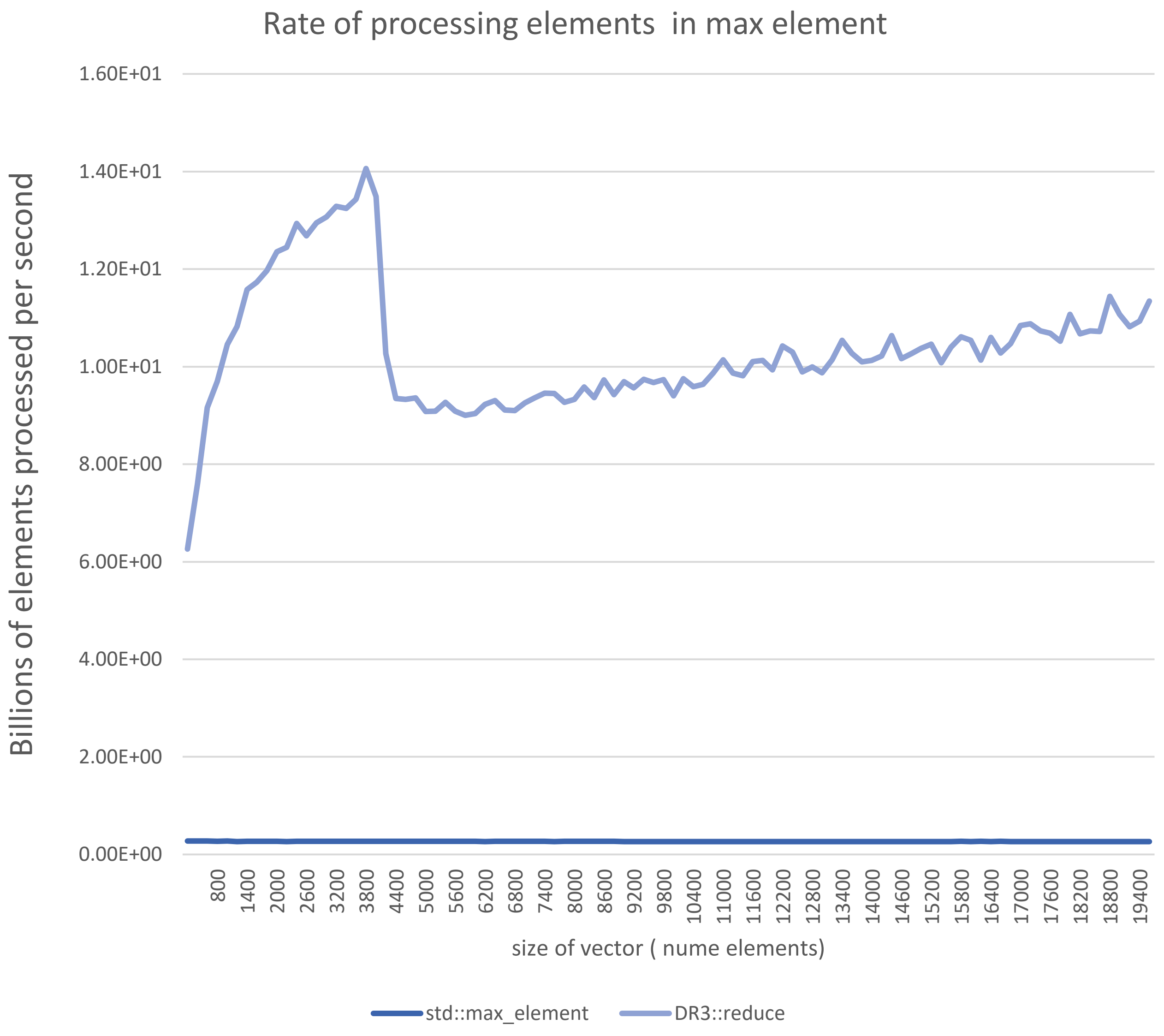
```
vmovsd      xmm0,qword ptr [rcx]     ≤ 1ms elapsed
vcomisd     xmm0,mmword ptr [rax]
cmova       rax,rcx
add         rcx,8
cmp         rcx,rbx
jne         doMax+160h (07FF62EF01980h)
```

```
for (long l = 0; l < TEST_LOOP_SZ; l++)
{
    res = *std::max_element(cbegin(v), cend(v));
}
```

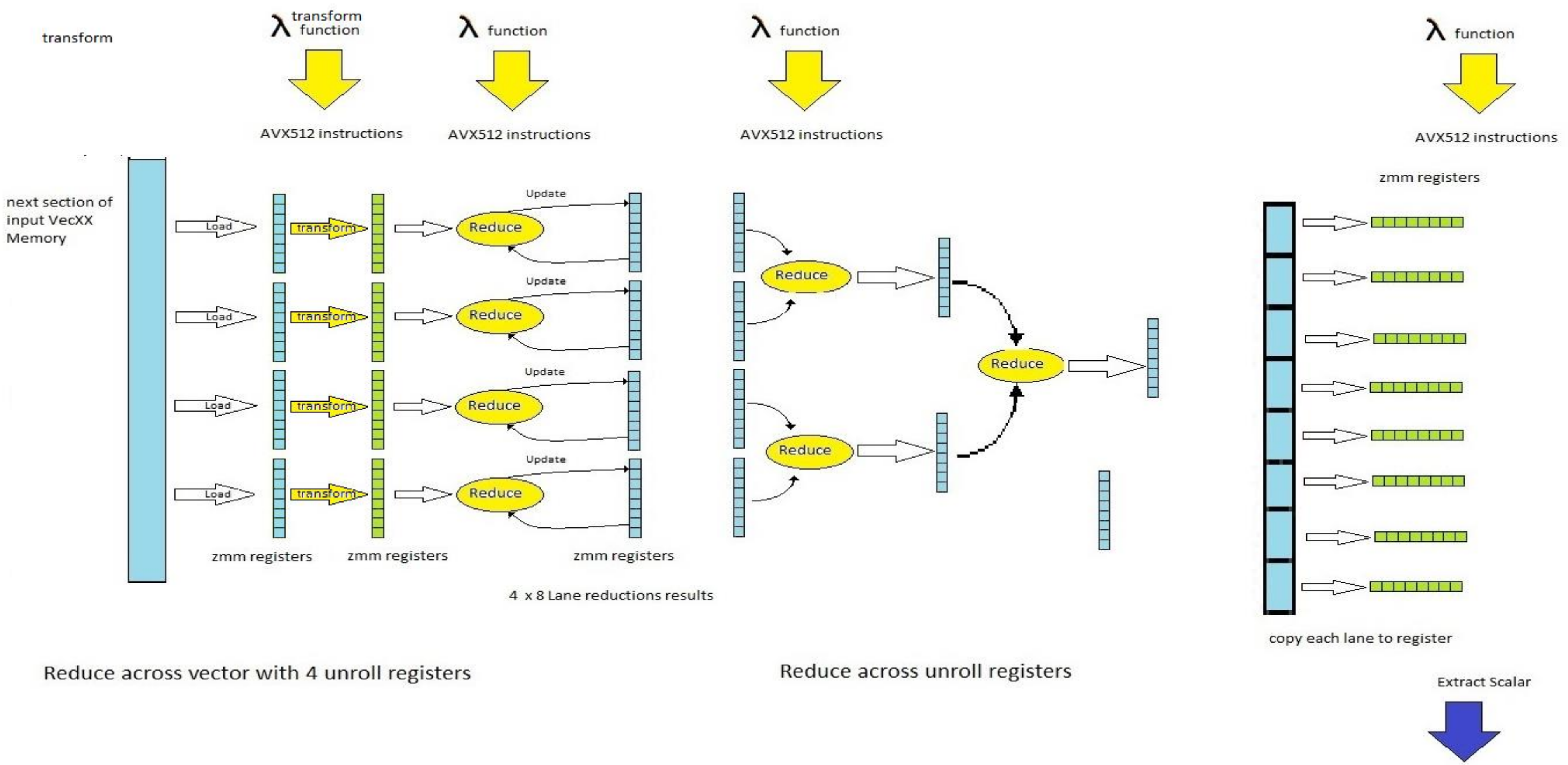
std::max_element   VC2019

using ***sd (scalar double) instructions
so not vectorised

Performance max element



transformReduce



Sum of squares using inner_product

```cpp
for (long l = 0; l < TEST_LOOP_SZ; l++)
{
    res = std::inner_product(v1.cbegin(), v1.cend(), v1.cbegin(), zero);
}
```

```
00007FF7C2321980  vmovsd      xmm0,qword ptr [rax]
00007FF7C2321984  vmulsd      xmm1,xmm0,xmm0
00007FF7C2321988  vaddsd      xmm2,xmm2,xmm1
00007FF7C232198C  add         rax,8
00007FF7C2321990  cmp         rax,rbx
00007FF7C2321993  jne         doSumSqrs+160h (07FF7C2321980h)
```

```
vmovupd      zmm1,zmmword ptr [r8]
vfmadd231pd  zmm2,zmm1,zmmword ptr [rcx]
vmovupd      zmm1,zmmword ptr [r8+40h]
vfmadd231pd  zmm4,zmm1,zmmword ptr [rcx+40h]
sub          rcx,0FFFFFFFFFFFFFF80h
sub          r8,0FFFFFFFFFFFFFF80h
add          r9,10h
cmp          r9,rax
jne          std::inner_product<std::_Vector_const_iter
```

transformReduce sum squares

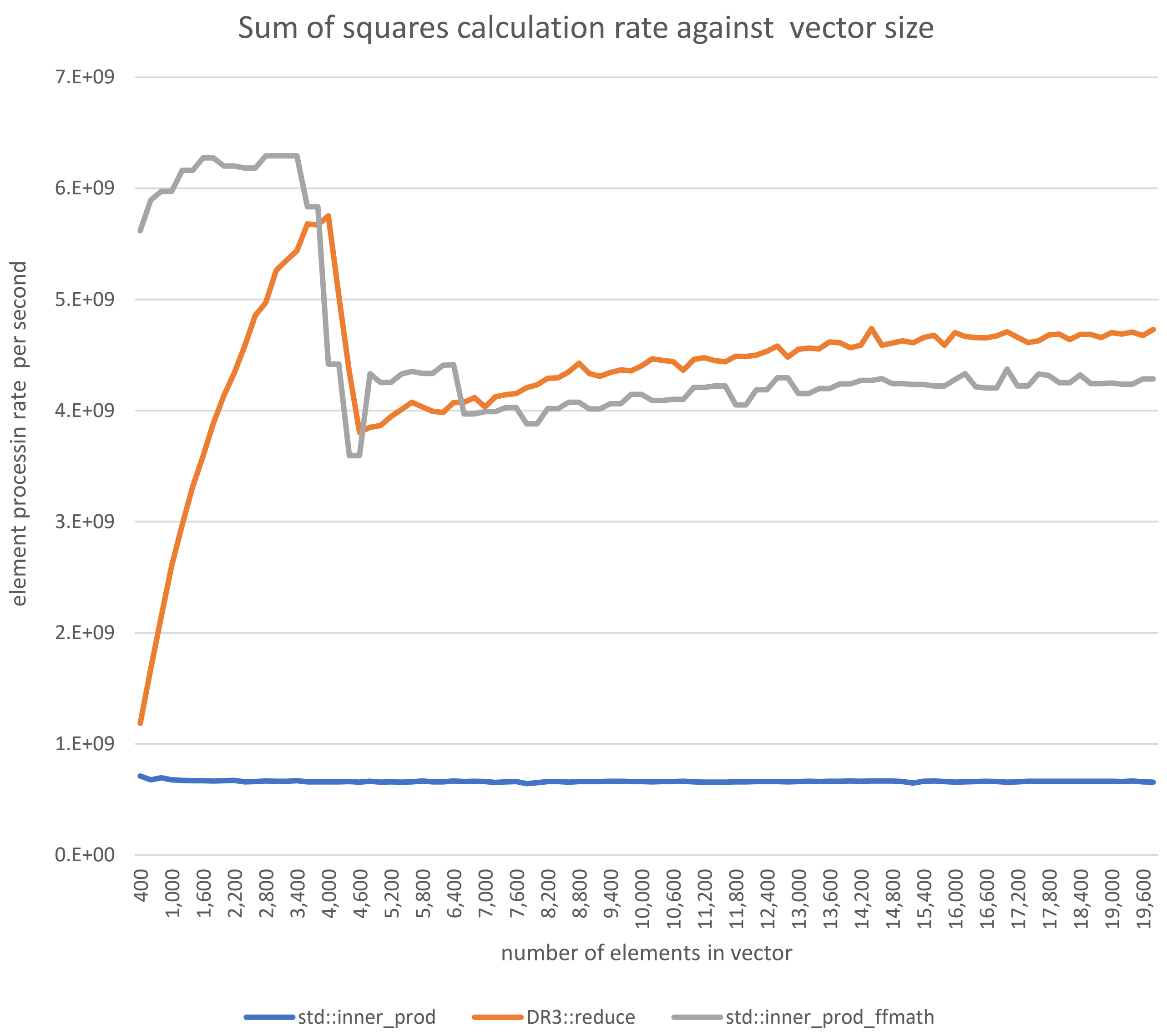
Both lambdas fused into a single vectorised fuse multiply add instruction vfmadd231pd !

```
auto Sum = [](auto lhs, auto rhs) { return lhs + rhs; };
auto SQR = [](auto X) { return X * X; };

for (long l = 0; l < TEST_LOOP_SZ; l++)
{
    res = transformReduce(t1, SQR, Sum);
}
```

```
vmovupd      zmm0,zmmword ptr [rdx]
vfmadd231pd  zmm3,zmm0,zmm0
vmovupd      zmm0,zmmword ptr [rcx-40h]
vfmadd231pd  zmm4,zmm0,zmm0
vmovupd      zmm0,zmmword ptr [rcx]
vfmadd231pd  zmm5,zmm0,zmm0
vmovupd      zmm0,zmmword ptr [rcx+40h]
vfmadd231pd  zmm2,zmm0,zmm0
lea          rdx,[rdx+100h]
lea          rcx,[rcx+100h]
sub          r10,1
jne          ApplyTransformAccumulate2UR_X<Vec8d,<lambda
```

Performance sum squares



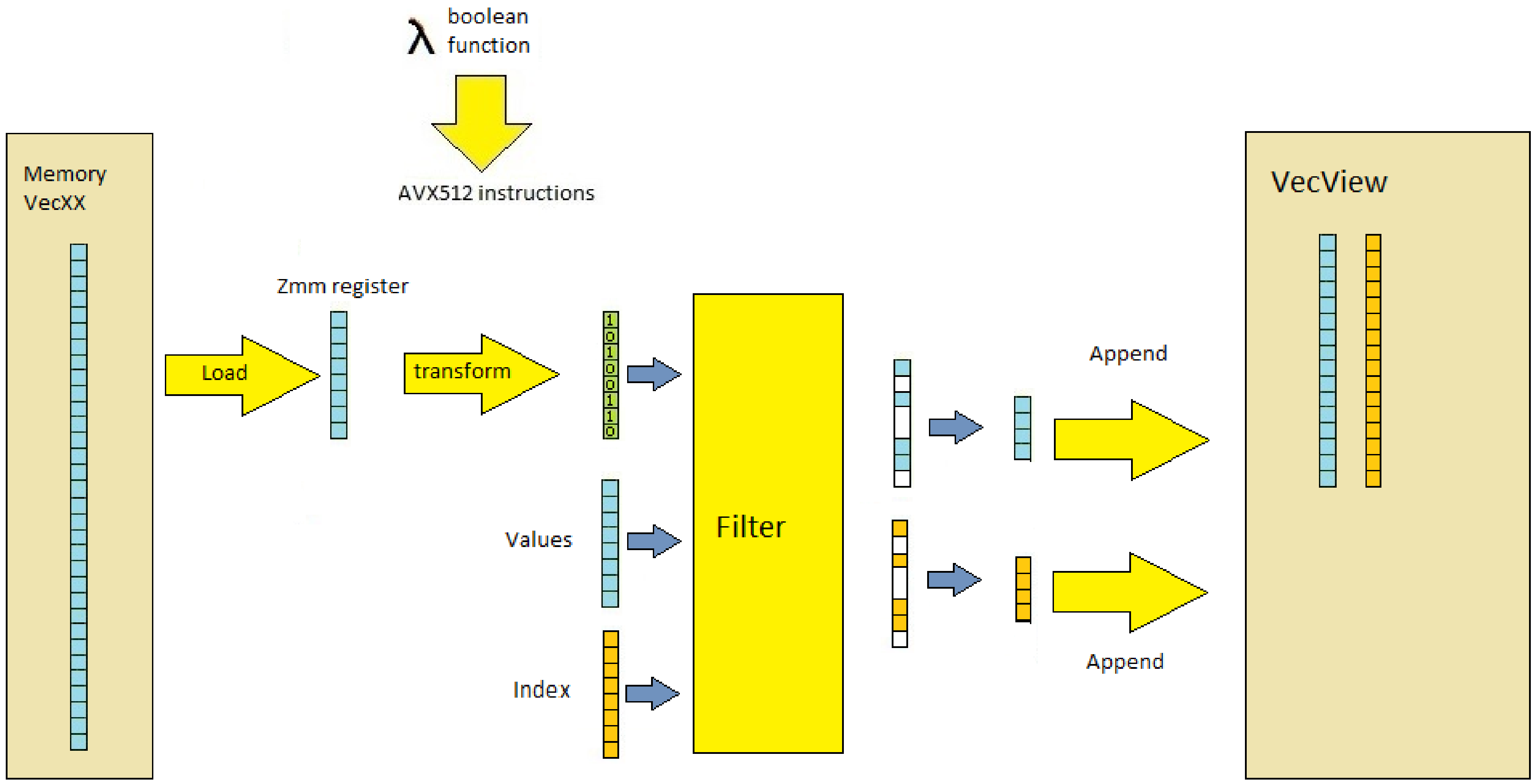
Composability

Joining lambdas together expression templates.

Filters and Views

- A filter is a boolean lambda function, returns true if an element should be copied to a view.
- View is contiguous and appropriately aligned, so elements can be further transformed and filtered
- View elements also have an index to the position in the original filtered source vector
- Views can be filtered (but retain index to their source vector)
- Views can be transformed by lambda’s

Filter to View



DR^3 Utilities

Views

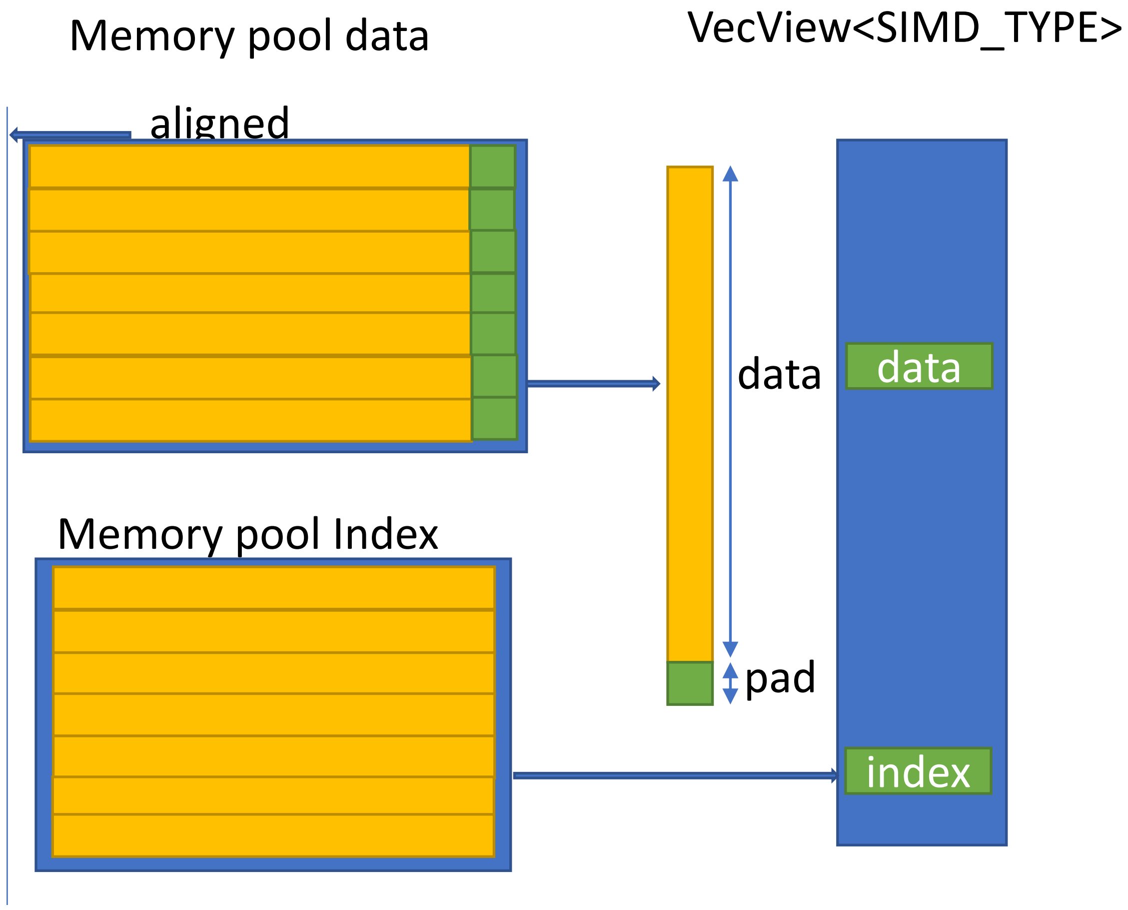
Contiguous,

Aligned,

Indexed,

By value

VecView



Operators and Functions

filter ,transform , write back

auto vecView_A = filter(vec_A,isEvenLambda)

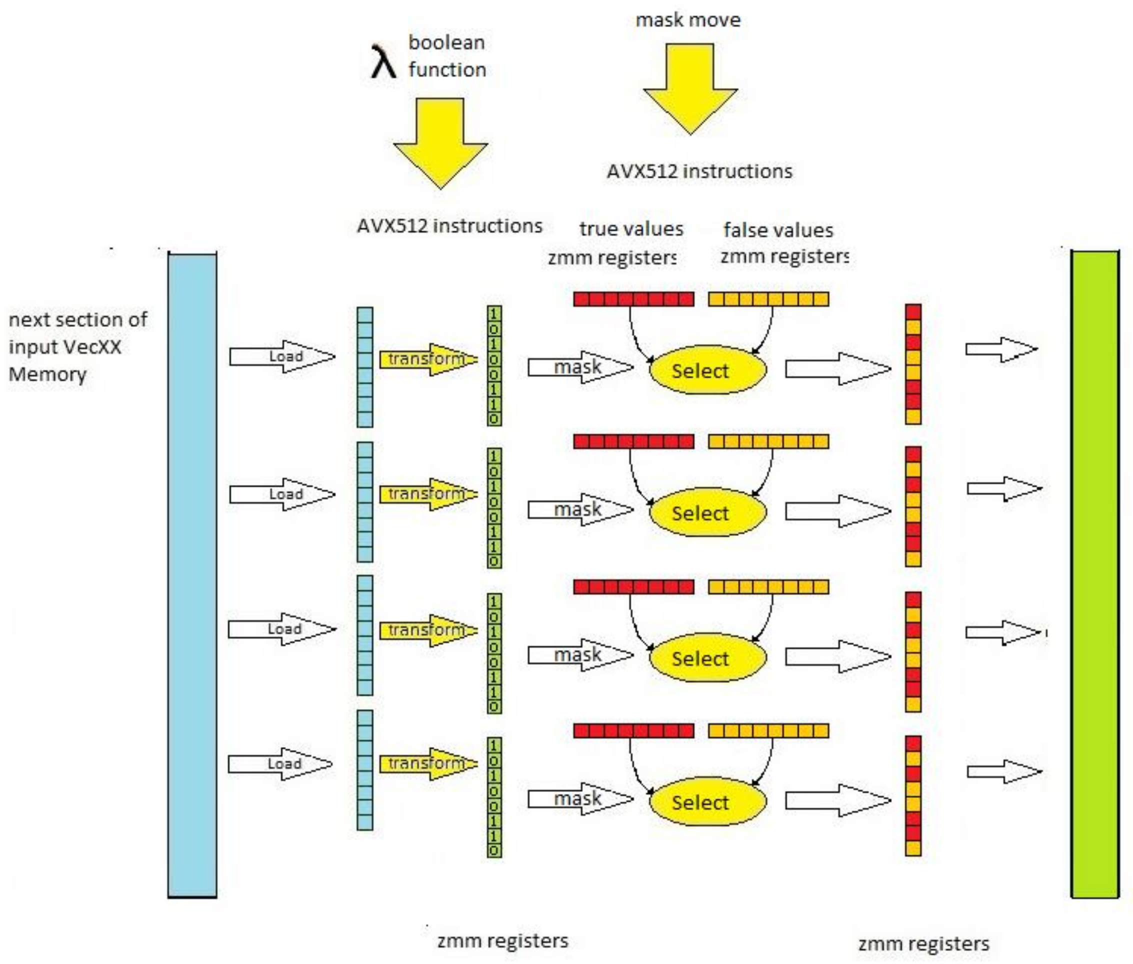
vecView_B =transform(vecView_A, square)

vecView_B.write(vec_A)

Branching

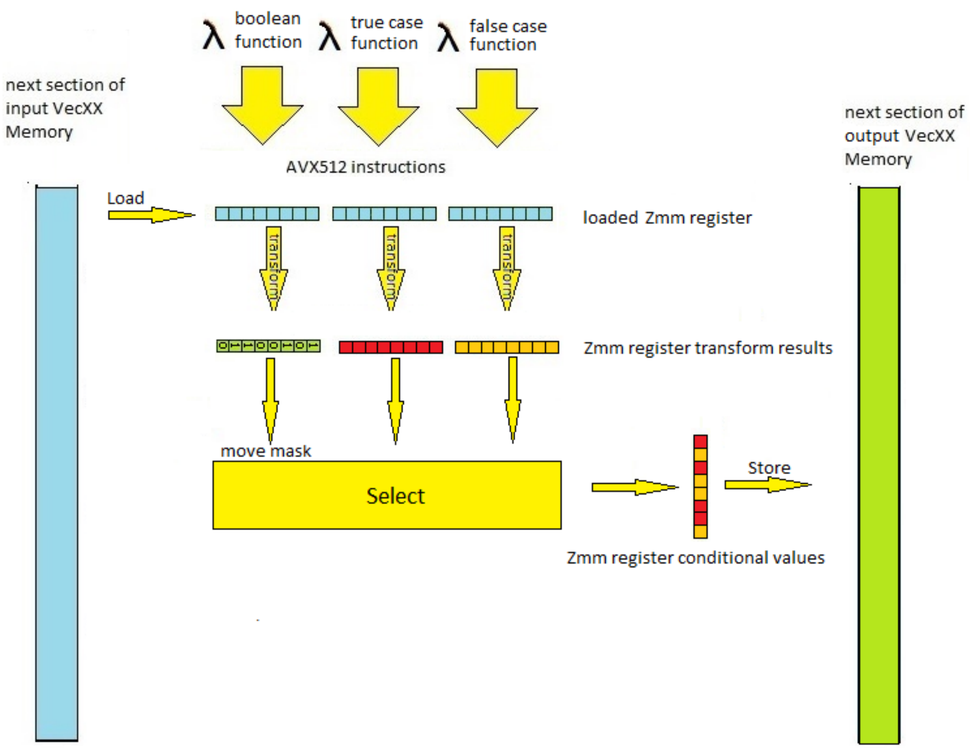
- select
- transformSelect
- filterTransform

select

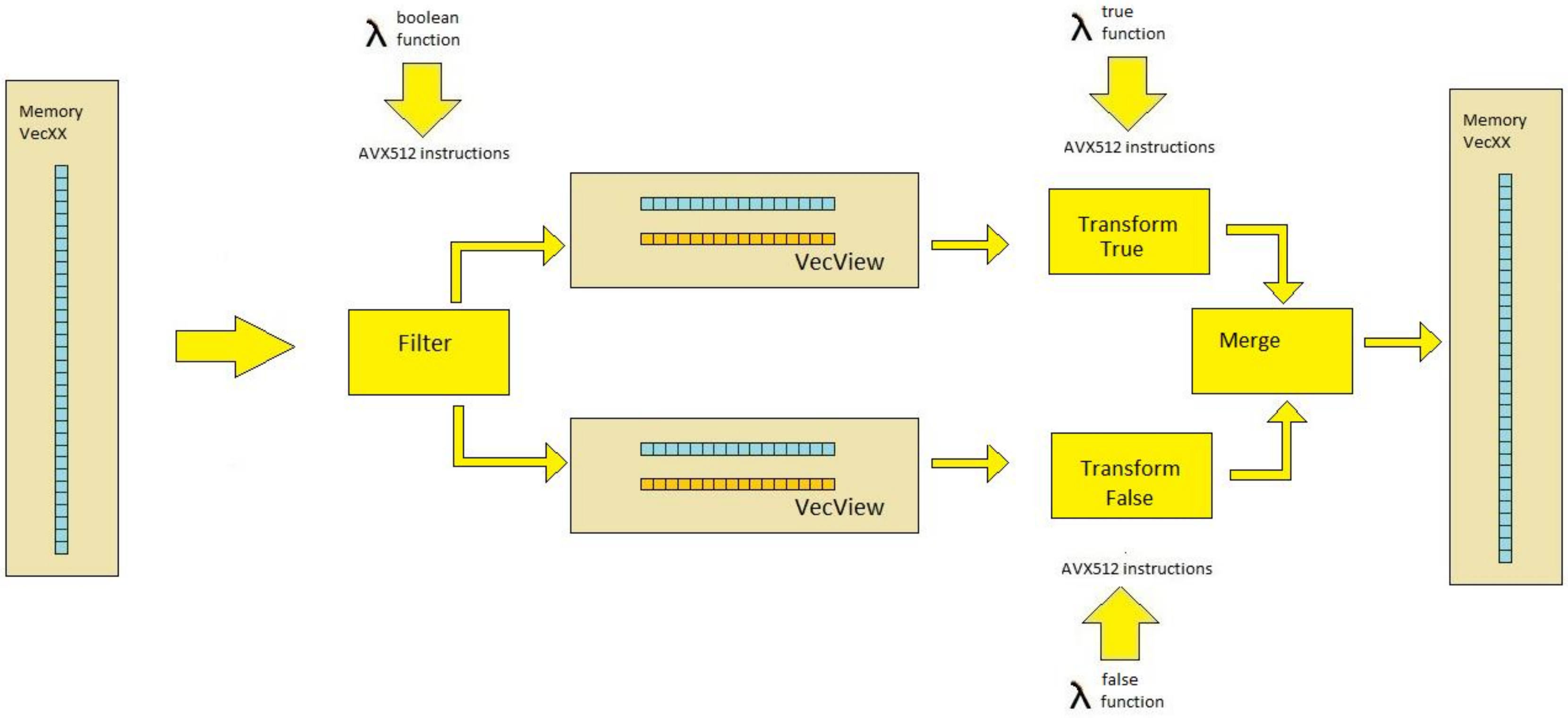


boolean function
mask move
AVX512 instructions
AVX512 instructions
true values zmm registers
false values zmm registers
next section of input VecXX Memory
Load
transform
mask
Select
zmm registers
zmm registers

transformSelect



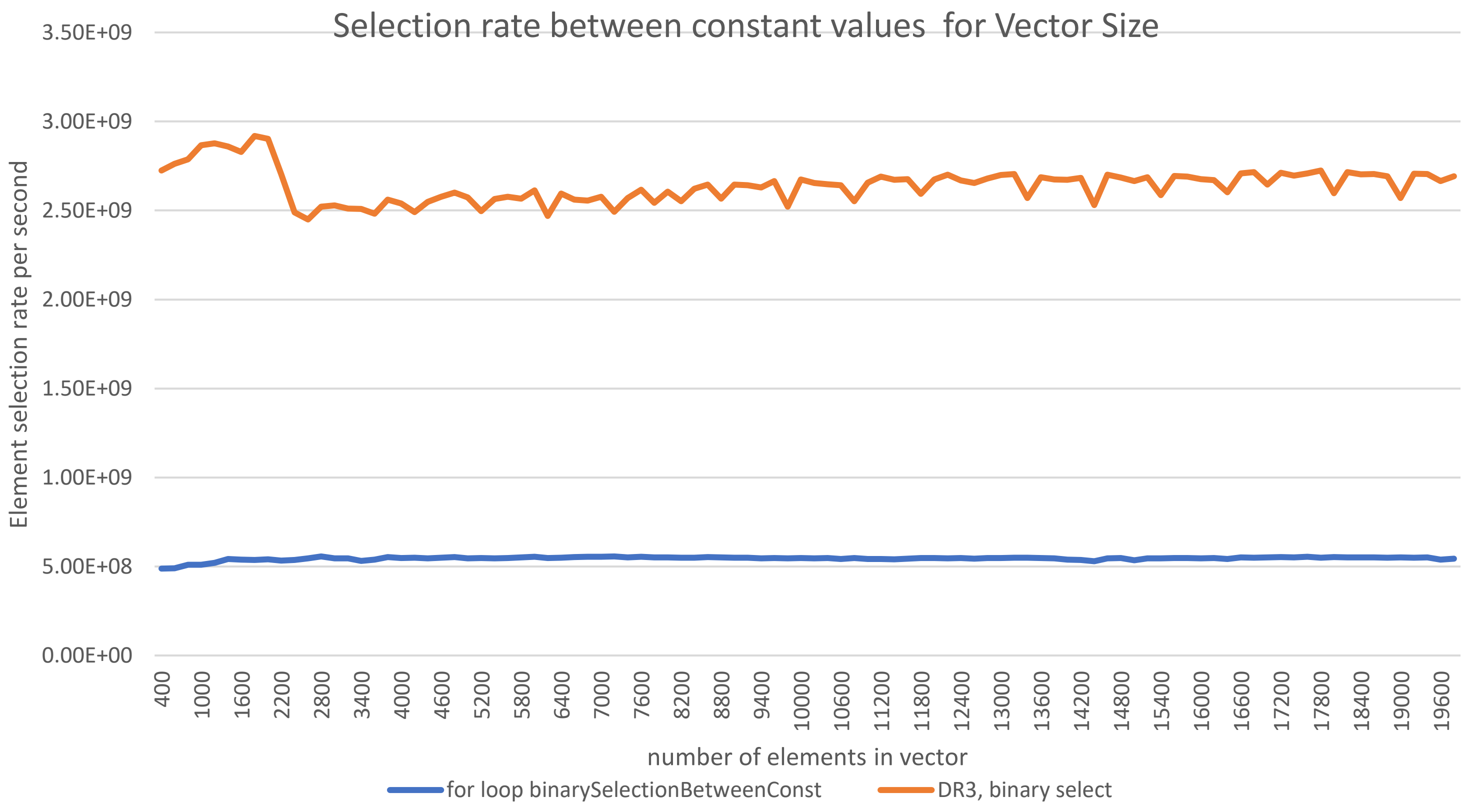
filterTransform



Select & transformSelect

more complex test condition

odd or even



transformSelect

- Deep inner loop using vectorised AVX512, including FMA

```cpp
//use auto on scalars so we can switch between float and double instruction sets too
auto one = VecXX::scalar(1.0);
auto two = VecXX::scalar(2.0);
auto half = VecXX::scalar(0.5);

auto MyOddLmbda = [&](auto x) { return  (x - two * floor(x * half)) >= one;  };
auto truVal = one;
auto falseVal = two;

for (long l = 0; l < TEST_LOOP_SZ; l++)
{
    auto res = select(MyOddLmbda, testVec, truVal, falseVal);
}
```

```
00007FF63AFB29C7  vbroadcastsd zmm4,mmword ptr [rax]
        RES = select(cond(RHS), TRU, FLS);
00007FF63AFB29CD  mov          rax,qword ptr [rdi+8]
00007FF63AFB29D1  mov          rcx,qword ptr [rdi]
        RES1 = select(cond(RHS1), TRU, FLS);
00007FF63AFB29D4  vmulpd       zmm1,zmm5,qword bcst [rax]
00007FF63AFB29DA  vrndscalepd  zmm3,zmm1,1
00007FF63AFB29E1  vbroadcastsd zmm2,mmword ptr [rcx]
00007FF63AFB29E7  vfnmadd231pd zmm5,zmm2,zmm3
00007FF63AFB29ED  vcmppd       k1,zmm5,zmm4,5
00007FF63AFB29F4  vmovupd      zmm0,zmm17
00007FF63AFB29FA  vmovapd      zmm0{k1},zmm16
        RES1.store_a(pRet + i + width);
00007FF63AFB2A00  vmovupd      zmmword ptr [r9],zmm0
```

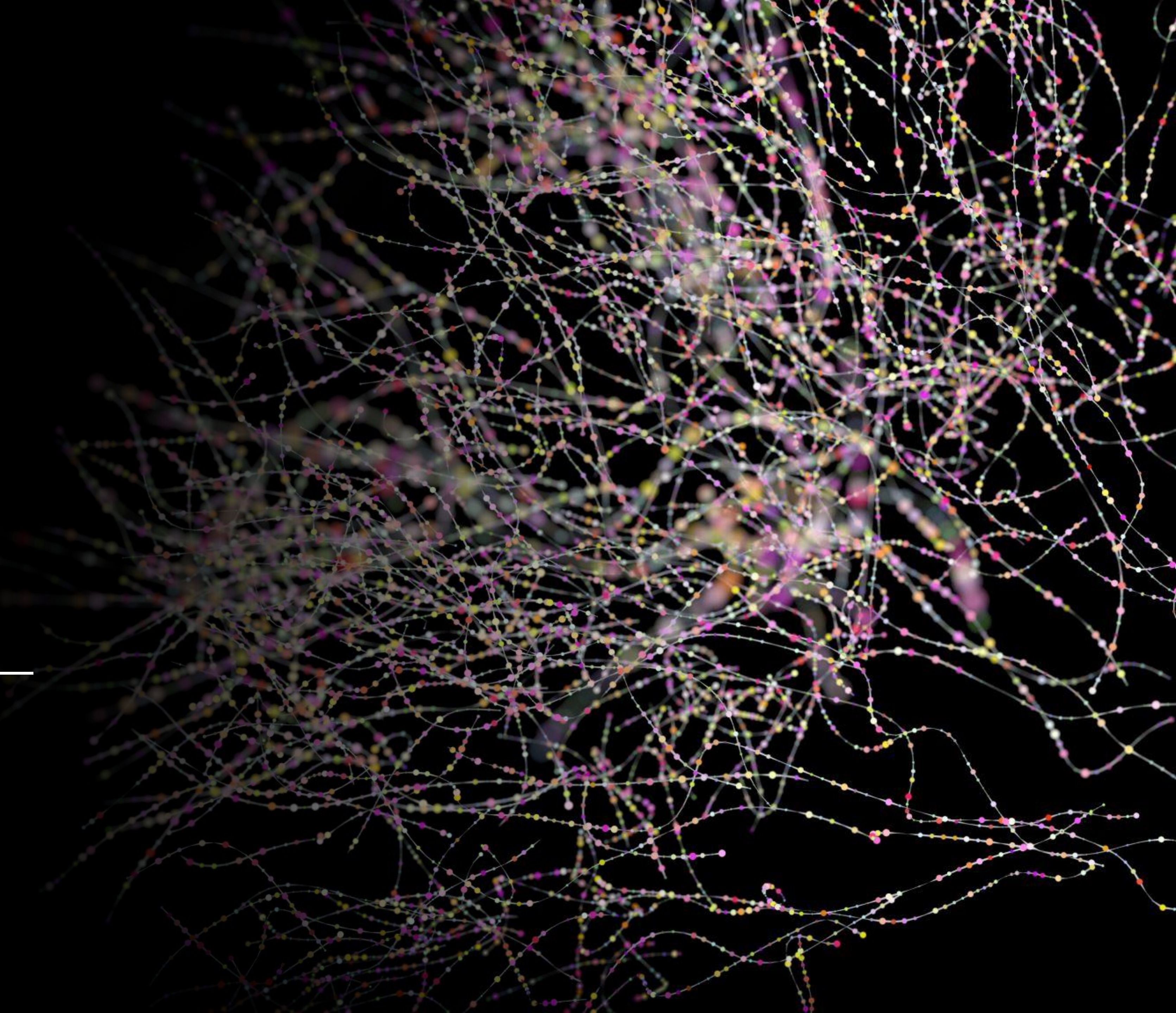
Standard for loop

- Using scalar double instructions ending in sd

```
for (int k = 0; k < SZ; k++)
{
    auto x = v1[k];
    if ((x - two * floor(x * half)) >= one)
    {
        C[k] = one;
    }
    else
    {
        C[k] = two;
    }
}
```

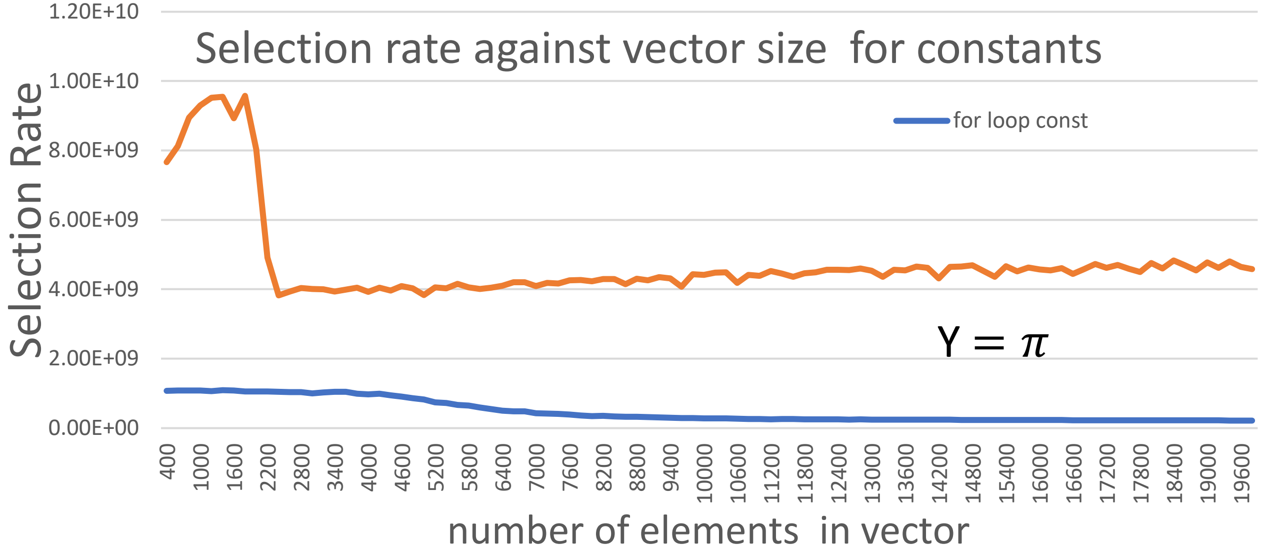
```
00007FF63AFB1A30  vmovsd      xmm2,qword ptr [rdi+rax]
00007FF63AFB1A35  vmulsd      xmm0,xmm2,mmword ptr [half]
00007FF63AFB1A3B  vxorpd      xmm1,xmm1,xmm1
00007FF63AFB1A3F  vroundsd    xmm1,xmm1,xmm0,1
00007FF63AFB1A45  vmovsd      xmm4,qword ptr [two]
00007FF63AFB1A4B  vmulsd      xmm0,xmm1,xmm4
00007FF63AFB1A4F  vsubsd      xmm2,xmm2,xmm0
00007FF63AFB1A53  vmovsd      xmm3,qword ptr [one]
00007FF63AFB1A59  vcmplesd    xmm0,xmm3,xmm2
00007FF63AFB1A5E  vblendvpd   xmm0,xmm4,xmm3,xmm0
00007FF63AFB1A64  vmovsd      qword ptr [rax],xmm0
00007FF63AFB1A68  inc         ecx
00007FF63AFB1A6A  lea         rax,[rax+8]
00007FF63AFB1A6E  mov         edx,dword ptr [rbp+0E0h]
00007FF63AFB1A74  cmp         ecx,edx
00007FF63AFB1A76  jl          binarySelectionBetweenConst+210h (07FF63AFB1A30h)
```

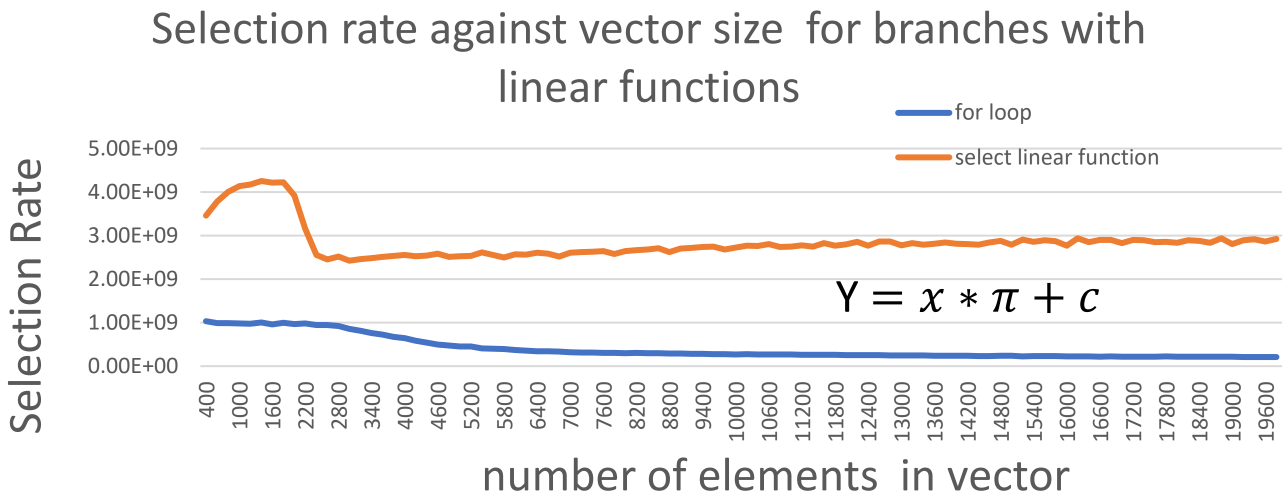
Branching examples



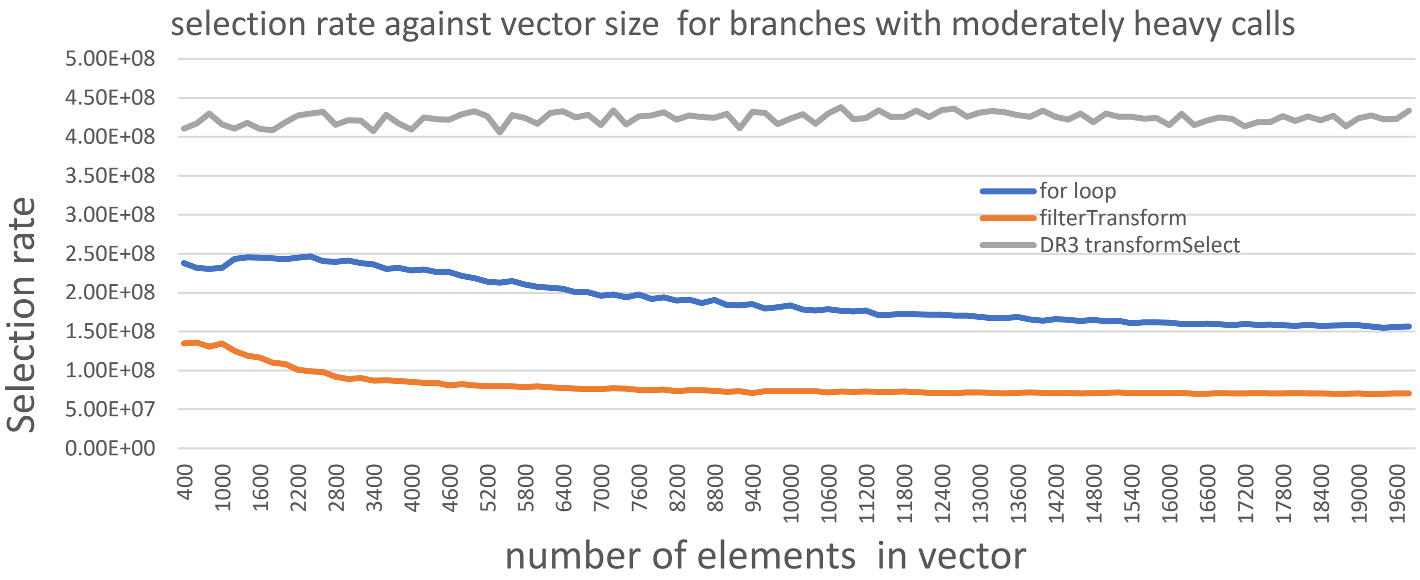
Branching with greater than as the conditional

Branching Rates with simple condition oper >

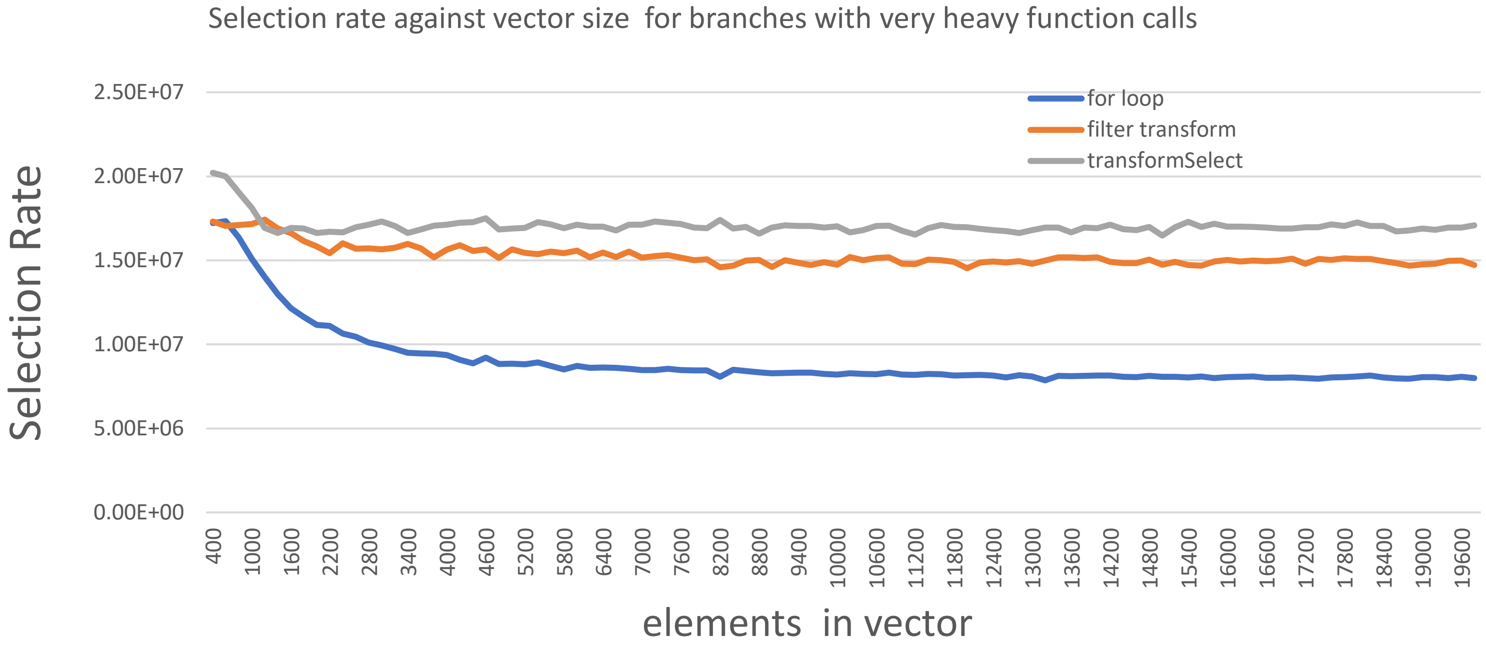




$$y=\frac{((((x+a)x+b)x+c)x+d)x+e}{((((x+a)x+b)x+c)x+d)x+e}$$



$$y=\exp\left(\frac{\sin x}{20}\right)+\frac{1}{1+\exp(\,x\sin(3.1x)+\frac{12.2}{x})}$$



Equi-probable branching

- Branchless, evaluate lambda for both sides of the branch. Blend results together masked move   results conditionally.

- Simple case, constant , linear , light weight polynomials.

- How expensive does a branch need to be, before we filter and evaluate separately?

$$y = \exp\left(\frac{\sin x}{20}\right) + \frac{1}{1+\exp(\,x\sin(3.1x)+\frac{12.2}{x})}$$

Objective

- Try out different branching strategies on a real problem.

Example Φ^{-1}

Inverse Cumulative Normal Distribution Function

Why Φ^{-1}

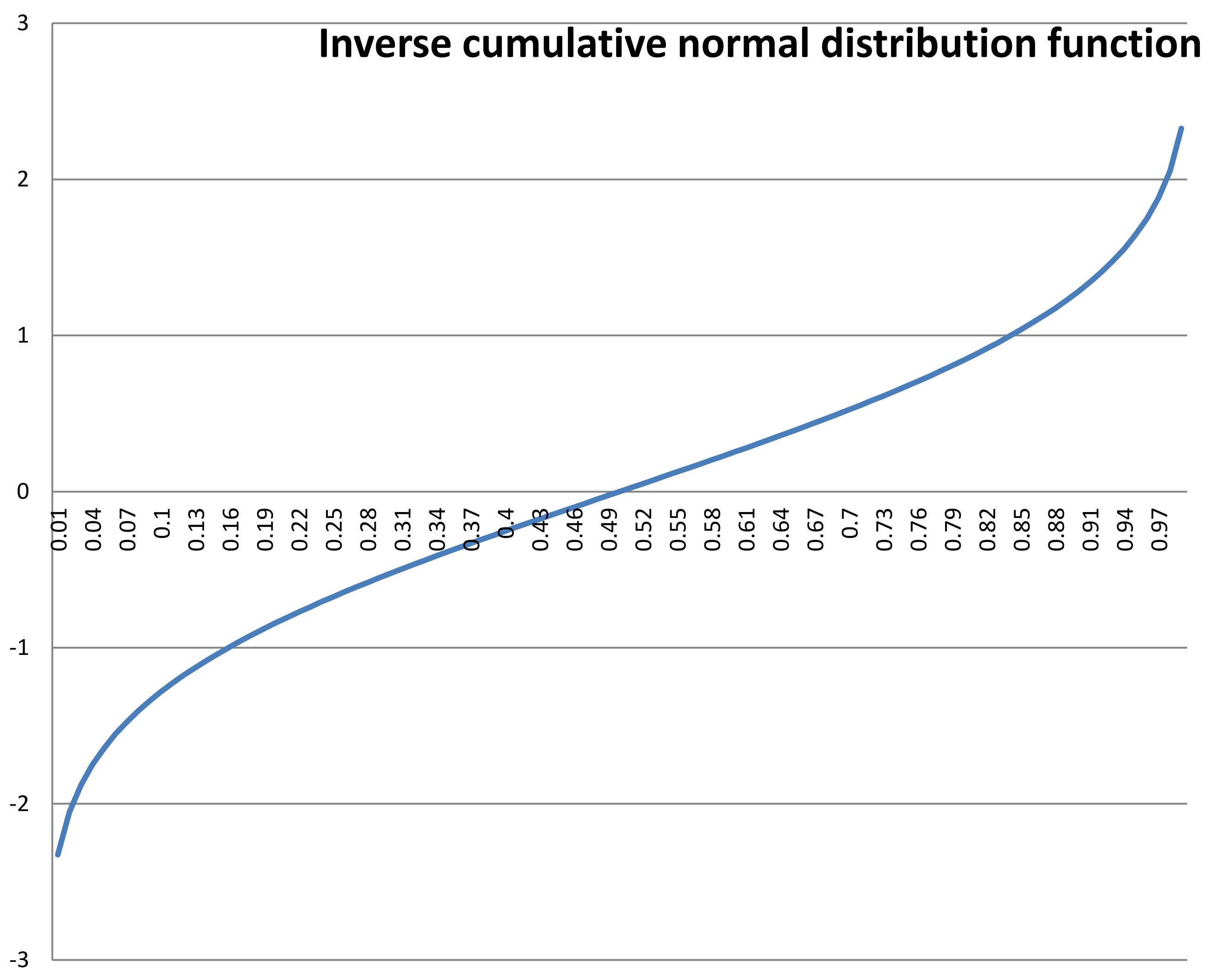
- Important for simulation of normally distributed random numbers

- Generate a set of uniformly distributed random number in the range 0.0->1.0. Transform them to normally distributed number using Φ^{-1}.

- Classic approach is to divide a function into different approximation regions and branch to the function which maps best for the input value. There may be mapping before and after the branching.

- It is a useful example for branching and perhaps writing custom special functions.

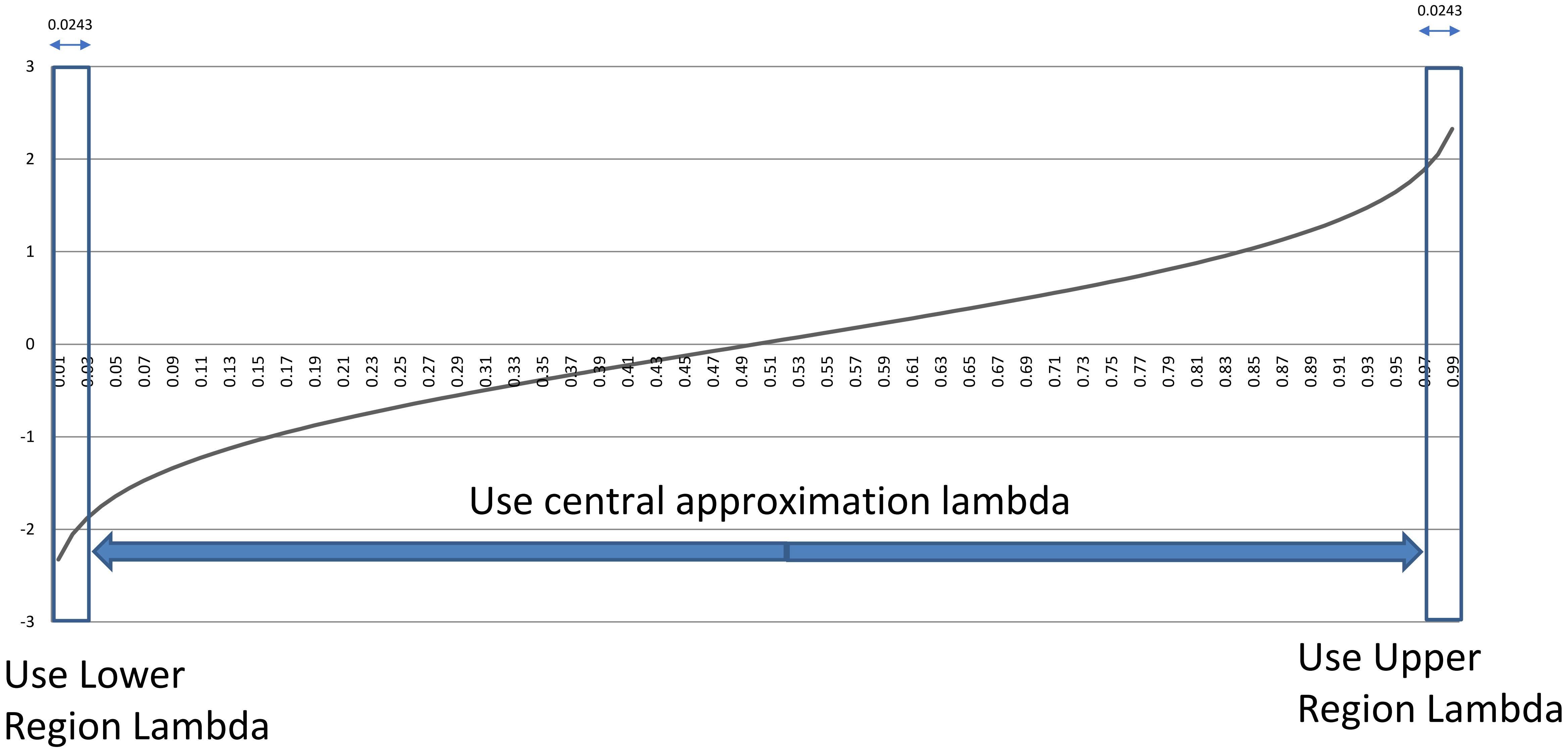
The function Φ^{-1}



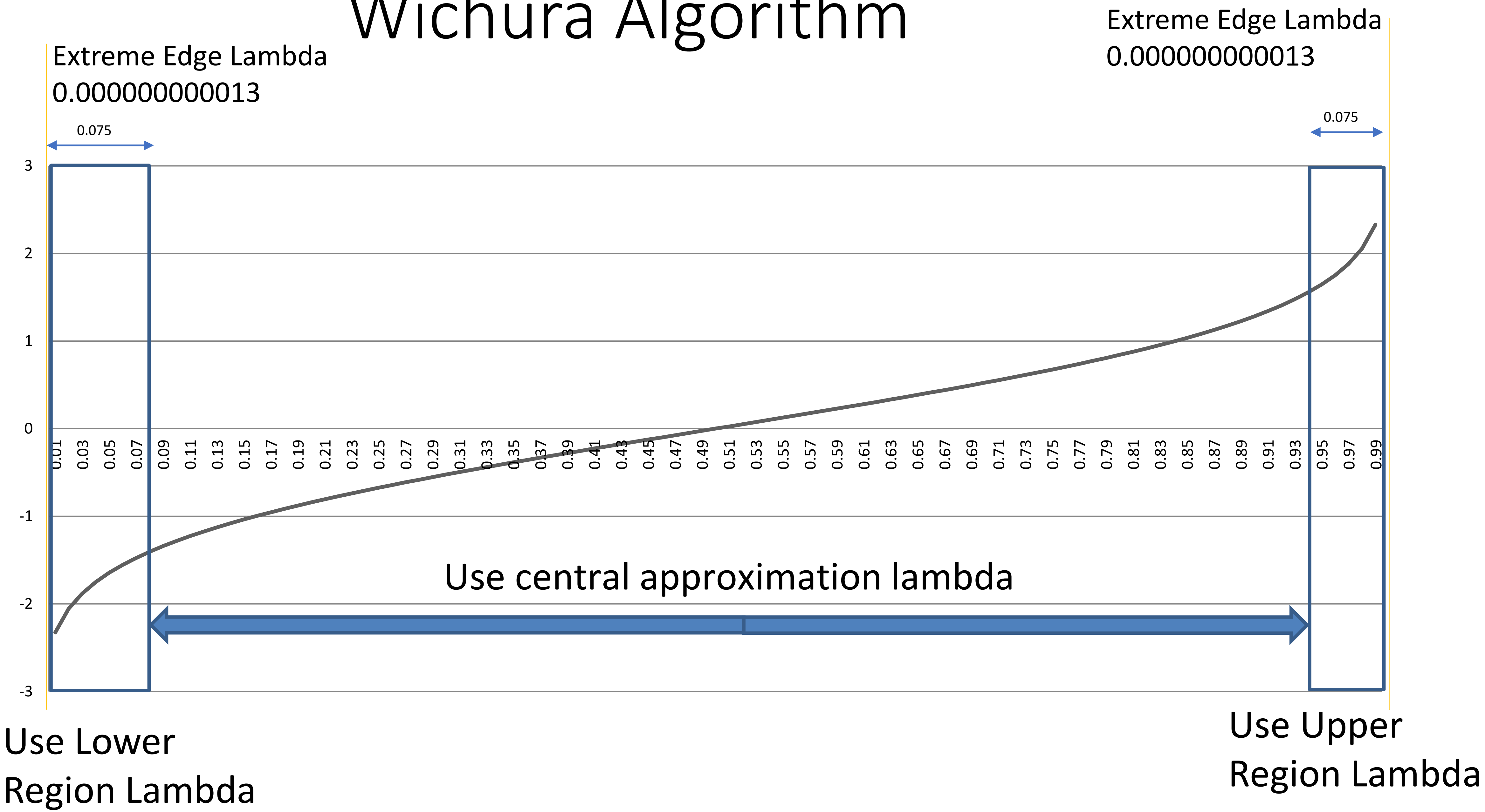
Approach

- Consider two approximation schemes
- Acklam - central region + upper and lower region 6^{th} order rational polynomial 10 digits
- Wichura241 - central region two upper and two lower regions. 8^{th} order rational polynomial – 16 digits

Acklam's Algorithm



Wichura Algorithm

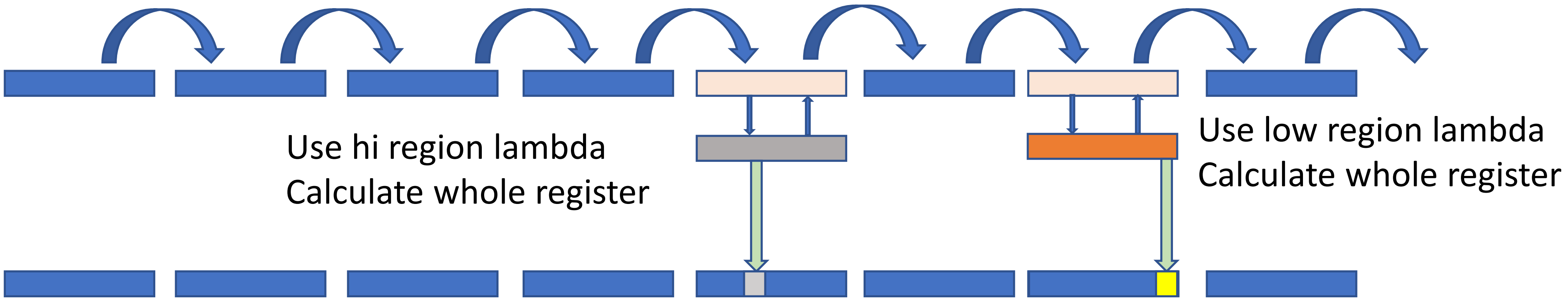


Implementation Approaches

- Complex Single Pass
- Sparse Update
- Filter to Views
- Transform-Filter , Transform-write

Complex Single Pass

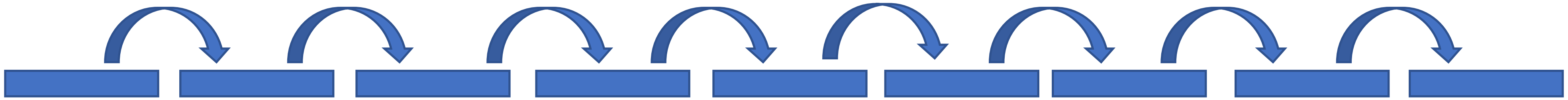
Calculate central approximation value for registers
If not all values calculated with hi/low region lambda



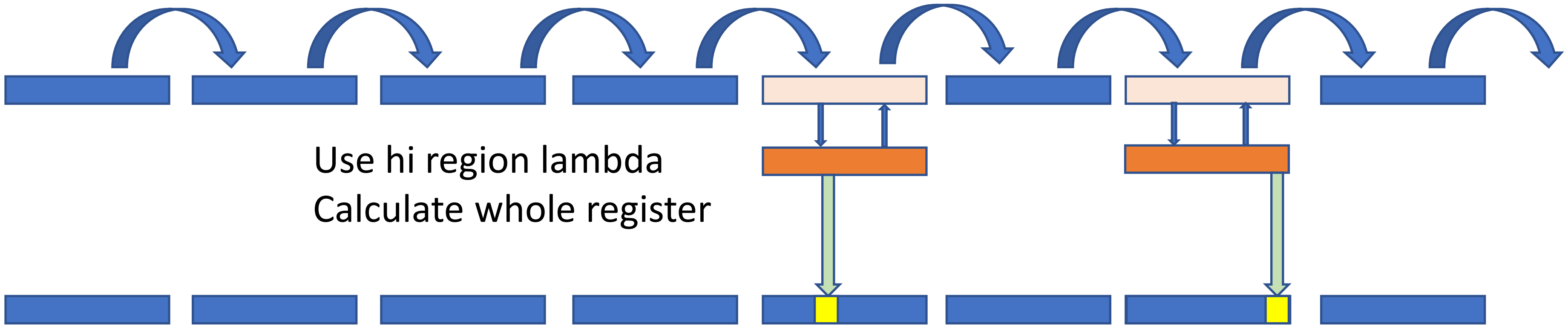
Blend outer region results in to appropriate target register lanes

Sparse Update

1) Apply central region lambda



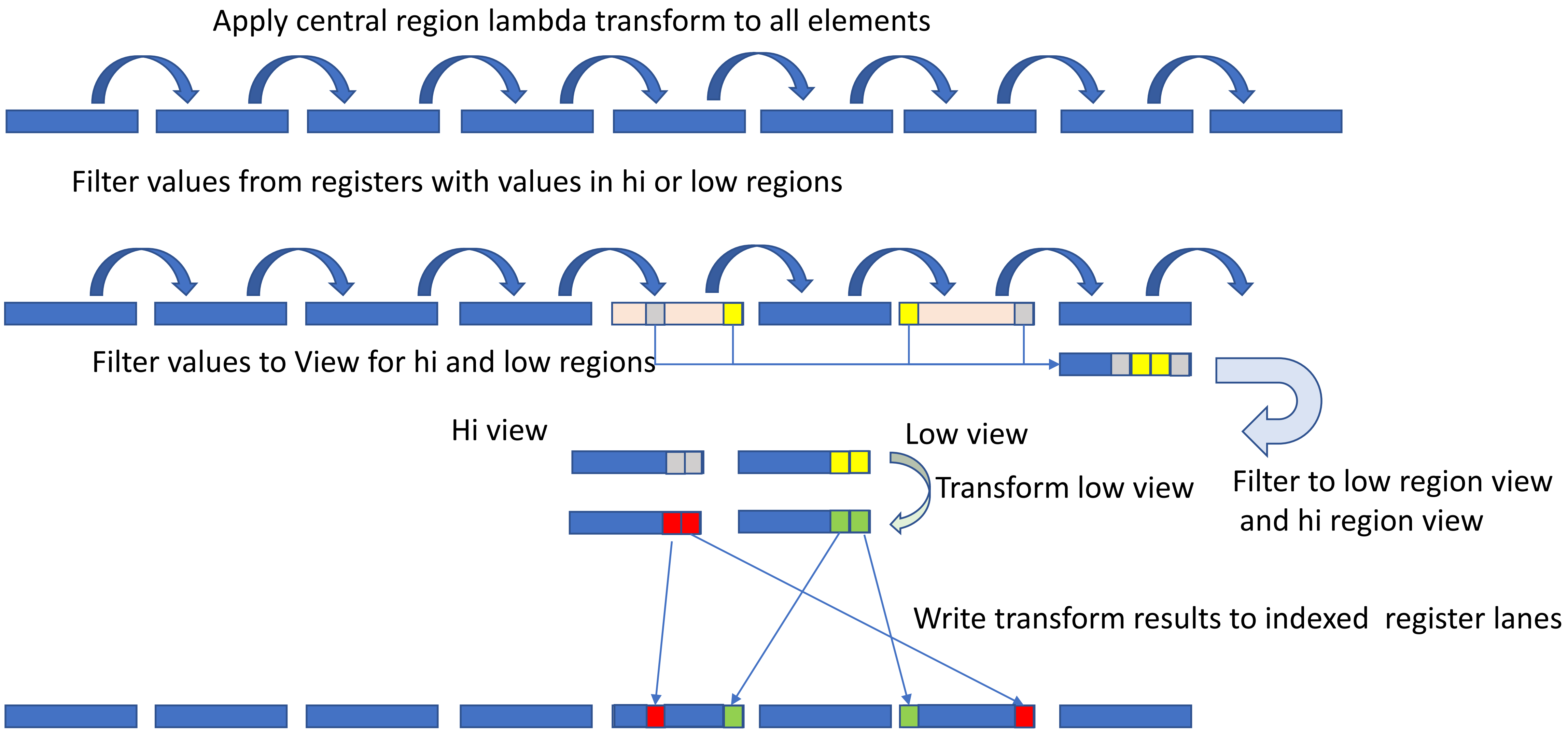
2) Scan for registers with values that should be calculated with region lambda hi



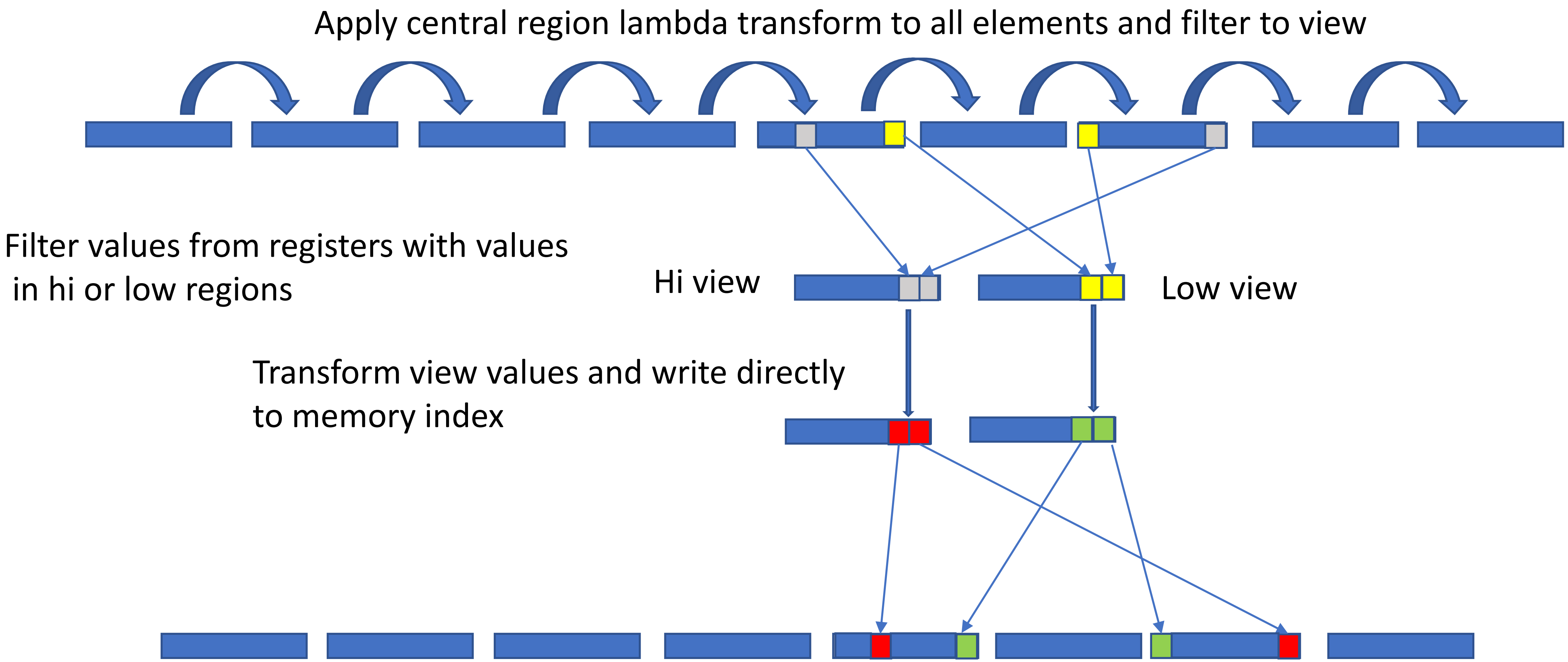
Blend outer region results in to appropriate target register lanes

3) Scan for registers with values that should be calculates with low region lambda and update as in previous stage

Filter to Views



Transform-Filter and Transform-Write



Coding Implementation Approach

```cpp
VecXX calcCDFNormWichuraViewsAndFMA2splits(const VecXX& inputVecX)
{
    //Extrema boundary constants
    constexpr auto ExtrmMin =0.00000000001388;// exp(-25.0);
    constexpr auto ExtrmMax = 1. - ExtrmMin;

    // region test lambdas
    auto isExtremeLambda = [&](auto p){ ... }

    auto isOuterRangelambda = [&](auto p){ ... }

    //region evaluation lambdas
    auto centralRegionLambda = [](auto p){ ... }

    auto outerRegionlambda = [](auto p){ ... }


    auto extremaRegionLambda = []( auto p){ ... }


    // Apply central region lambda to all elements and filter outer range and extrema range elements to views
    // return all inside a tupple.
    auto tple = ApplyOperationAndFilter(centralRegionLambda, isOuterRangelambda, isExtremeLambda, inputVecX);

    auto& res = std::get<0>(tple);  // main result initially filed with values from applying central Region Lambda
    auto& outside = std::get<1>(tple); // view containing outer range elements
    auto& extreme = std::get<2>(tple); // view containing extreme range elements ( usually empty)

    // use outerRegionLambda to transform  filtered  outer range values in  the view (outside) and write the results directly to result object res
    ApplyUnitaryOperationWrite(outerRegionlambda, outside, res);
    if (extreme.size() < 1)
    {
        return res;
    }

    //  transform values filtered to the extrema view using  lambda  extremaRegionLambda  and write transformed values to res
    ApplyUnitaryOperationWrite(extremaRegionLambda, extreme, res);
    return res;
}
```

```cpp
VecXX calcCDFNormWichuraViewsAndFMA2splits(const VecXX& inputVecX)
{
    //Extrema boundary constants
    constexpr auto ExtrmMin =0.00000000001388;// exp(-25.0);
    constexpr auto ExtrmMax = 1. - ExtrmMin;

    // region test lambdas
    auto isExtremeLambda = [&](auto p)
    {
        return (p < ExtrmMin) || (p > ExtrmMax);
    };

    auto isOuterRangelambda = [&](auto p)
    {
        constexpr auto p_low = 0.5 - 0.425;
        constexpr auto p_high = 1.0 - p_low;
        return ((p_high < p) && (ExtrmMax > p)) || ((p > ExtrmMin) && (p < p_low));

    };

    //region evaluation lambdas
    auto centralRegionLambda = [](auto p)
    {
        auto q = p - 0.5;
        auto r = .180625 - q * q;
        auto denom = 1. / (mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(5226.495278852854561, r, 28729.085735721942674), r, 39307.89580009271061), r, 21213.794301586595867), r, 5
        auto num = (mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(2509.0809287301226727, r, 33430.575583588128105), r, 67265.770927008700853), r, 45921.953931549871457), r, 13731
        return denom * num;
    };

    auto outerRegionlambda = [](auto p)
    {
        //wichura polynomial coefficients
        constexpr static  double c[] = { 7.7454501427834140764e-4 , .0227238449892691845833 ,.24178072517745061177, 1.27045825245236838258 ,  3.64784832476320460504, 5.769497221460691405 5,
        constexpr static  double d[] = { 1.05075007164441684324e-9 , 5.475938084995344946e-4, .0151986665636164571966, .14810397642748007459, .68976733498510000455,  1.6763848301838038494,
        auto  r = min(p, 1 - p);
        r = sqrt(-log(r));

        auto q = p - 0.5;
        r += -1.6;
        auto denom = (decltype(p))(1.0) / mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(d[0], r, d[1]), r, d[2]), r, d[3]), r, d[4]), r, d[5]), r, d[6]), r, (decltype(p))(1.0));
        auto    val = mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(mul_add(c[0], r, c[1]), r, c[2]), r, c[3]), r, c[4]), r, c[5]), r, c[6]), r, c[7]);
        val = val * denom;
        auto valMult = iff(q < (decltype(p))(0.0), (decltype(p))(-1.0), (decltype(p))(1.0));
        val *= valMult;
        return val;
    };

```

Relative Performance Table

Processor	W2123	W2123	W2123	W2123	4114	4114	4114	4114
Instruction set	AVX512	AVX512	AVX2	AVX2	AVX512	AVX512	AVX2	AVX2
compiler	ICC	VS2019	Vs2019	ICC	ICC	VC2019	ICC	VS2019
Implementations								
10 digit Acklam implementation								
Multiple Sparse Passes with FMA	5.83E+08	1.92E+08	4.26E+08	5.45E+08	2.92E+08	9.62+07	4.31E+08	3.58+08
Complex Single Pass with FMA	6.5E+08	1.65E+08	3.21E+08	6.66E+08	3.46E+08	1.36E+08	5.68E+08	2.58+08
Filter To Views and transform	8.47E+08	6.56E+08	5.98E+08	5.00E+08	3.49E+08	3.3E+08	4.25E+08	2.98+08
Filter To Views and transform with FMA	9.44E+08	6.63E+08	6.85E+08	7.36E+08	4.6E+08	3.39E+08	5.99E+08	3.57+08
Transform-filter +Transform-Write with FMA	9.7E+08	6.57E+08	6.84E+08	7.45E+08	4.6E+08	3.38E+08	6.15E+08	3.64+08
16 digit implementation								
Transform-filter + Transform-Write with FMA (WS241)	6E+08	2.96E+08	3.20E+08	3.79E+08	2.88E+08	2.41E+08	3.75E+08	1.65+08
Intel short vector math library	2.61E+08	4.96E+08	2.13E+08	2.14E+08	2.15E+08	3.8E+08	1.62E+08	1.66+08
transform	1.67E+08	9.84E+07	1.00E+08	1.63E+08	1.38E+08	7.86+07	1.30E+08	7.87+07

Table 1: Maximum Observed Single-threaded double precision calculation through-put for Φ-*1* per second. For different implementations, execution hardware, compiler, and intrinsic vector instruction set.

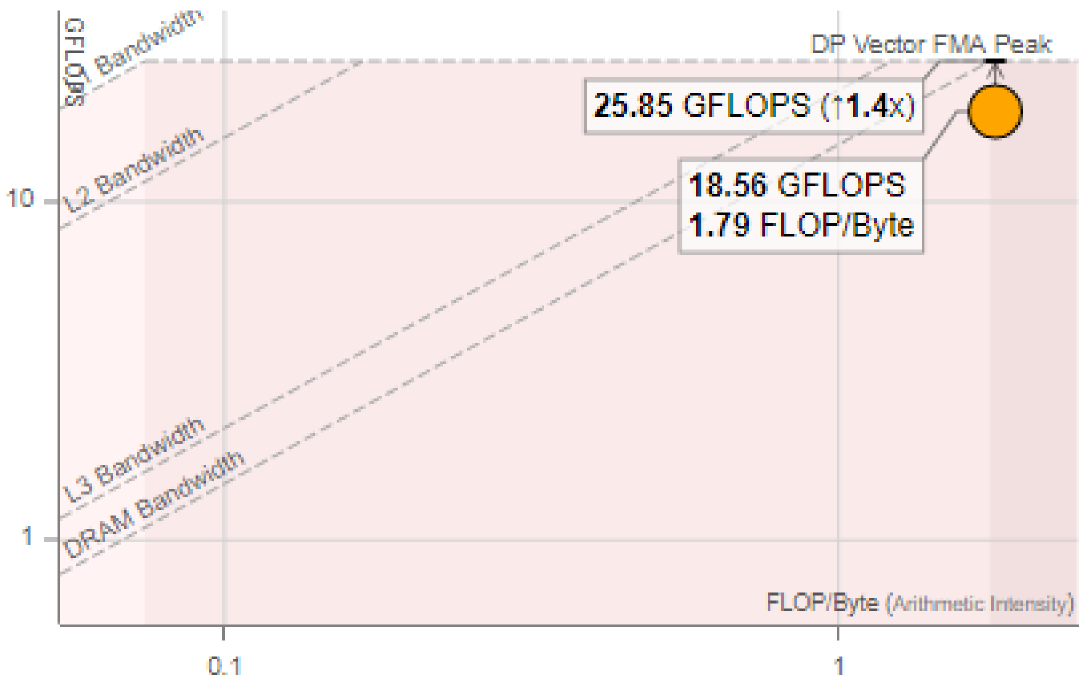
Roof Line analysis and inner loop assembly

- 2/3 of time in first stage transform whole vector with central approximation and filter to view outer region values
- 1/3 of time processing outer region values

Central Lambda WS241 AVX2

	Address	Line	Assembly	Total Time	%	Self Time	%	
	0x14000425d		**Block 2: 617520000**					
	0x14000425d	230	vmovupd ymm6, ymmword ptr [rdx+rcx*8-0x78]	0.016s		0.016s		
	0x140004263	230	vaddpd ymm7, ymm6, ymm0	0.032s		0.032s		
	0x140004267	230	vmovapd ymm8, ymm7	0.127s		0.127s		
	0x14000426b	230	vfnmadd213pd ymm8, ymm7, ymm1	0.064s		0.064s		FMA
	0x140004270	230	vmovapd ymm9, ymm2	0.016s		0.016s		
	0x140004274	230	vfmadd213pd ymm9, ymm8, ymm3	0.016s		0.016s		FMA
	0x140004279	230	vfmadd213pd ymm9, ymm8, ymm4	0.124s		0.124s		FMA
	0x14000427e	230	vfmadd213pd ymm9, ymm8, ymm5	0.079s		0.079s		FMA
	0x140004283	230	vfmadd213pd ymm9, k0, ymm8, ymm16	0.032s		0.032s		FMA
	0x140004289	230	vfmadd213pd ymm9, k0, ymm8, ymm17					FMA
	0x14000428f	230	vfmadd213pd ymm9, k0, ymm8, ymm18	0.298s		0.298s		FMA
	0x140004295	230	vfmadd213pd ymm9, k0, ymm8, ymm19	0.092s		0.092s		FMA
	0x14000429b	230	vdivpd ymm9, k0, ymm19, ymm9					Divisions
	0x1400042a1	230	vmovapd ymm10, k0, ymm20	1.364s		1.364s		
	0x1400042a7	230	vfmadd213pd ymm10, k0, ymm8, ymm21	0.016s		0.016s		FMA
	0x1400042ad	230	vfmadd213pd ymm10, k0, ymm8, ymm22	0.016s		0.016s		FMA
	0x1400042b3	230	vfmadd213pd ymm10, k0, ymm8, ymm23					FMA
	0x1400042b9	230	vfmadd213pd ymm10, k0, ymm8, ymm24	0.266s		0.266s		FMA
	0x1400042bf	230	vfmadd213pd ymm10, k0, ymm8, ymm25	0.016s		0.016s		FMA
	0x1400042c5	230	vfmadd213pd ymm10, k0, ymm8, ymm26					FMA
	0x1400042cb	230	vfmadd213pd ymm10, k0, ymm8, ymm27					FMA
	0x1400042d1	230	vmulpd ymm7, ymm10, ymm7	0.361s		0.361s		
	0x1400042d5	230	vmulpd ymm7, ymm9, ymm7	0.016s		0.016s		
	0x1400042d9	230	vmovupd ymmword ptr [r11+rcx*8], ymm7	0.404s		0.404s		
	0x1400042df	230	vcmppd k1, k0, ymm6, ymm28, 0x1	0.206s		0.206s		
	0x1400042e6	230	vcmppd k0, k1, ymm6, ymm29, 0xe					
	0x1400042ed	230	vcmppd k1, k0, ymm6, ymm30, 0x1					
	0x1400042f4	230	vcmppd k1, k1, ymm6, ymm31, 0xe					
	0x1400042fb	230	korw k0, k1, k0	0.295s		0.295s		Mask Manipulations
	0x1400042ff	230	kshiftlb k0, k0, 0x4					Mask Manipulations
	0x140004305	230	kshiftrb k0, k0, 0x4	0.016s		0.016s		Mask Manipulations
	0x14000430b	230	kortestb k0, k0	0.015s		0.015s		Mask Manipulations
	0x14000430f	230	jz 0x140004349 <Block 8>	0.186s		0.186s		

Roofline



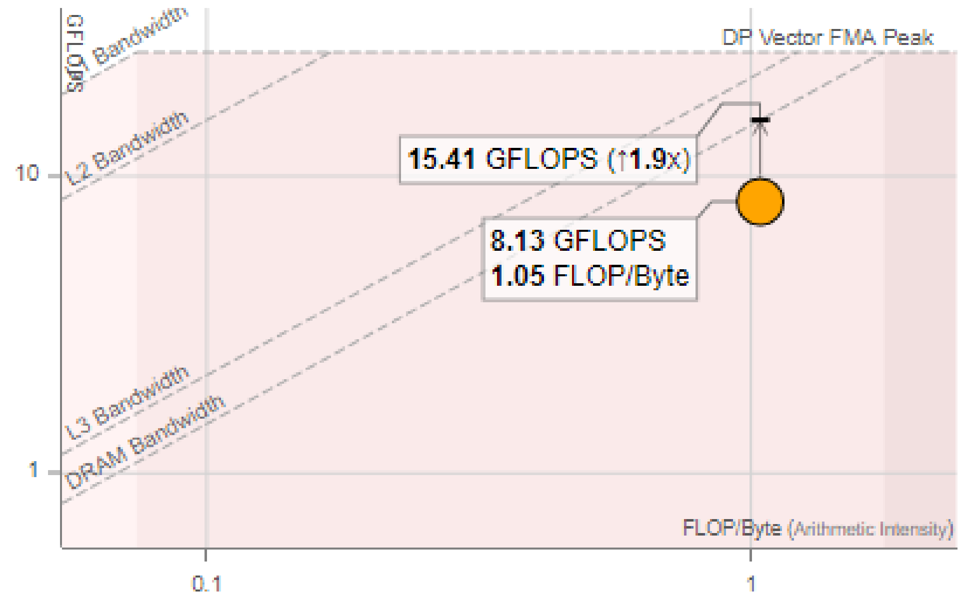
No performance recommendations are available based on current Roofline data

GFLOPS: 18.56
GINTOPS: 0.23
AVX-512 Mask Usage: 100%

Outer Region Lambda WS241 AVX2

0x140005730		**Block 1: 92020000**			
0x140005730	237	vxorpd ymm21, k0, ymm27, ymm31			
0x140005736	237	vsqrtpd ymm21, k0, ymm21			Square Roots
0x14000573c	237	vaddpd ymm21, k0, ymm21, ymm7	0.010s	0.010s	
0x140005742	237	vmovapd ymm26, k0, ymm8	0.142s	0.142s	
0x140005748	237	vfmadd213pd ymm26, k0, ymm21, ymm9			FMA
0x14000574e	237	vfmadd213pd ymm26, k0, ymm21, ymm10	0.016s	0.016s	FMA
0x140005754	237	vfmadd213pd ymm26, k0, ymm21, ymm11	0.016s	0.016s	FMA
0x14000575a	237	vfmadd213pd ymm26, k0, ymm21, ymm12	0.048s	0.048s	FMA
0x140005760	237	vfmadd213pd ymm26, k0, ymm21, ymm13	0.110s	0.110s	FMA
0x140005766	237	vfmadd213pd ymm26, k0, ymm21, ymm14	0.126s	0.126s	FMA
0x14000576c	237	vfmadd213pd ymm26, k0, ymm21, ymm1	0.158s	0.158s	FMA
0x140005772	237	vdivpd ymm26, k0, ymm1, ymm26	0.154s	0.154s	Divisions
0x140005778	237	vaddpd ymm25, k0, ymm25, ymm23	0.707s	0.707s	
0x14000577e	237	vmovapd ymm27, k0, ymm15			
0x140005784	237	vfmadd213pd ymm27, k0, ymm21, ymm0			FMA
0x14000578a	237	vfmadd213pd ymm27, k0, ymm21, ymm16			FMA
0x140005790	237	vfmadd213pd ymm27, k0, ymm21, ymm18			FMA
0x140005796	237	vfmadd213pd ymm27, k0, ymm21, ymm20			FMA
0x14000579c	237	vfmadd213pd ymm27, k0, ymm21, ymm22			FMA
0x1400057a2	237	vfmadd213pd ymm27, k0, ymm21, ymm17			FMA
0x1400057a8	237	vfmadd213pd ymm27, k0, ymm21, ymm24	0.013s	0.013s	FMA
0x1400057ae	237	vmulpd ymm21, k0, ymm27, ymm26			
0x1400057b4	237	vcmppd k1, k0, ymm25, ymm4, 0x1	0.094s	0.094s	
0x1400057bb	237	vxorpd ymm21, k1, ymm21, ymm31			
0x1400057c1	237	vpcmpud k1, k0, xmm6, xmm19, 0x1			
0x1400057c8	237	vscatterdpd qword ptr [rax+xmm6*8], k1, ymm21			Scatters
0x1400057cf	237	add rdx, 0x10	0.222s	0.222s	
0x1400057d3	237	cmp esi, edx			
0x1400057d5	237	jl 0x140004c34			
0x1400057db		**Block 2: 92020000**			

Roofline



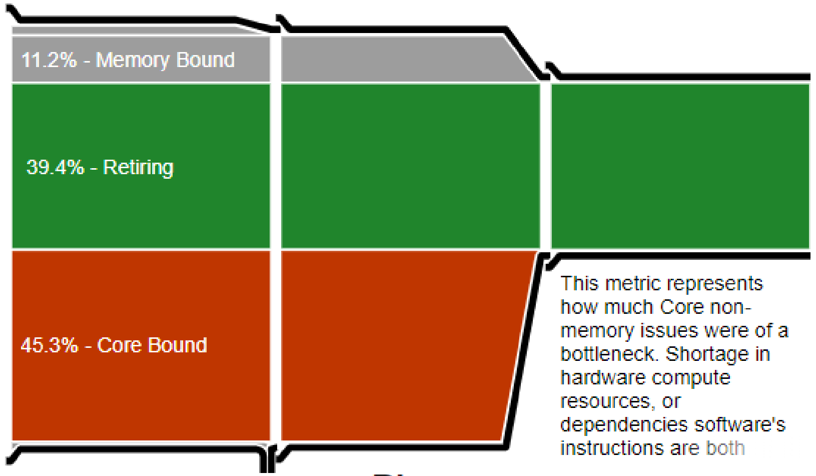
No performance recommendations are available based on current Roofline data

GFLOPS: 8.13
GINTOPS: 0.17
AVX-512 Mask Usage: 74%

Micro architecture for AVX2 version

Elapsed Time: 3.721s

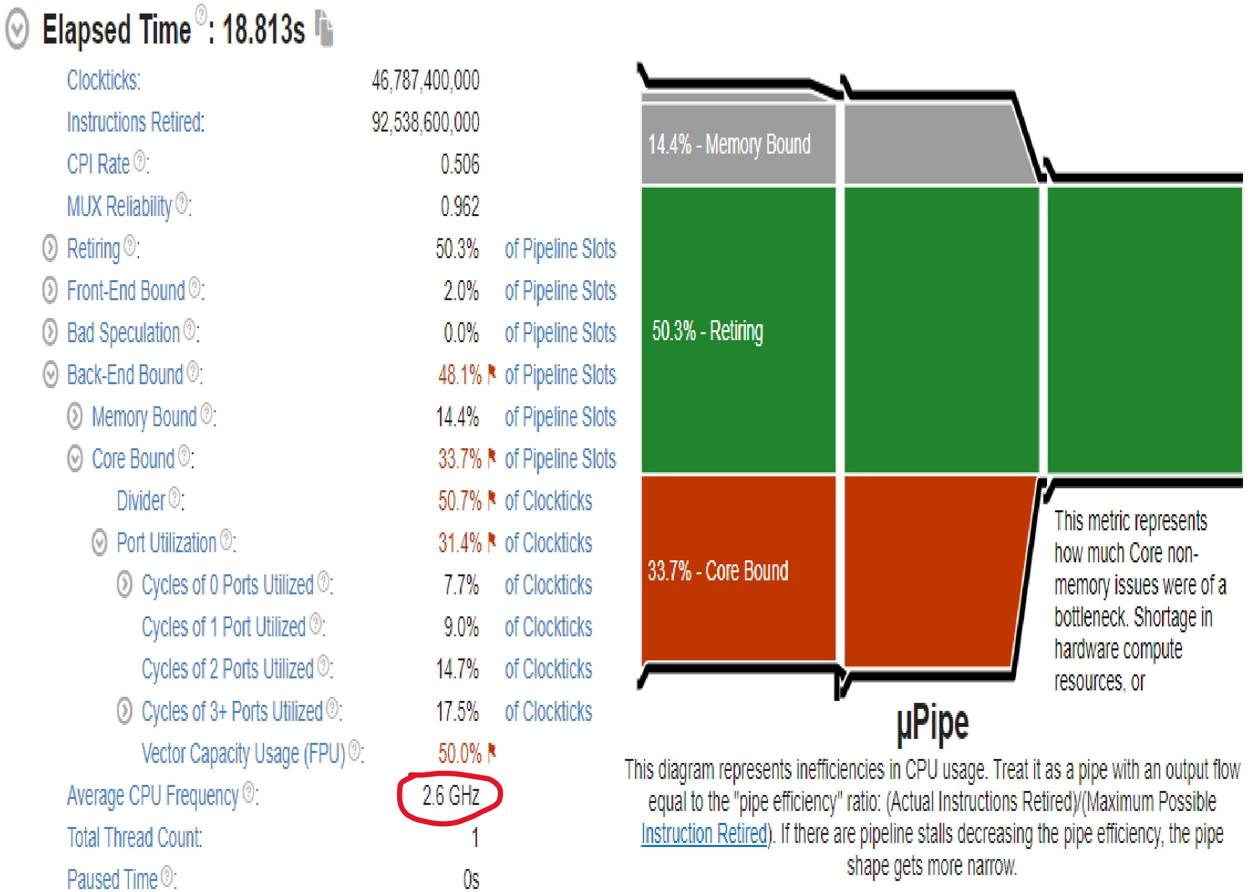
Clockticks:	5,700,200,000	
Instructions Retired:	8,357,800,000	
CPI Rate:	0.682	
MUX Reliability:	0.918	
Retiring:	39.4%	of Pipeline Slots
Light Operations:	33.6%	of Pipeline Slots
Heavy Operations:	5.8%	of Pipeline Slots
Front-End Bound:	2.3%	of Pipeline Slots
Bad Speculation:	1.8%	of Pipeline Slots
Branch Mispredict:	0.0%	of Pipeline Slots
Machine Clears:	1.8%	of Pipeline Slots
Back-End Bound:	56.5%	of Pipeline Slots
Memory Bound:	11.2%	of Pipeline Slots
Core Bound:	45.3%	of Pipeline Slots
Divider:	51.9%	of Clockticks
Port Utilization:	49.5%	of Clockticks
Cycles of 0 Ports Utilized:	5.6%	of Clockticks
Cycles of 1 Port Utilized:	19.3%	of Clockticks
Cycles of 2 Ports Utilized:	13.2%	of Clockticks
Cycles of 3+ Ports Utilized:	8.6%	of Clockticks
Vector Capacity Usage (FPU):	100.0%	
Average CPU Frequency:	1.6 GHz	
Total Thread Count:	1	
Paused Time:	0s	

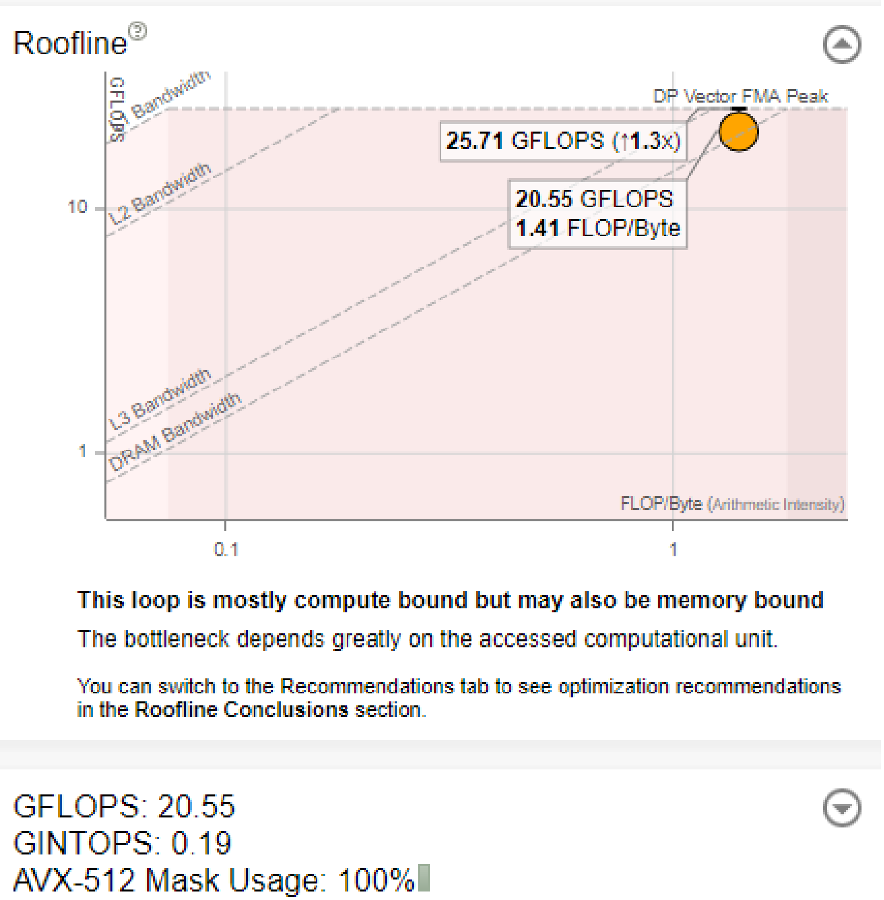


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

AVX2 Acklam- main region a bit quicker





Φ^{-1}

- We can create our own vectorised math functions that are very performant.
- We can make our own trade offs on the approximation accuracy
- We get faster code if we traverse and move less memory
- Filtering to contiguous, indexed view can be useful for handling branch conditions
- Predicting if a function will make the chip hot enough to slow down is not a good game for software engineers.
- Using SIMD types with generic lambdas creates very fast code.

Its not all about instruction set

- We Cheated
- SVML takes a _m512d with 8 doubles 0.0->1.0 and returns a _m512d of transformed.
- We take a vector of arbitrary length and structure our evaluations
- The act of selecting a direction to vectorise in is choosing your inner loop (loop interchange)
- Exploiting how you sequence your calculations and how you lay out the data is down to you.
- Its the other 90% of going faster! (Mike Acton)

Resources

- code available on https://github.com/andyD123/DR3
- Contact andreedrakeford@hotmail.com