Optimizing Matrix-Vector Calculations

Computation

$$\begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix} \times \begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

$$= \begin{bmatrix} R_0 = M_{0,0} * V_0 + M_{0,1} * V_1 + M_{0,2} * V_2 + M_{0,3} * V_3 \\ R_1 = M_{1,0} * V_0 + M_{1,1} * V_1 + M_{1,2} * V_2 + M_{1,3} * V_3 \\ R_2 = M_{2,0} * V_0 + M_{2,1} * V_1 + M_{2,2} * V_2 + M_{2,3} * V_3 \\ R_3 = M_{3,0} * V_0 + M_{3,1} * V_1 + M_{3,2} * V_2 + M_{3,3} * V_3 \end{bmatrix}$$

Computation

- typedef float matrix4[16]; ← Column or Row Major.
- typedef float vector4[4];

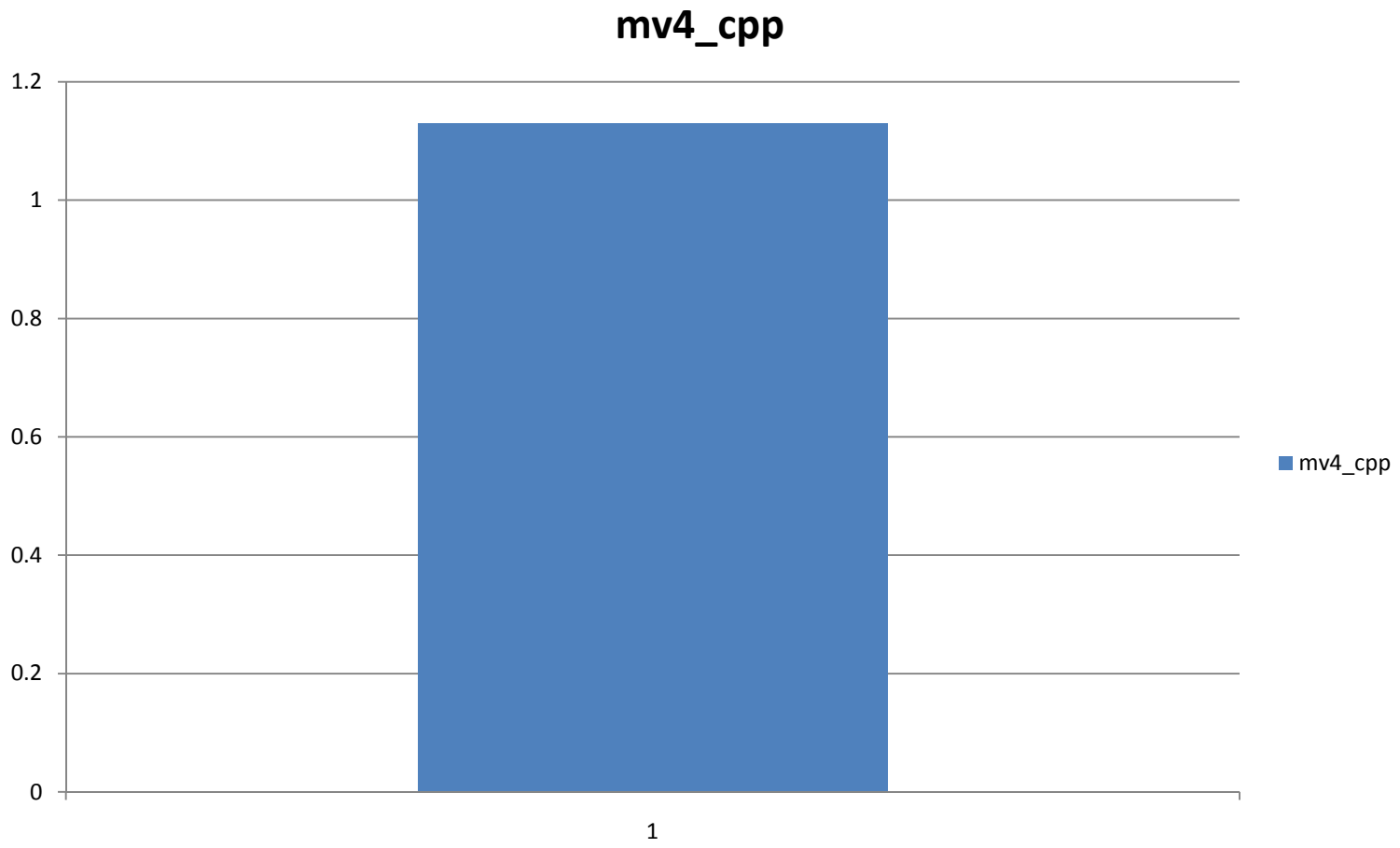
C++ Approach

- Simple Approach:

```
void mat4_vec4_multiply(vector4 &dst, const matrix4 &a, const vector4 &b)
{
        size_t matrix_index = 0;

        for(size_t i = 0; i < 3; ++i)
        {
                dst[i] = 0.0f;
                for(size_t j = 0; j < 3; ++j)
                {
                        dst[i] += a[matrix_index] * b[j];
                        ++matrix_index;
                }
        }
}
```

C++ Approach

256K Vertices, 100 repetitions



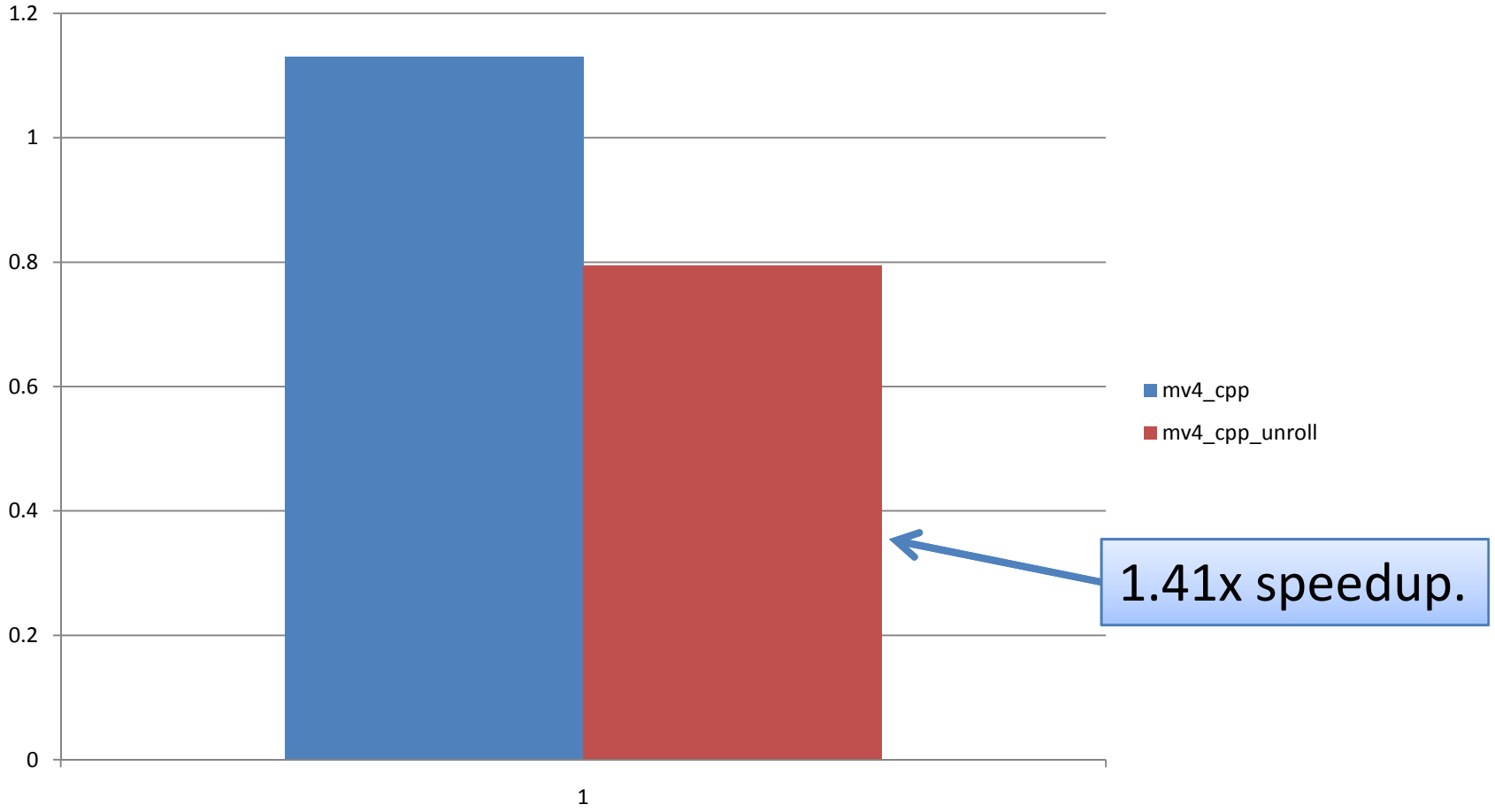
C++ Approach

- Unroll

```
void mat4_vec4_multiply_unroll(vector4 &dst, const matrix4 &a, const vector4 &b)
{
        dst[0] = a[0]*b[0]  + a[1]*b[1]  + a[2]*b[2]  + a[3]*b[3];
        dst[1] = a[4]*b[0]  + a[5]*b[1]  + a[6]*b[2]  + a[7]*b[3];
        dst[2] = a[8]*b[0]  + a[9]*b[1]  + a[10]*b[2] + a[11]*b[3];
        dst[3] = a[12]*b[0] + a[13]*b[1] + a[14]*b[2] + a[15]*b[3];
}
```

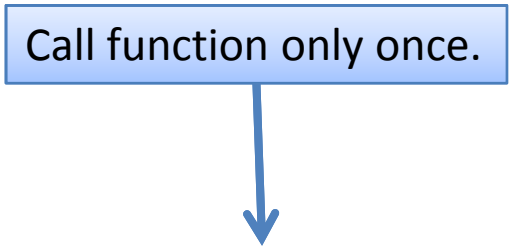
C++ Approach

256K Vertices, 100 repetitions



C++ Approach

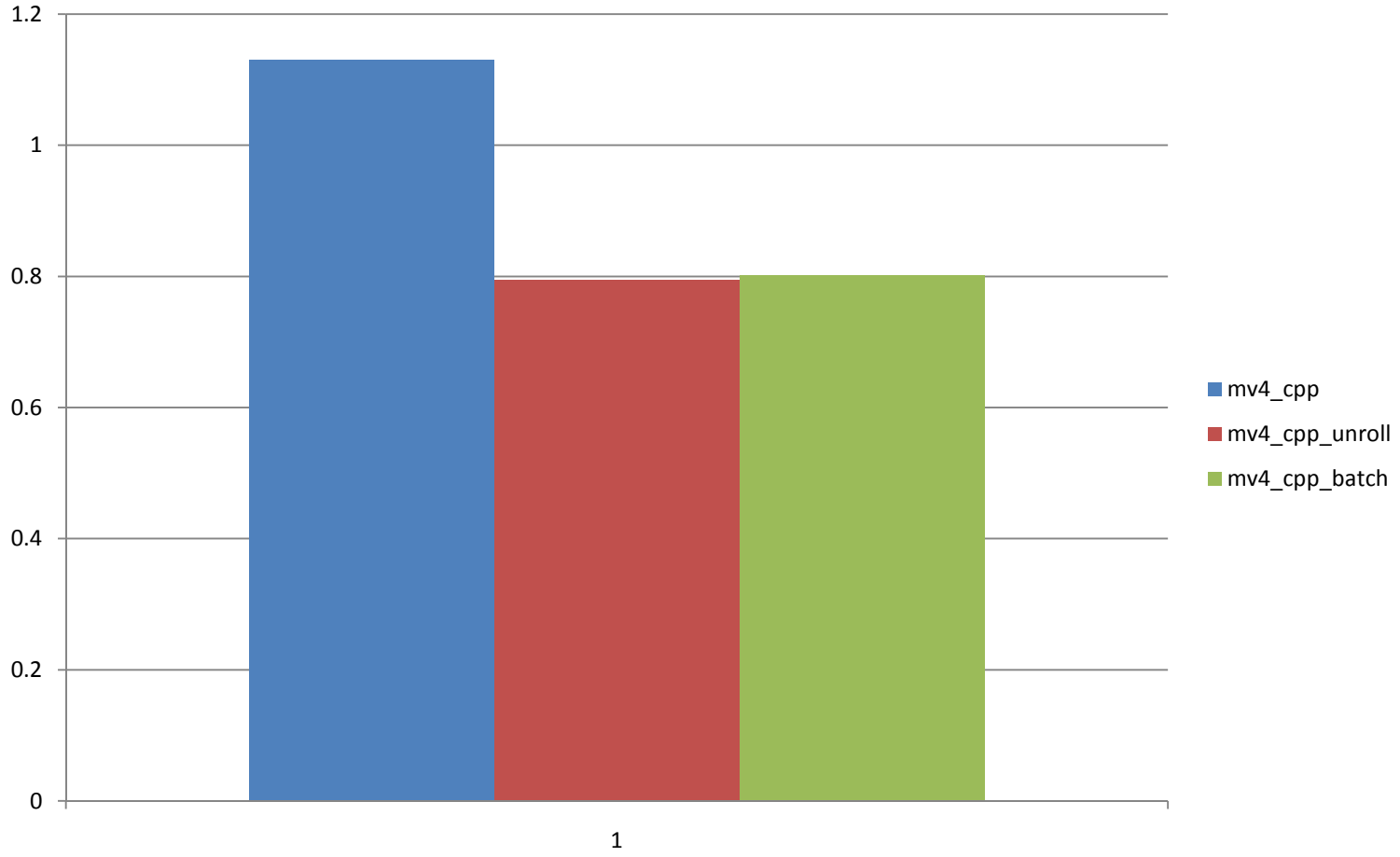
- Batch




```
float test_mv4_batch(vector4 *dst, const matrix4 &m, const vector4 *src, size_t size)
{
        for(size_t i = 0; i < size; ++i)
        {
                dst[i][0] = m[0]*src[i][0]  + m[1]*src[i][1]  + m[2]*src[i][2]  + m[3]*src[i][3];
                dst[i][1] = m[4]*src[i][0]  + m[5]*src[i][1]  + m[6]*src[i][2]  + m[7]*src[i][3];
                dst[i][2] = m[8]*src[i][0]  + m[9]*src[i][1]  + m[10]*src[i][2] + m[11]*src[i][3];
                dst[i][3] = m[12]*src[i][0] + m[13]*src[i][1] + m[14]*src[i][2] + m[15]*src[i][3];
        }
}
```

C++ Approach

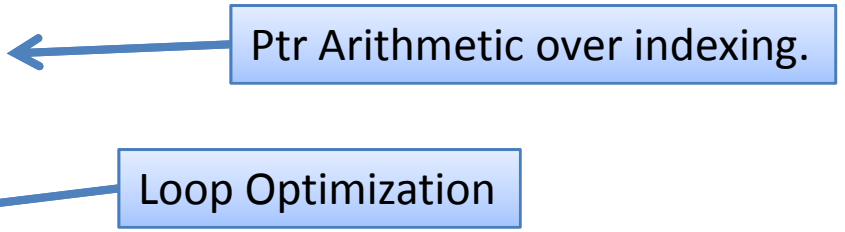
256K Vertices, 100 repetitions



C++ Approach

- Batch Ptr

```
void mat4_vec4_multiply_ptr(vector4 *dst, const matrix4 &m, const vector4 *src, size_t size)
{
        vector4 *d = dst;
        const vector4 *a = src;
        size_t tmp_size = size;
        while(tmp_size--)
        {
                *d[0] = m[0]*(*a[0])  + m[1]*(*a[1])  + m[2]*(*a[2])  + m[3]*(*a[3]);
                *d[1] = m[4]*(*a[0])  + m[5]*(*a[1])  + m[6]*(*a[2])  + m[7]*(*a[3]);
                *d[2] = m[8]*(*a[0])  + m[9]*(*a[1])  + m[10]*(*a[2]) + m[11]*(*a[3]);
                *d[3] = m[12]*(*a[0]) + m[13]*(*a[1]) + m[14]*(*a[2]) + m[15]*(*a[3]);
                ++d;
                ++a;
        }
}
```



C++ Approach

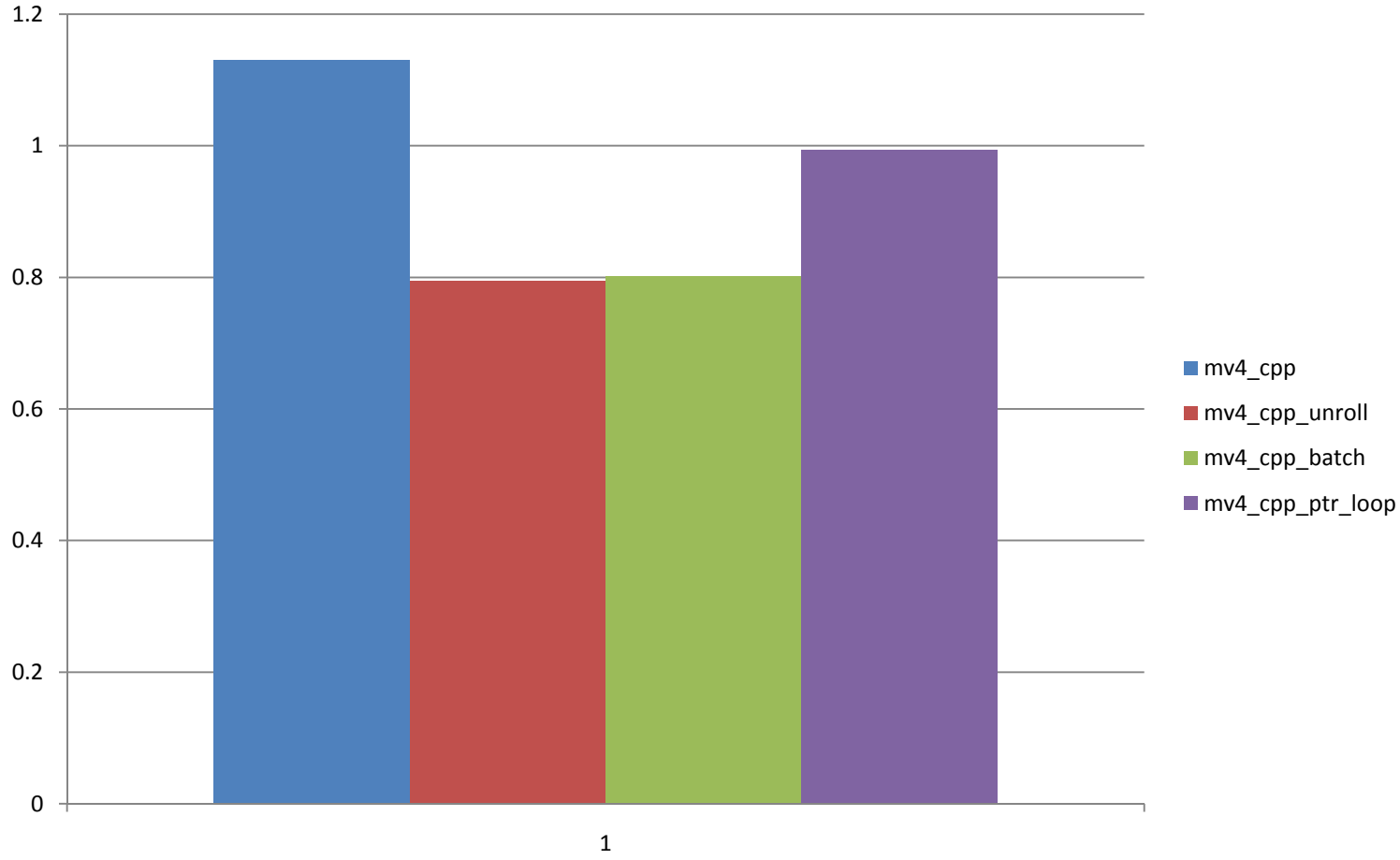
- Loop Optimization:

```
for(size_t i = 0; i < size; ++i)

jmp       test_mv4_batch+5Eh (13832AEh)
mov       eax,dword ptr [i]
add       eax,1
mov       dword ptr [i],eax
mov       eax,dword ptr [i]
cmp       eax,dword ptr [size]
jae       test_mv4_batch+215h (1383465h)
…
sub       dword ptr [esp+10h],1
jne       test_mv4_batch+3Fh (402B5Fh)
```

```
while(tmp_size--)

test      edi,edi
je        test_mv4_batch+0A7h (13924C7h)
jmp       test_mv4_batch+60h (1392480h)
lea       ebx,[ebx]
…

dec       ecx
jne       test_mv4_batch+60h (1392480h)
```

C++ Approach

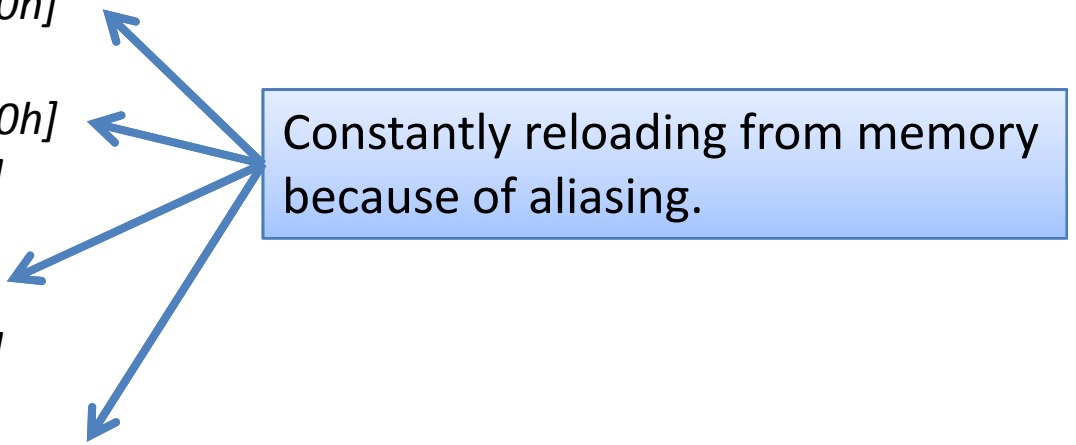
256K Vertices, 100 repetitions



C++ Approach

```
*d[0] = m[0]*(*a[0])  + m[1]*(*a[1])  + m[2]*(*a[2])  + m[3]*(*a[3]);
```

```
movss     xmm0,dword ptr [eax-20h]
mulss     xmm0,dword ptr [ecx]
movss     xmm1,dword ptr [eax-10h]
mulss     xmm1,dword ptr [ecx+4]
addss     xmm0,xmm1
movss     xmm1,dword ptr [eax]
mulss     xmm1,dword ptr [ecx+8]
addss     xmm0,xmm1
movss     xmm1,dword ptr [eax+10h]
mulss     xmm1,dword ptr [ecx+0Ch]
addss     xmm0,xmm1
movss     dword ptr [esi],xmm0
```



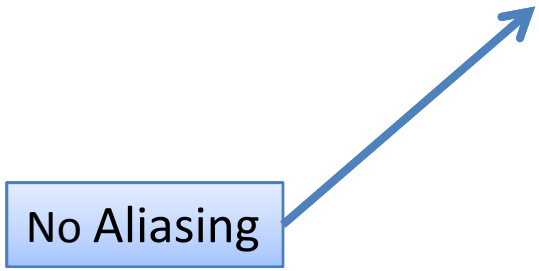
C++ Approach

```
vector4 *d = dst;
const vector4 *a = src;
size_t tmp_size = size;

float ax, ay, az, aw;

while(tmp_size--)
{
        ax = *a[0]; ay = *a[1]; az = *a[2]; aw = *a[3];

        *d[0] = m[0]*ax  + m[1]*ay  + m[2]*az  + m[3]*aw;
        *d[1] = m[4]*ax  + m[5]*ay  + m[6]*az  + m[7]*aw;
        *d[2] = m[8]*ax  + m[9]*ay  + m[10]*az + m[11]*aw;
        *d[3] = m[12]*ax + m[13]*ay + m[14]*az + m[15]*aw;

        ++d;
        ++a;
}
```

C++ Approach

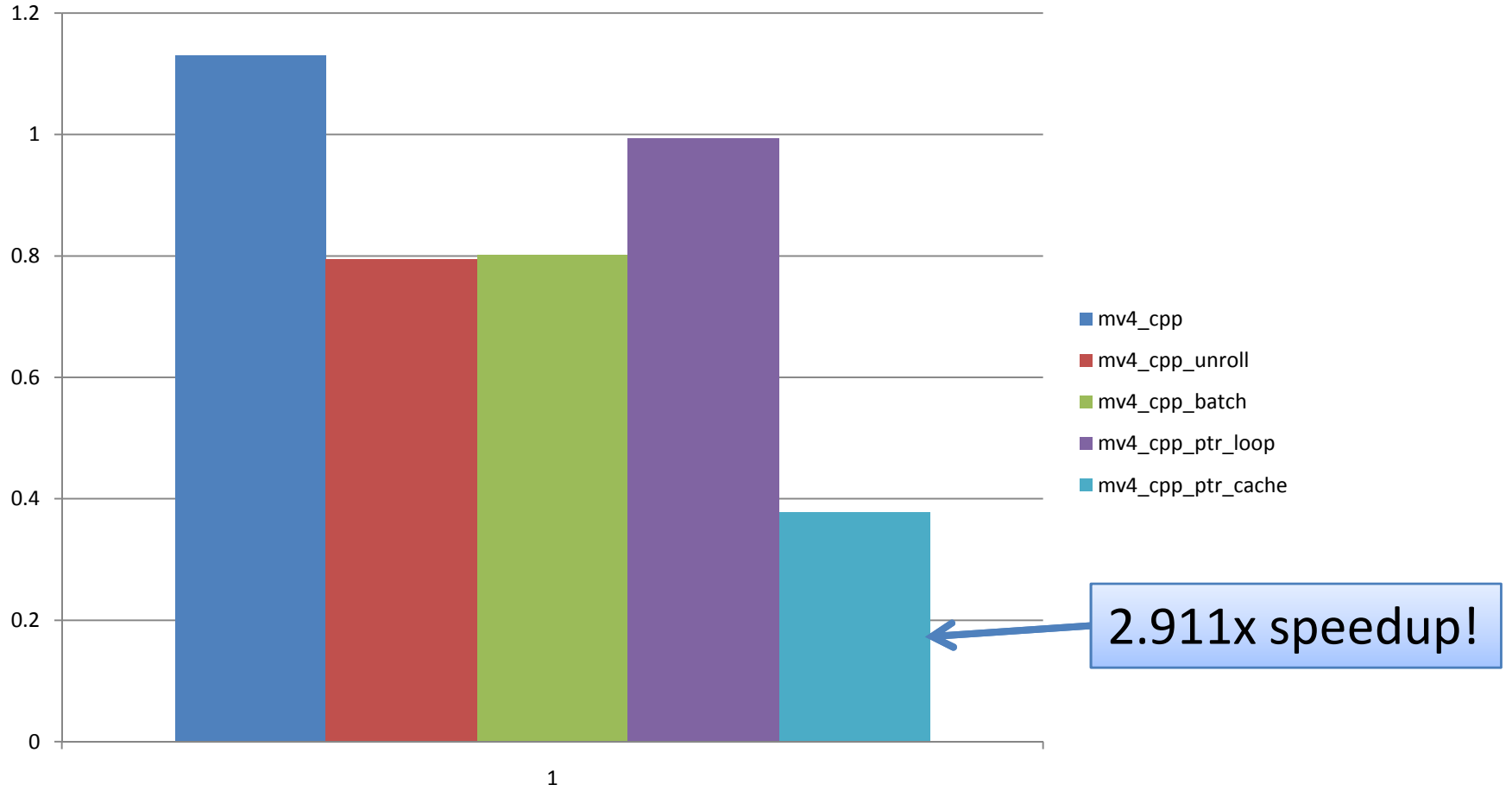
```
const float ax = *a[0];
movss      xmm1,dword ptr [ecx-20h]

const float ay = *a[1];
movss      xmm0,dword ptr [ecx-10h]

const float az = *a[2];
movss      xmm2,dword ptr [ecx]

const float aw = *a[3];
movss      xmm3,dword ptr [ecx+10h]
```

```
*d[0] = m[0]*ax  + m[1]*ay  + m[2]*az  + m[3]*aw;

movss      xmm4,dword ptr [eax+4]
movss      xmm5,dword ptr [eax]
mulss      xmm5,xmm1
mulss      xmm4,xmm0
addss      xmm4,xmm5
movss      xmm5,dword ptr [eax+8]
mulss      xmm5,xmm2
addss      xmm4,xmm5
movaps       xmm5,xmm3
mulss      xmm5,dword ptr [eax+0Ch]
addss      xmm4,xmm5
movss      dword ptr [edx-30h],xmm4
```



C++ Approach

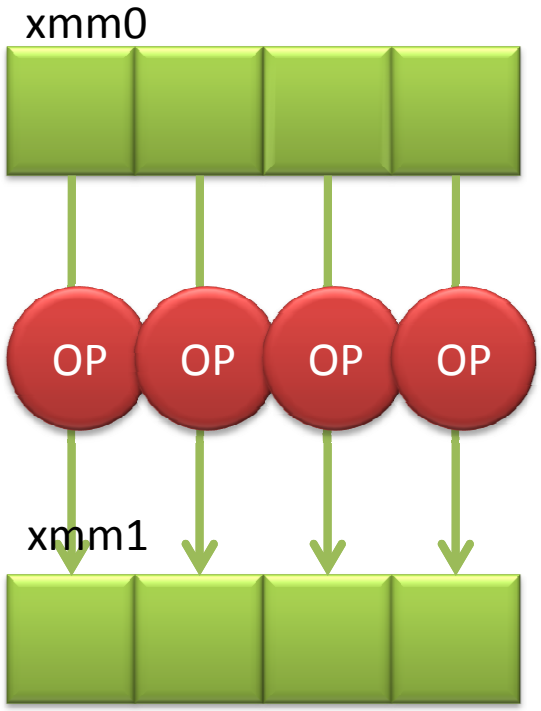
256K Vertices, 100 repetitions

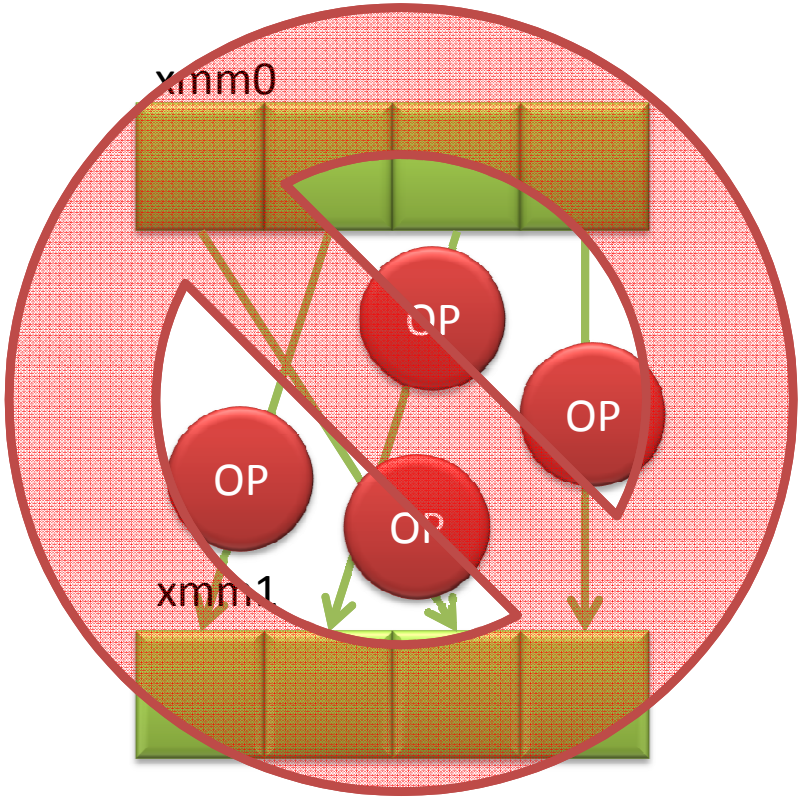


SSE Approach

- SSE provides 8 (xmm0 – xmm7) 128 bit registers:
  - 4/32bit registers.
  - 2/64 bit registers.
- Component wise operations.
  - Some shuffling, but slow.

SSE Approach





SSE Approach

- Ex:
  - addps xmm0, xmm2
    - xmm0[0] = xmm0[0] + xmm2[0]
    - xmm0[1] = xmm0[1] + xmm2[1]
    - xmm0[2] = xmm0[2] + xmm2[2]
    - xmm0[3] = xmm0[3] + xmm2[3]

SSE Approach

- Ex:
  - mulps xmm0, xmm2
    - xmm0[0] = xmm0[0] * xmm2[0]
    - xmm0[1] = xmm0[1] * xmm2[1]
    - xmm0[2] = xmm0[2] * xmm2[2]
    - xmm0[3] = xmm0[3] * xmm2[3]

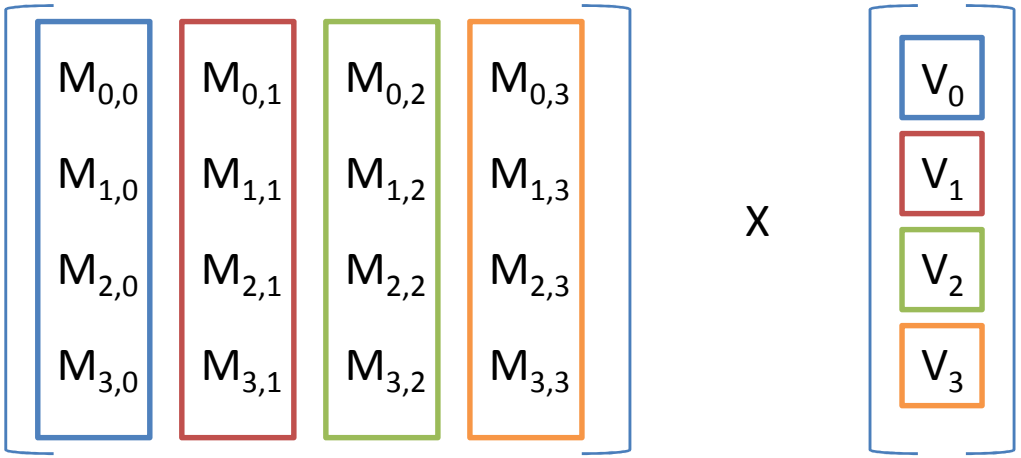
SSE Approach

- Ex:
  - shufps xmm0, xmm2, BIT_MASK
    - xmm0[0] = xmm2[BIT_SELECT3]
    - xmm0[1] = xmm2[BIT_SELECT2]
    - xmm0[2] = xmm0[BIT_SELECT1]
    - xmm0[3] = xmm0[BIT_SELECT0]

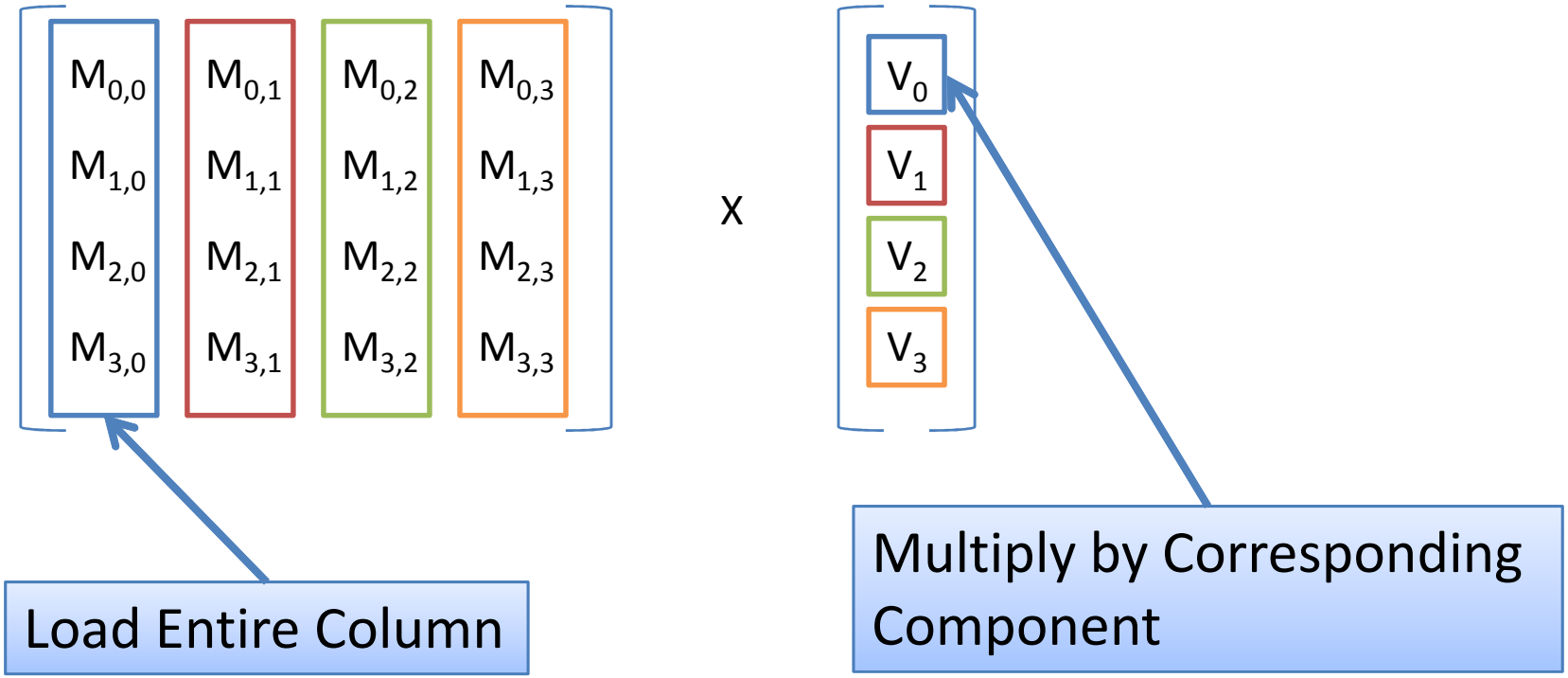
SSE Approach

- Ex:
  - shufps xmm0, xmm0, 0
    - xmm0[0] = xmm0[0]
    - xmm0[1] = xmm0[0]
    - xmm0[2] = xmm0[0]
    - xmm0[3] = xmm0[0]
  - In other words, take the value in xmm0 and copy to other registers.

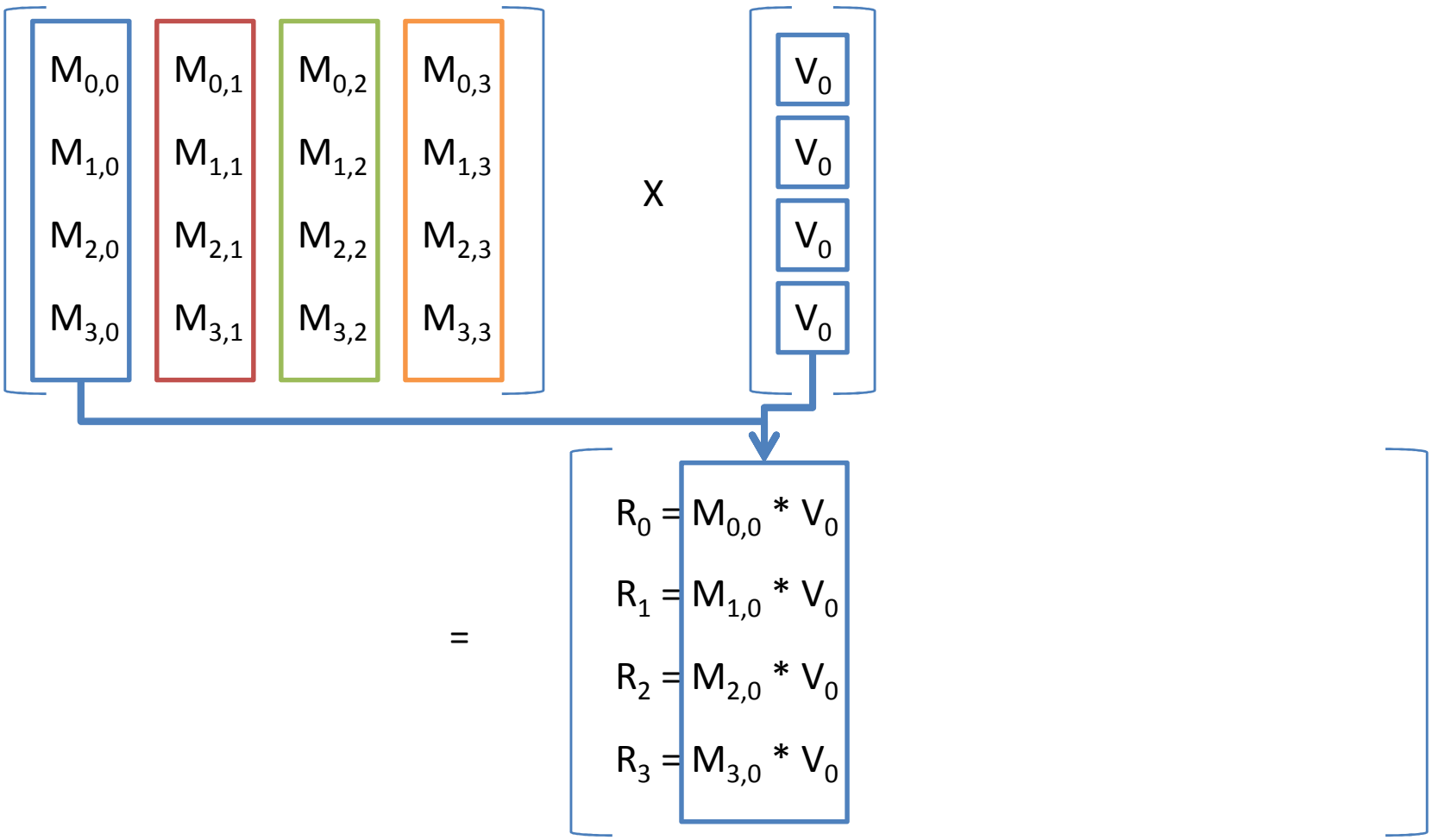
SSE Approach



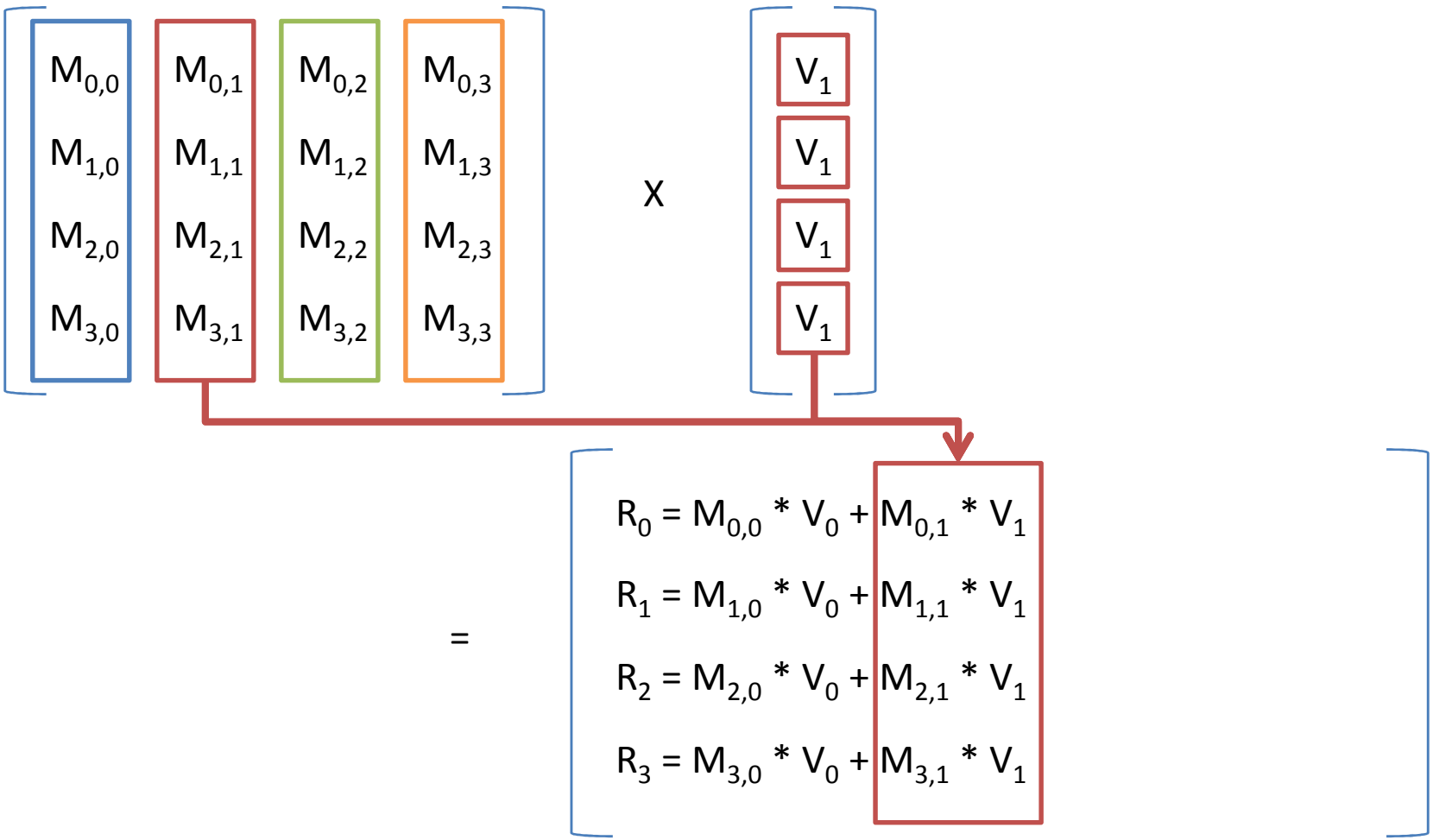
SSE Approach



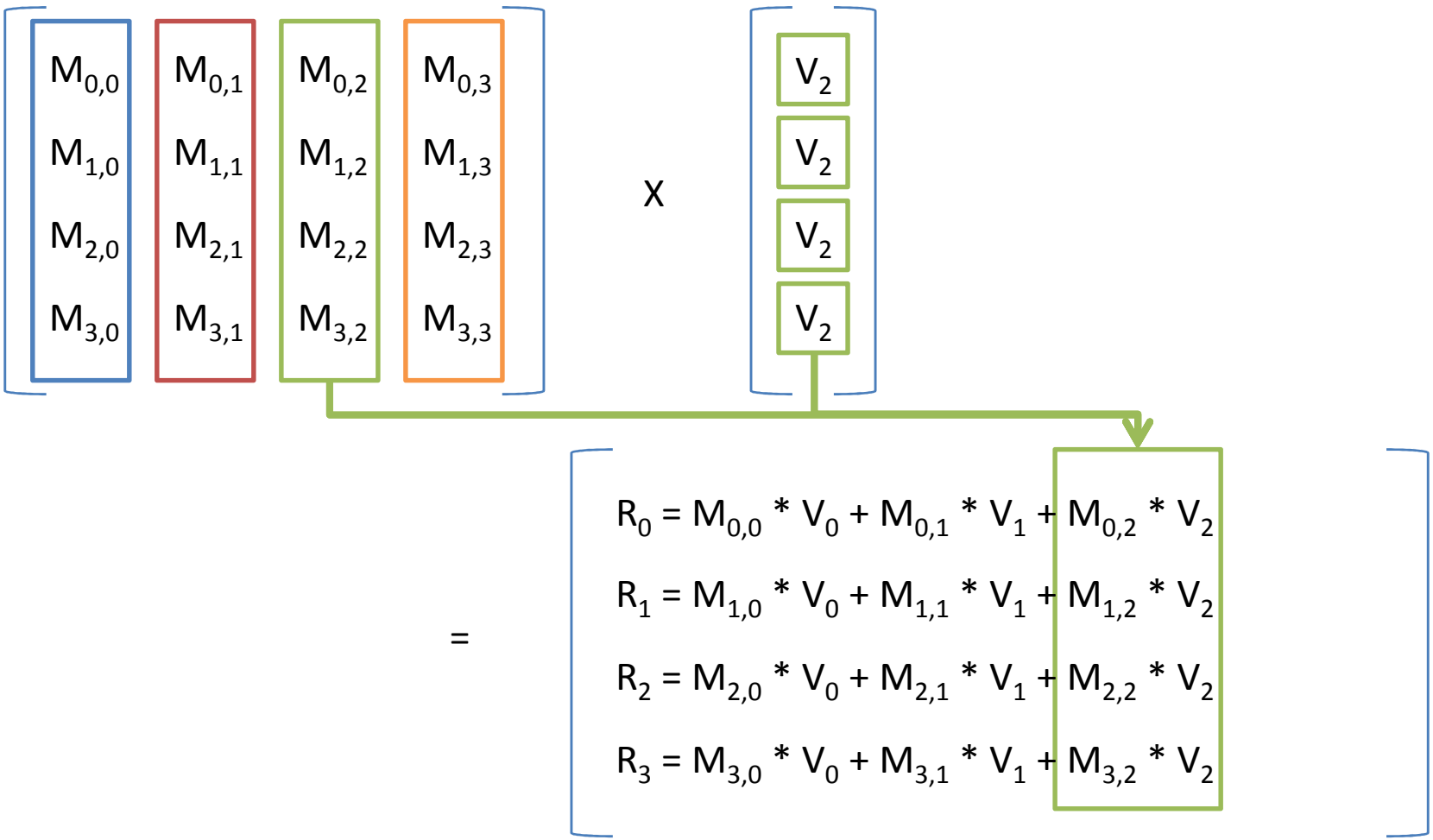
SSE Approach



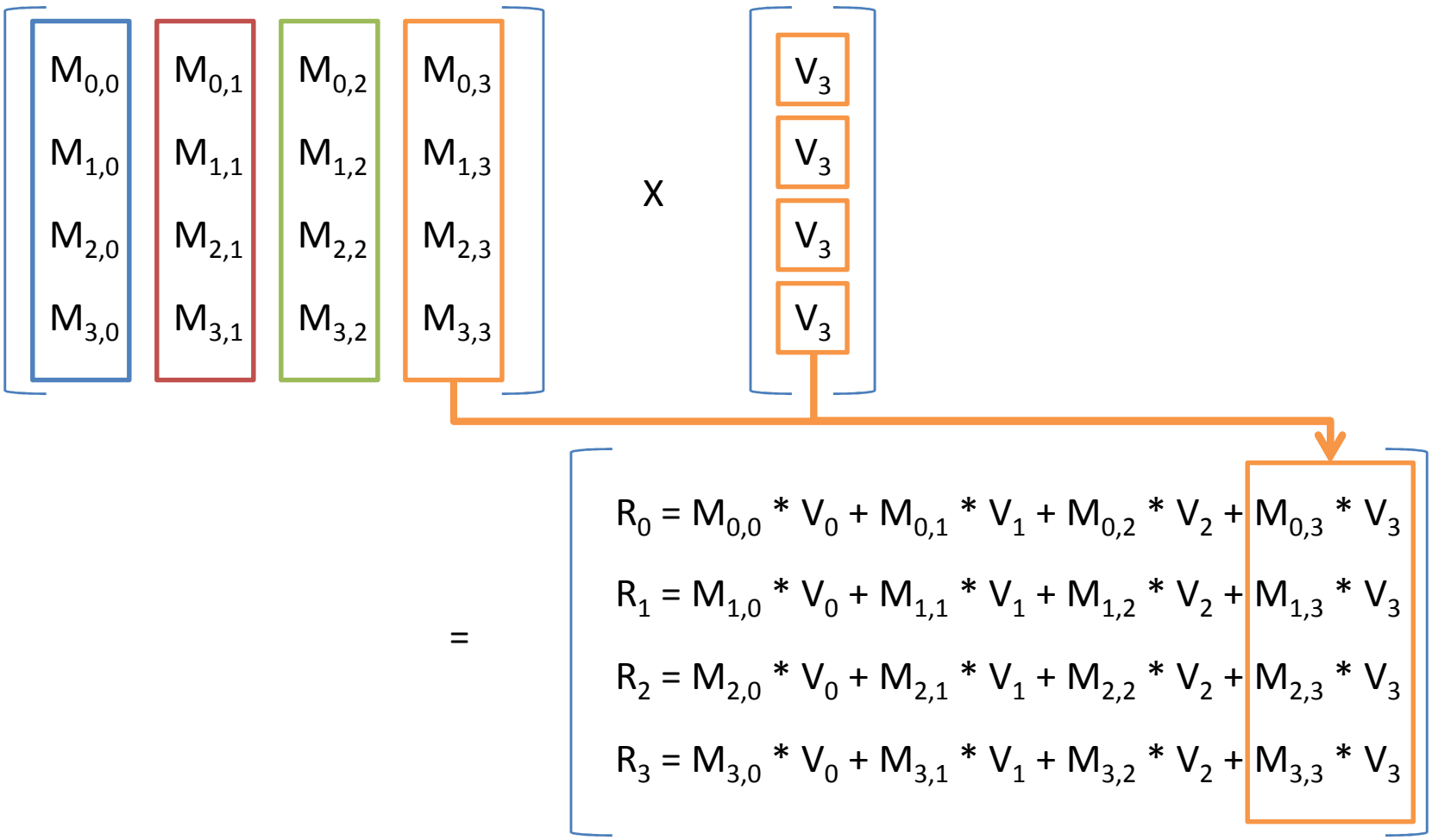
SSE Approach



SSE Approach



SSE Approach



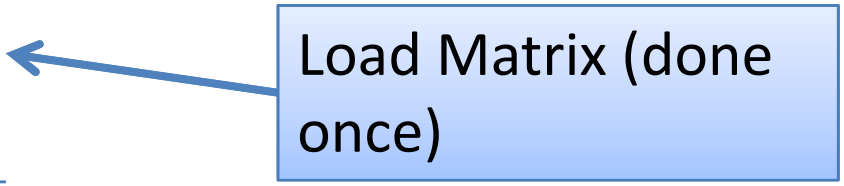
SSE Approach

```
const __m128 m0 = _mm_loadu_ps(m);
const __m128 m1 = _mm_loadu_ps(m + 4);
const __m128 m2 = _mm_loadu_ps(m + 8);
const __m128 m3 = _mm_loadu_ps(m + 12);
```

```
__m128 v0 = _mm_load_ps1(src++);
__m128 accum = _mm_mul_ps(v0, m0);

v0 = _mm_load_ps1(src++);
__m128  tmp = _mm_mul_ps(v0, m1);
accum = _mm_add_ps(accum, tmp);

v0 = _mm_load_ps1(src++);
tmp = _mm_mul_ps(v0, m2);
accum = _mm_add_ps(accum, tmp);

v0 = _mm_load_ps1(src++);
tmp = _mm_mul_ps(v0, m3);
accum = _mm_add_ps(accum, tmp);

_mm_storeu_ps(*dst, accum);
```

SSE Approach

```
mov       eax,dword ptr [m]
movups      xmm2,xmmword ptr [eax]
movups      xmm3,xmmword ptr [eax+10h]
movups      xmm4,xmmword ptr [eax+20h]
movups      xmm5,xmmword ptr [eax+30h]
movss      xmm1,dword ptr [eax]
movss      xmm0,dword ptr [eax+4]
shufps     xmm0,xmm0,0
shufps     xmm1,xmm1,0
mulps      xmm0,xmm3
mulps      xmm1,xmm2
addps      xmm0,xmm1
movss      xmm1,dword ptr [eax+8]
shufps     xmm1,xmm1,0
mulps      xmm1,xmm4
addps      xmm1,xmm0
movss      xmm0,dword ptr [eax+0Ch]
shufps     xmm0,xmm0,0
mulps      xmm0,xmm5
addps      xmm0,xmm1
movups      xmmword ptr [edx],xmm0
```



SSE Approach

Scalar:
51 instructions

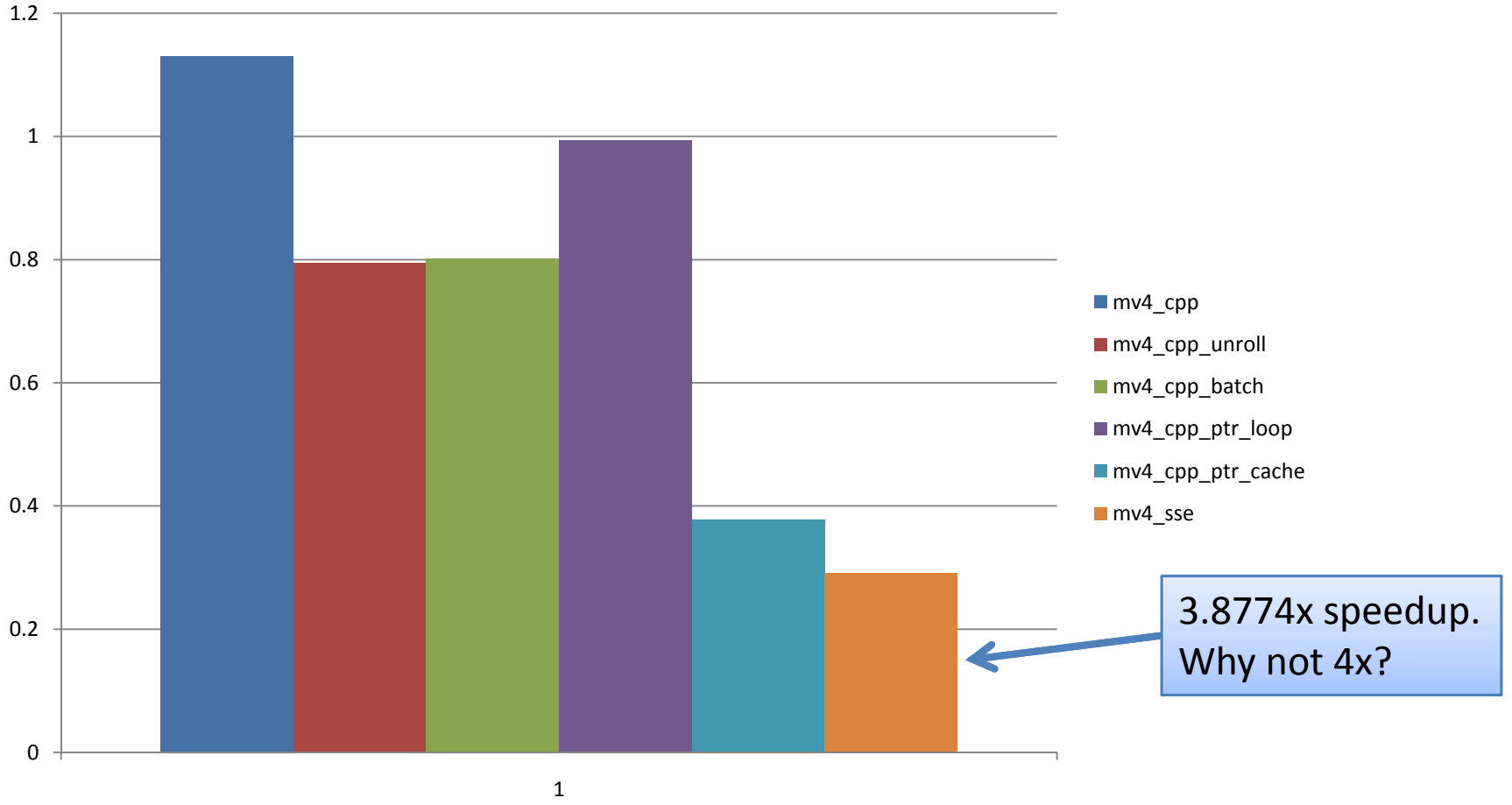
```
movss     xmm1,dword ptr [ecx-20h]
movss     xmm0,dword ptr [ecx-10h]
movss     xmm2,dword ptr [ecx]
movss     xmm4,dword ptr [eax+4]
movss     xmm5,dword ptr [eax]
movss     xmm3,dword ptr [ecx+10h]
mulss     xmm5,xmm1
mulss     xmm4,xmm0
addss     xmm4,xmm5
movss     xmm5,dword ptr [eax+8]
mulss     xmm5,xmm2
addss     xmm4,xmm5
movaps      xmm5,xmm3
mulss     xmm5,dword ptr [eax+0Ch]
addss     xmm4,xmm5
movss     dword ptr [edx-30h],xmm4
movss     xmm4,dword ptr [eax+14h]
movss     xmm5,dword ptr [eax+10h]
mulss     xmm5,xmm1
mulss     xmm4,xmm0
addss     xmm4,xmm5
movss     xmm5,dword ptr [eax+18h]
mulss     xmm5,xmm2
addss     xmm4,xmm5
movaps      xmm5,xmm3
mulss     xmm5,dword ptr [eax+1Ch]
addss     xmm4,xmm5
movss     dword ptr [edx-20h],xmm4
movss     xmm4,dword ptr [eax+24h]
movss     xmm5,dword ptr [eax+20h]
mulss     xmm5,xmm1
mulss     xmm4,xmm0
addss     xmm4,xmm5
movaps      xmm5,xmm2
mulss     xmm5,dword ptr [eax+28h]
addss     xmm4,xmm5
movss     xmm5,dword ptr [eax+2Ch]
mulss     xmm5,xmm3
addss     xmm4,xmm5
movss     dword ptr [edi+ecx],xmm4
mulss     xmm1,dword ptr [eax+30h]
mulss     xmm0,dword ptr [eax+34h]
addss     xmm0,xmm1
movss     xmm1,dword ptr [eax+38h]
mulss     xmm1,xmm2
addss     xmm0,xmm1
movss     xmm1,dword ptr [eax+3Ch]
mulss     xmm1,xmm3
addss     xmm0,xmm1
movss     dword ptr [edx],xmm0
add       ecx,10h
```

Vector:
13 instructions

```
movss     xmm1,dword ptr [eax]
movss     xmm0,dword ptr [eax+4]
shufps    xmm0,xmm0,0
shufps    xmm1,xmm1,0
mulps     xmm0,xmm3
mulps     xmm1,xmm2
addps     xmm0,xmm1
movss     xmm1,dword ptr [eax+8]
shufps    xmm1,xmm1,0
mulps     xmm1,xmm4
addps     xmm1,xmm0
movss     xmm0,dword ptr [eax+0Ch]
shufps    xmm0,xmm0,0
mulps     xmm0,xmm5
addps     xmm0,xmm1
movups      xmmword ptr [edx],xmm0
```

SSE Approach

256K Vertices, 100 repetitions

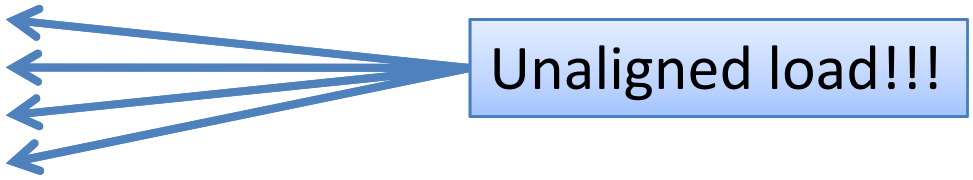


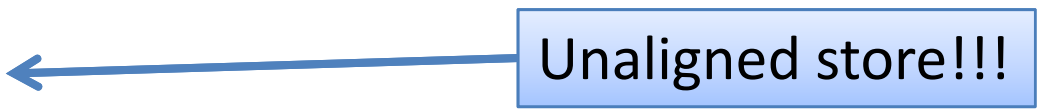
SSE Approach

- movups:
  - Moves data from memory into a xmm register. Address does not have to be aligned.
- movaps:
  - Moves data from memory into a xmm register. Address *MUST* be 16 bytes aligned.
- movaps 6-17 cycles faster than movups.

SSE Approach

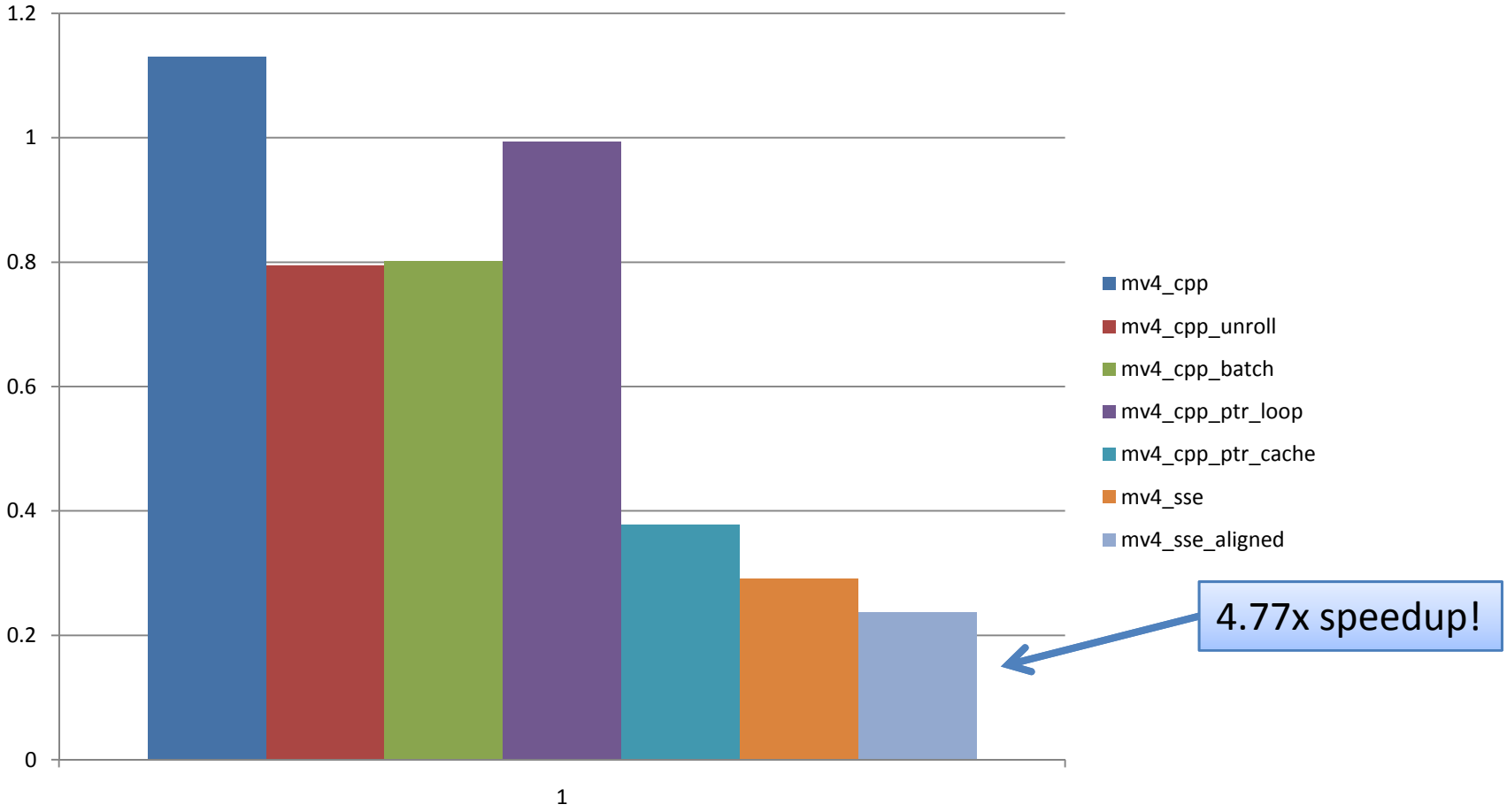
```
mov       eax,dword ptr [m]
movups     xmm2,xmmword ptr [eax]
movups     xmm3,xmmword ptr [eax+10h]
movups     xmm4,xmmword ptr [eax+20h]
movups     xmm5,xmmword ptr [eax+30h]
movss      xmm1,dword ptr [eax]
movss      xmm0,dword ptr [eax+4]
shufps     xmm0,xmm0,0
shufps     xmm1,xmm1,0
mulps      xmm0,xmm3
mulps      xmm1,xmm2
addps      xmm0,xmm1
movss      xmm1,dword ptr [eax+8]
shufps     xmm1,xmm1,0
mulps      xmm1,xmm4
addps      xmm1,xmm0
movss      xmm0,dword ptr [eax+0Ch]
shufps     xmm0,xmm0,0
mulps      xmm0,xmm5
addps      xmm0,xmm1
movups     xmmword ptr [edx],xmm0
```





SSE Approach

256K Vertices, 100 repetitions



MT Approach

- Can still go faster by using multiple threads!
- Each thread should multiply N vectors with matrix M.
  - Referred as grain size.

MT Approach

```
const int sfull = (int)(size / grain_size);

#pragma omp parallel for
for(int i = 0; i < sfull; ++i)
{
        mv4_batch_cache_ptr(dst + i * grain_size, m, src + i * grain_size, grain_size);
}

const int rest = size % grain_size;
mv4_batch_cache_ptr(dst + sfull * grain_size, m, src + sfull * grain_size, rest);
```

Get # full grain size iterations.

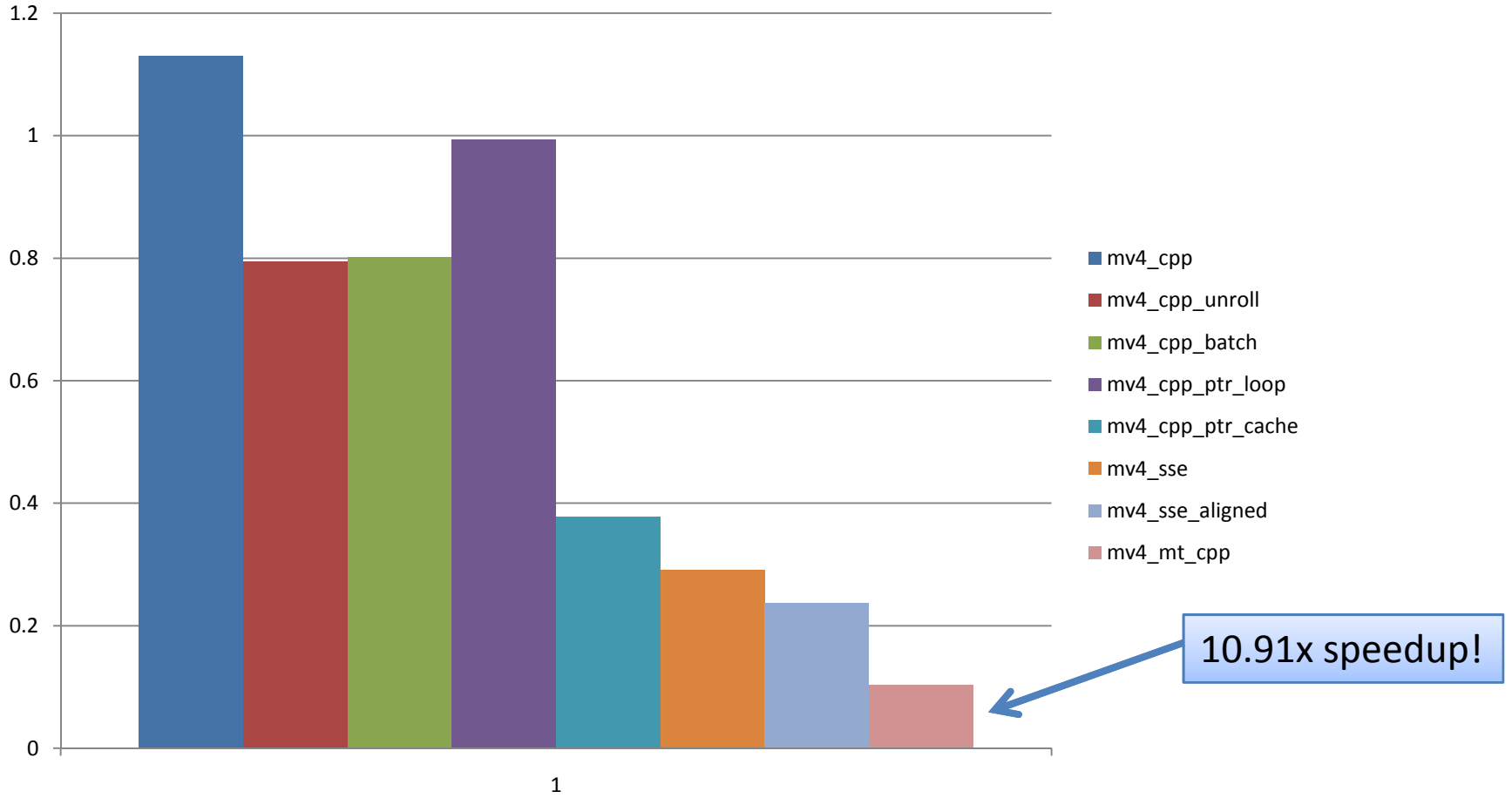
Process using multiple threads.

Invoke normal C++ matrix-vector multiply

Process left overs.

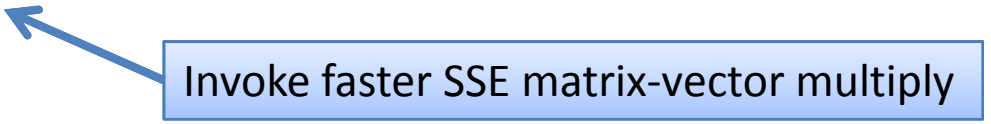
MT Approach

256K Vertices, 100 repetitions, Grain Size = 32



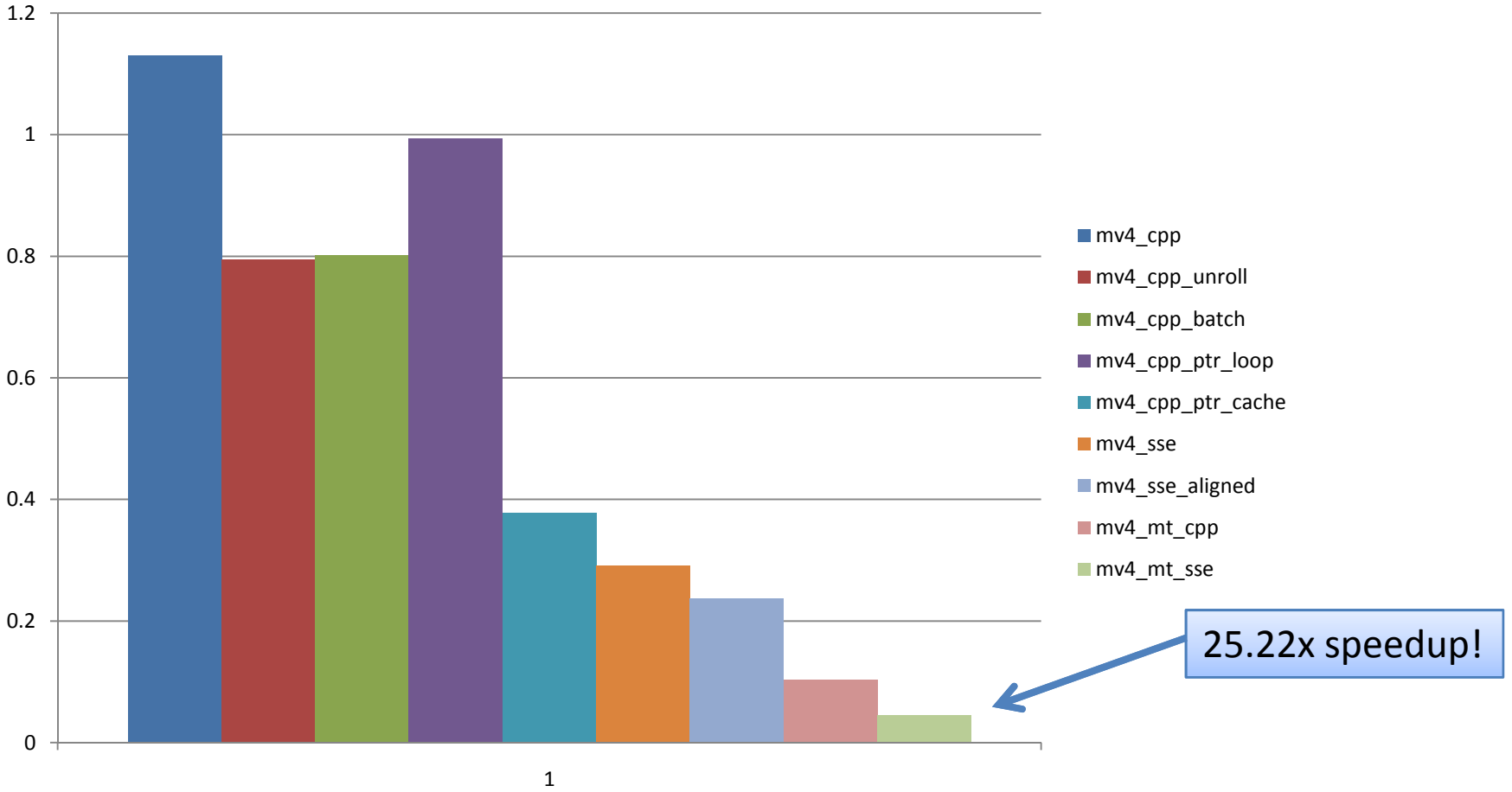
MT Approach

```
const int sfull = (int)(size / grain_size);

#pragma omp parallel for
for(int i = 0; i < sfull; ++i)
{
        mv4_sse2_aligned (dst + i * grain_size, m, src + i * grain_size, grain_size);
}

const int rest = size % grain_size;
mv4_batch_cache_ptr(dst + sfull * grain_size, m, src + sfull * grain_size, rest);
```



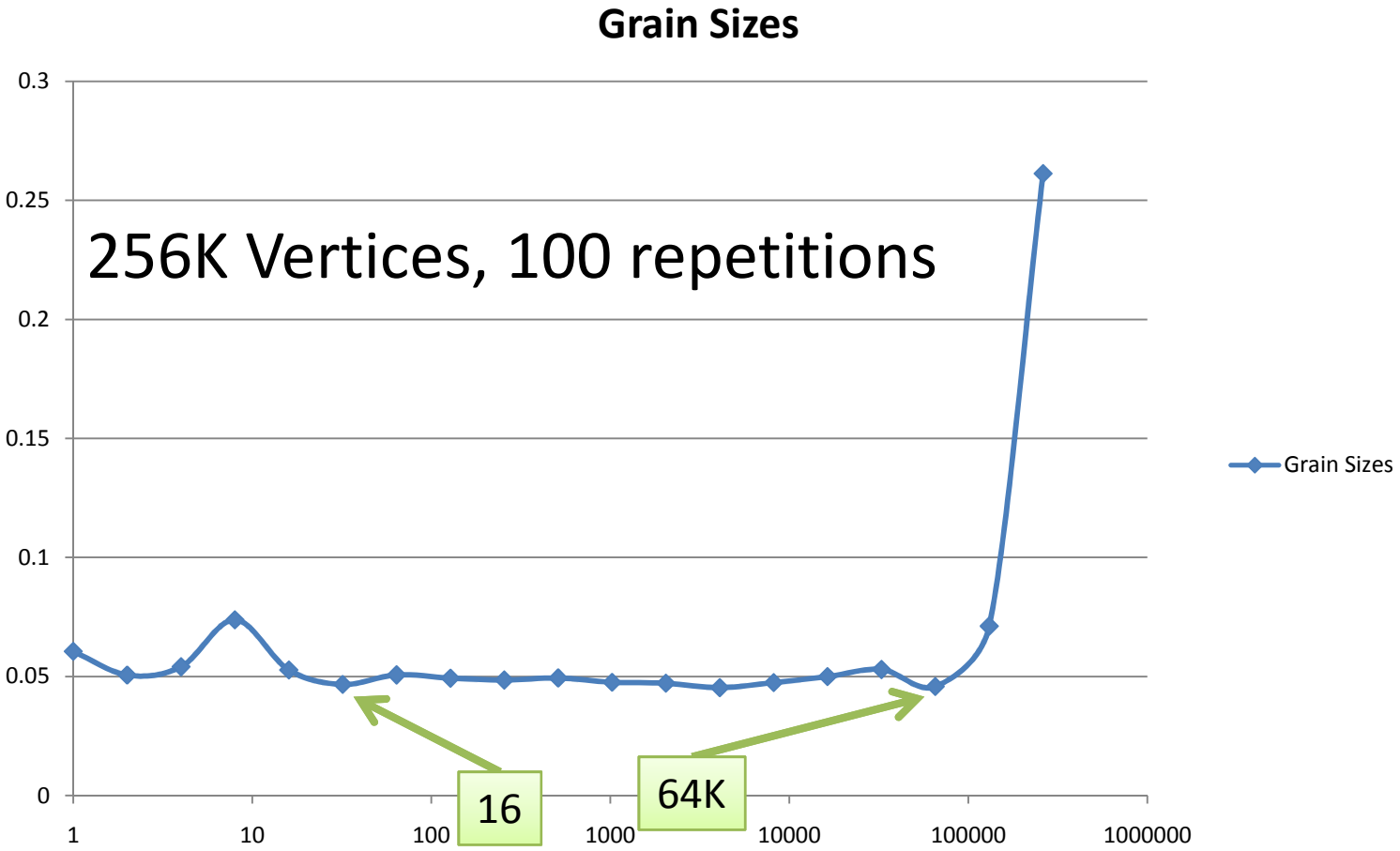
MT Approach

256K Vertices, 100 repetitions, Grain Size = 32



MT Approach

- How to pick grain size?



CUDA Approach

- If N threads gave us large speedup, then M>>N should give us even larger speedup.
- CUDA can run upwards of 1024 threads.
  - On NVIDIA's latest and greatest, around 32K threads.

CUDA Approach

- Each thread gets 1 vertex.
- Assign as many threads as there are vertices.
- If # vertices > max # threads, do multiple runs with subset of vertices.

CUDA Approach

```
bool mv4_send_data(
        const matrix4 *host_m, const matrix4 *dev_m,
        const vector4 *host_src, const vector4 *dev_src,
        size_t size)
{
        if (cudaMemcpy(
                (void*)dev_src, (void*)host_src, size * sizeof(vector4),
                cudaMemcpyHostToDevice))
                return false;

        if (cudaMemcpy(
                (void*)dev_m, (void*)host_m, sizeof(matrix4),
                cudaMemcpyHostToDevice))
                return false;

        return true;
}
```

CUDA Approach

```
bool mv4_get(dim3 &threads, dim3 &blocks, size_t size)
{
	//how many blocks do we need to process all N vertices?
	size_t num_blocks = size / ((size_t)g_Device.maxThreadsPerBlock);

	if(num_blocks == 0)
	{
		threads.x = size;
		return false;
	}
	else
	{
		//until we have less than one block to process.
		threads.x = (g_Device.maxThreadsPerBlock);
		blocks.x = (int)(min(num_blocks, (size_t)g_Device.maxGridSize[0]));
		return true;
	}
}
```

CUDA Approach

```
if(!mv4_send_data(host_m, dev_m, host_src, dev_src, size))
        return false;

dim3 t, g;          size_t offset = 0;

while(mv4_get(t,  g, size))
{
        // Execute the kernel
        mv4_cuda_kernel<<<g,t>>>(
                (svector4*)dev_dst + offset,
                (const smatrix4*)dev_m,
                (const svector4*)dev_src + offset);
        cudaThreadSynchronize();

        offset += g.x * t.x;size -= g.x * t.x;
}
```

CUDA Approach

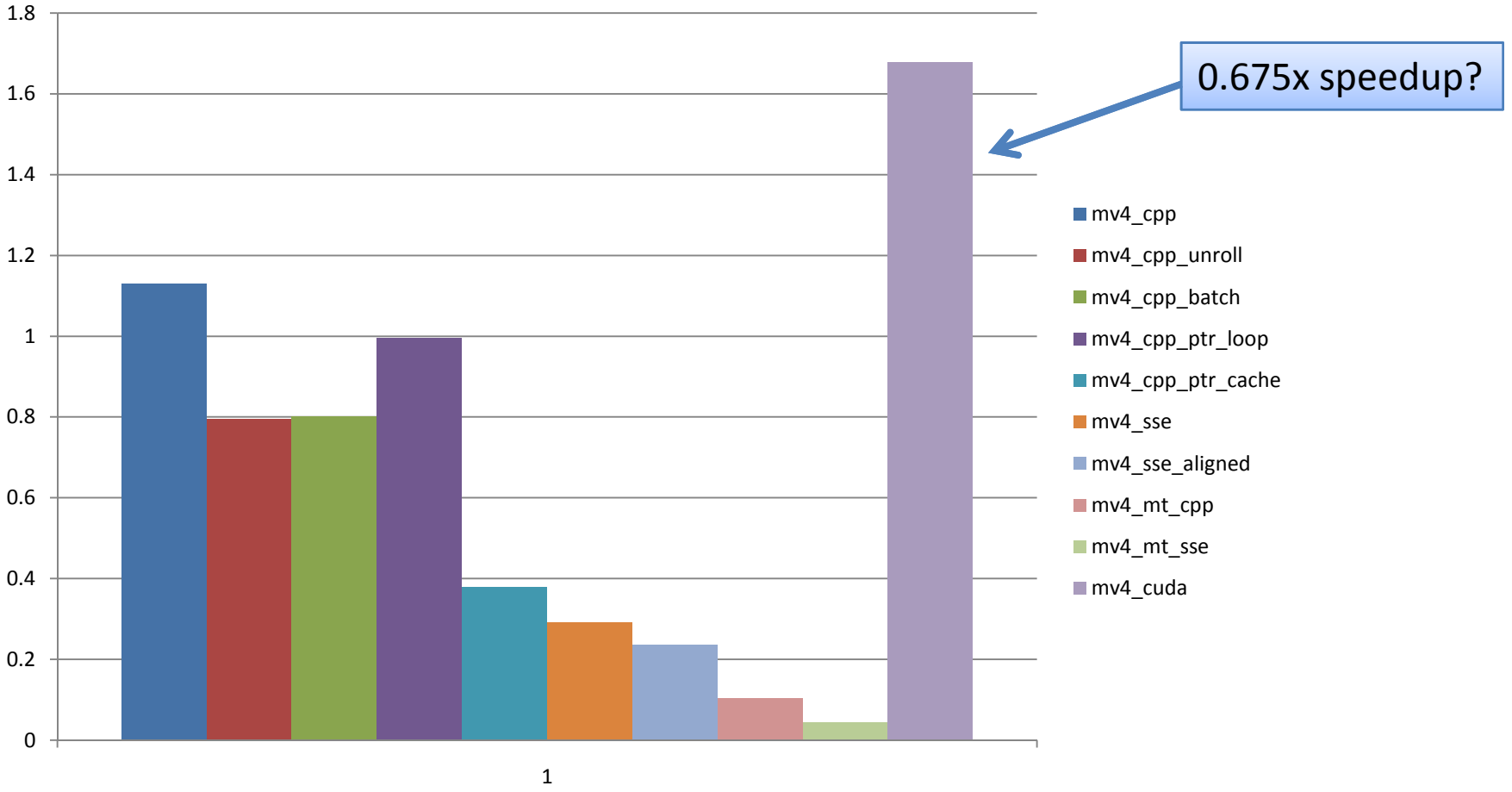
```
//execute the remainder vertices.
if(t.x != 0)
{
        mv4_cuda_kernel<<<g,t>>>(
                (svector4*)dev_dst + offset,
                (const smatrix4*)dev_m,
                (const svector4*)dev_src + offset);
        cudaThreadSynchronize();
}

// Copy from device to host
cudaMemcpy(
        (void*)host_dst, (void*)dev_dst, size * sizeof(vector4),
        cudaMemcpyDeviceToHost);
}
```

CUDA Approach

```
__global__ void mv4_cuda_kernel (
        svector4 *dst, const smatrix4 *a, const svector4 *b)
{
        const size_t index = blockIdx.x * blockDim.x + threadIdx.x;
        const smatrix4 cm = *a;
        const svector4 cb = b[index];
        svector4 cdst;

        cdst.x = cm.m0 *cb.x  + cm.m1 *cb.y  + cm.m2 *cb.z  + cm.m3 *cb.w;
        cdst.y = cm.m4 *cb.x  + cm.m5 *cb.y  + cm.m6 *cb.z  + cm.m7 *cb.w;
        cdst.z = cm.m8 *cb.x  + cm.m9 *cb.y  + cm.m10*cb.z  + cm.m11*cb.w;
        cdst.w = cm.m12*cb.x  + cm.m13*cb.y  + cm.m14*cb.z  + cm.m15*cb.w;

        dst[index] = cdst;
}
```

CUDA Approach

256K Vertices, 100 repetitions



CUDA Approach

- Not enough work is given to kernel.
- Add grain size to kernel
  - Each thread will now process N matrix-vertex multiplications.

CUDA Approach

```
bool mv4_get(dim3 &threads, dim3 &blocks, size_t scale, size_t size)
{
	//how many blocks do we need to process all N vertices?
	size_t num_blocks = size / ((size_t)g_Device.maxThreadsPerBlock * scale);

	if(num_blocks == 0)
	{
		threads.x = size / scale;
		return false;
	}
	else
	{
		//until we have less than one block to process.
		threads.x = (g_Device.maxThreadsPerBlock);
		blocks.x = (int)(min(num_blocks, (size_t)g_Device.maxGridSize[0]));
		return true;
	}
}
```

CUDA Approach

```
while(mv4_get(t, g, 4, size))
{
        // Execute the kernel
        mv4_cuda_kernel_cached_4<<<g,t>>>(
                (svector4*)dev_dst + offset,
                (const smatrix4*)dev_m,
                (const svector4*)dev_src + offset);
        cudaThreadSynchronize();

        offset += g.x * t.x * 4;
        size -= g.x * t.x * 4;
}
```

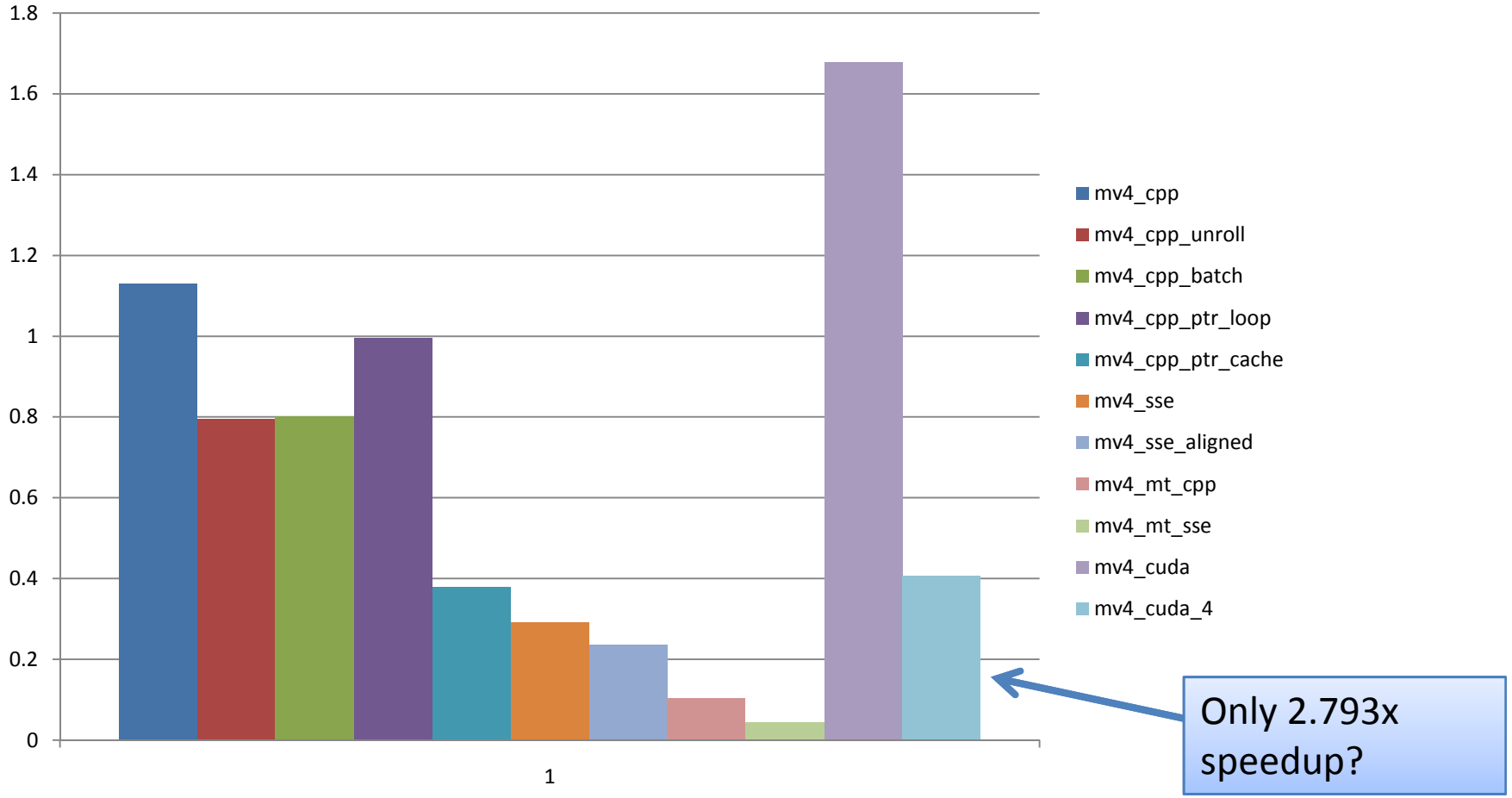
CUDA Approach

```
unsigned int index = blockIdx.x * blockDim.x + threadIdx.x * 4;
const smatrix4 cm = *a;
svector4 cb;
svector4 cdst;

cb = b[index];
cdst.x = cm.m0 *cb.x  + cm.m1 *cb.y  + cm.m2 *cb.z  + cm.m3 *cb.w;
cdst.y = cm.m4 *cb.x  + cm.m5 *cb.y  + cm.m6 *cb.z  + cm.m7 *cb.w;
cdst.z = cm.m8 *cb.x  + cm.m9 *cb.y  + cm.m10*cb.z  + cm.m11*cb.w;
cdst.w = cm.m12*cb.x  + cm.m13*cb.y  + cm.m14*cb.z  + cm.m15*cb.w;
dst[index] = cdst;

++index;
cb = b[index];
cdst.x = cm.m0 *cb.x  + cm.m1 *cb.y  + cm.m2 *cb.z  + cm.m3 *cb.w;
cdst.y = cm.m4 *cb.x  + cm.m5 *cb.y  + cm.m6 *cb.z  + cm.m7 *cb.w;
cdst.z = cm.m8 *cb.x  + cm.m9 *cb.y  + cm.m10*cb.z  + cm.m11*cb.w;
cdst.w = cm.m12*cb.x  + cm.m13*cb.y  + cm.m14*cb.z  + cm.m15*cb.w;
dst[index] = cdst;

++index;
cb = b[index];
cdst.x = cm.m0 *cb.x  + cm.m1 *cb.y  + cm.m2 *cb.z  + cm.m3 *cb.w;
cdst.y = cm.m4 *cb.x  + cm.m5 *cb.y  + cm.m6 *cb.z  + cm.m7 *cb.w;
cdst.z = cm.m8 *cb.x  + cm.m9 *cb.y  + cm.m10*cb.z  + cm.m11*cb.w;
cdst.w = cm.m12*cb.x  + cm.m13*cb.y  + cm.m14*cb.z  + cm.m15*cb.w;
dst[index] = cdst;

++index;
cb = b[index];
cdst.x = cm.m0 *cb.x  + cm.m1 *cb.y  + cm.m2 *cb.z  + cm.m3 *cb.w;
cdst.y = cm.m4 *cb.x  + cm.m5 *cb.y  + cm.m6 *cb.z  + cm.m7 *cb.w;
cdst.z = cm.m8 *cb.x  + cm.m9 *cb.y  + cm.m10*cb.z  + cm.m11*cb.w;
cdst.w = cm.m12*cb.x  + cm.m13*cb.y  + cm.m14*cb.z  + cm.m15*cb.w;
dst[index] = cdst;
```

Same as original kernel,
But unrolled 4 times.

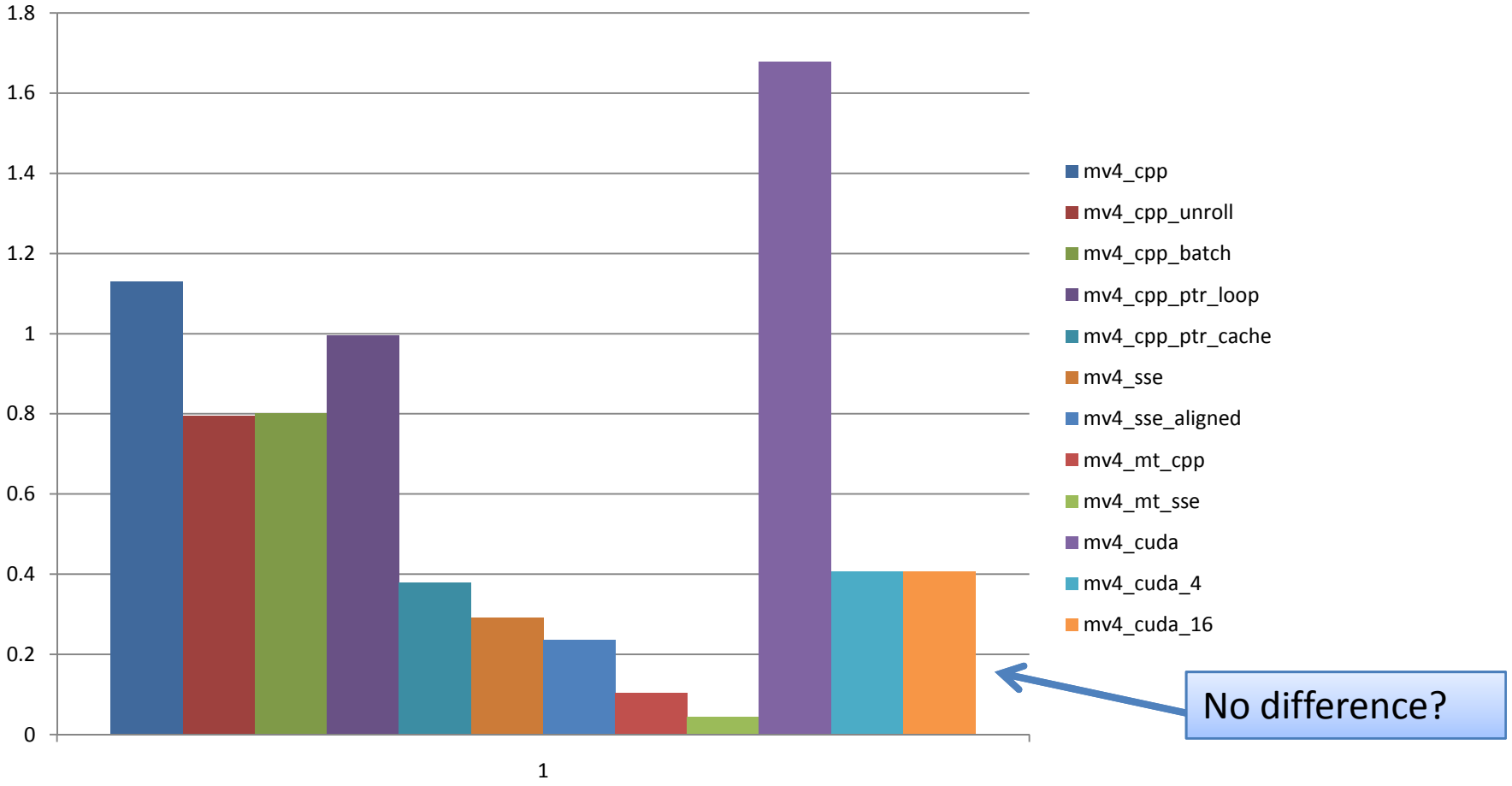
CUDA Approach

256K Vertices, 100 repetitions



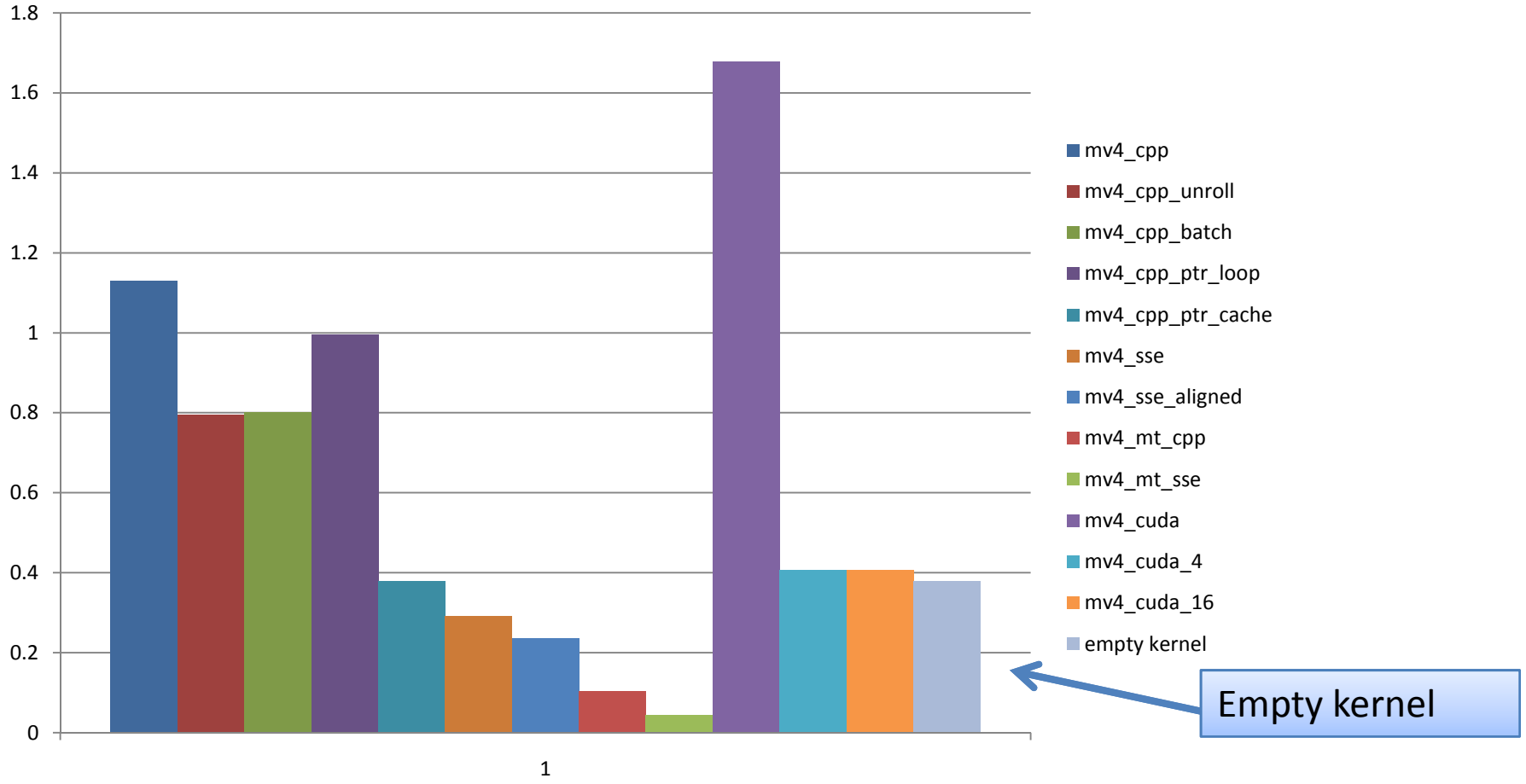
CUDA Approach

256K Vertices, 100 repetitions



CUDA Approach

256K Vertices, 100 repetitions

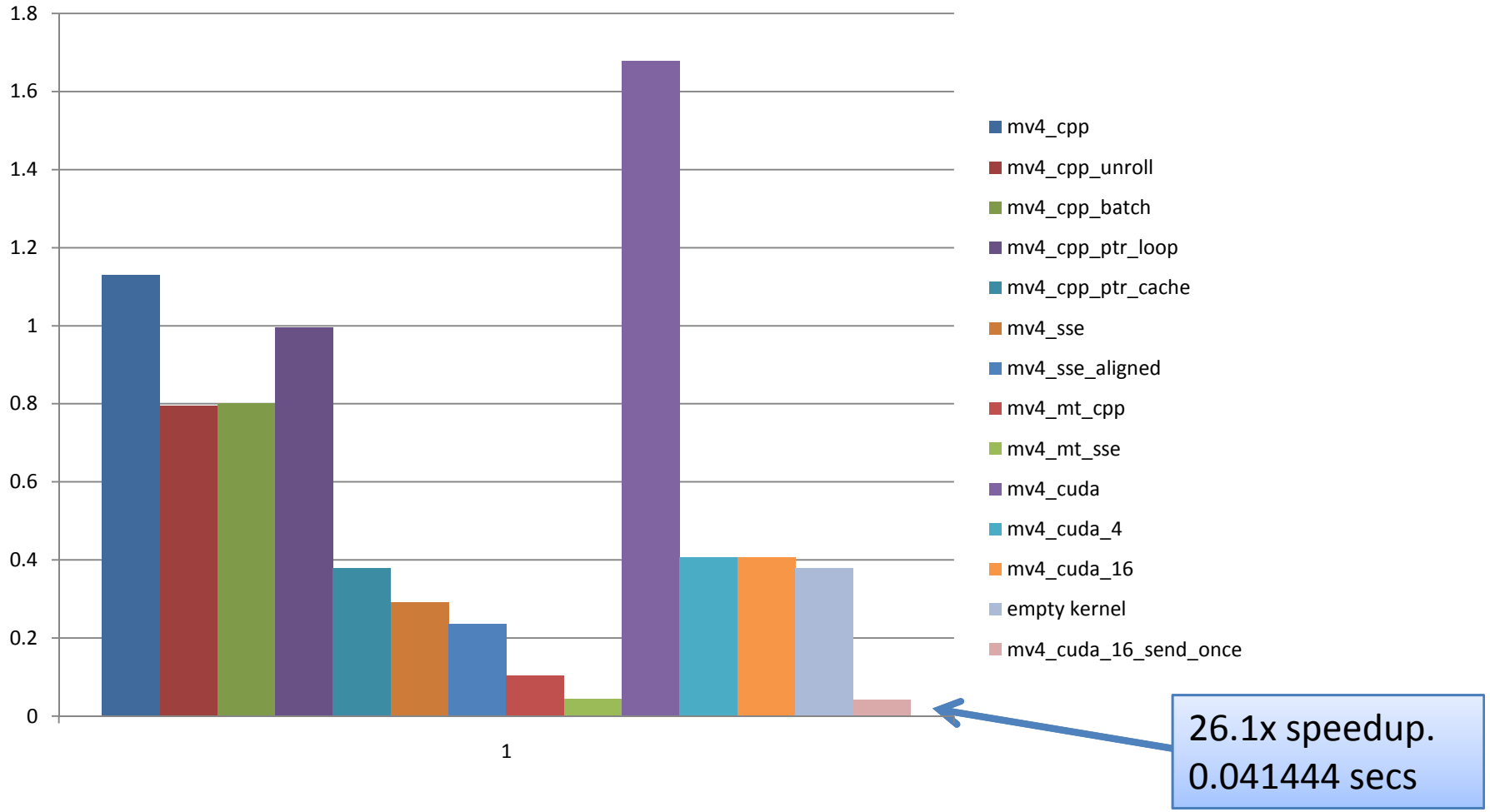


CUDA Approach

- We are memory bound!
- I/O from main memory to GPU has to large a latency.
  - Hides benefits of using GPU.
- If data was never changed, we could send it once up front…

CUDA Approach

256K Vertices, 100 repetitions

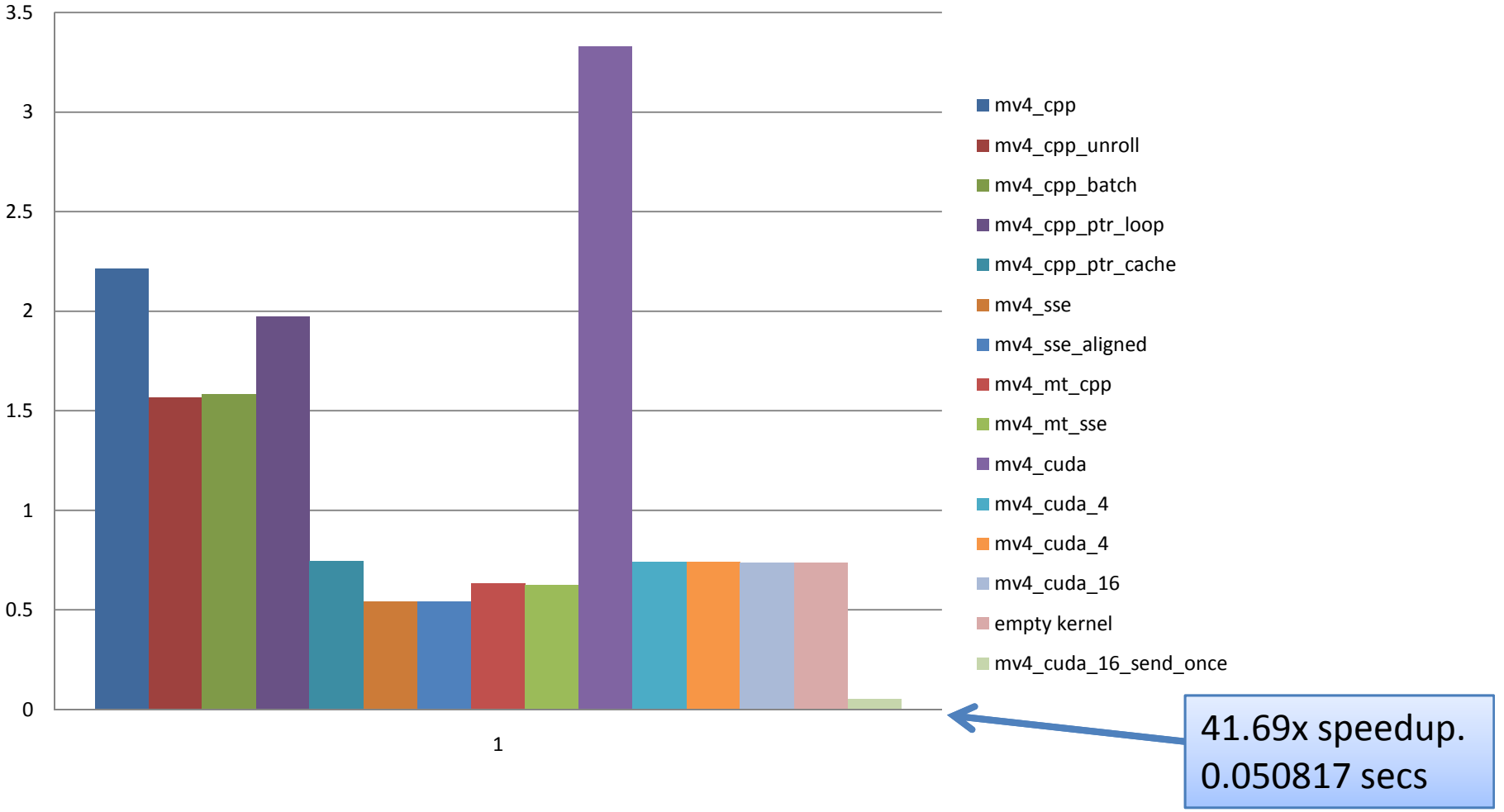


CUDA Approach

- Faster than SSE implementation, but barely.
  - Still memory bound.
- As it turns out, CUDA is not a good candidate for matrix4-vector4 multiplications…
- But what if we have more vertices to transform?
  - Could the memory costs disappear?

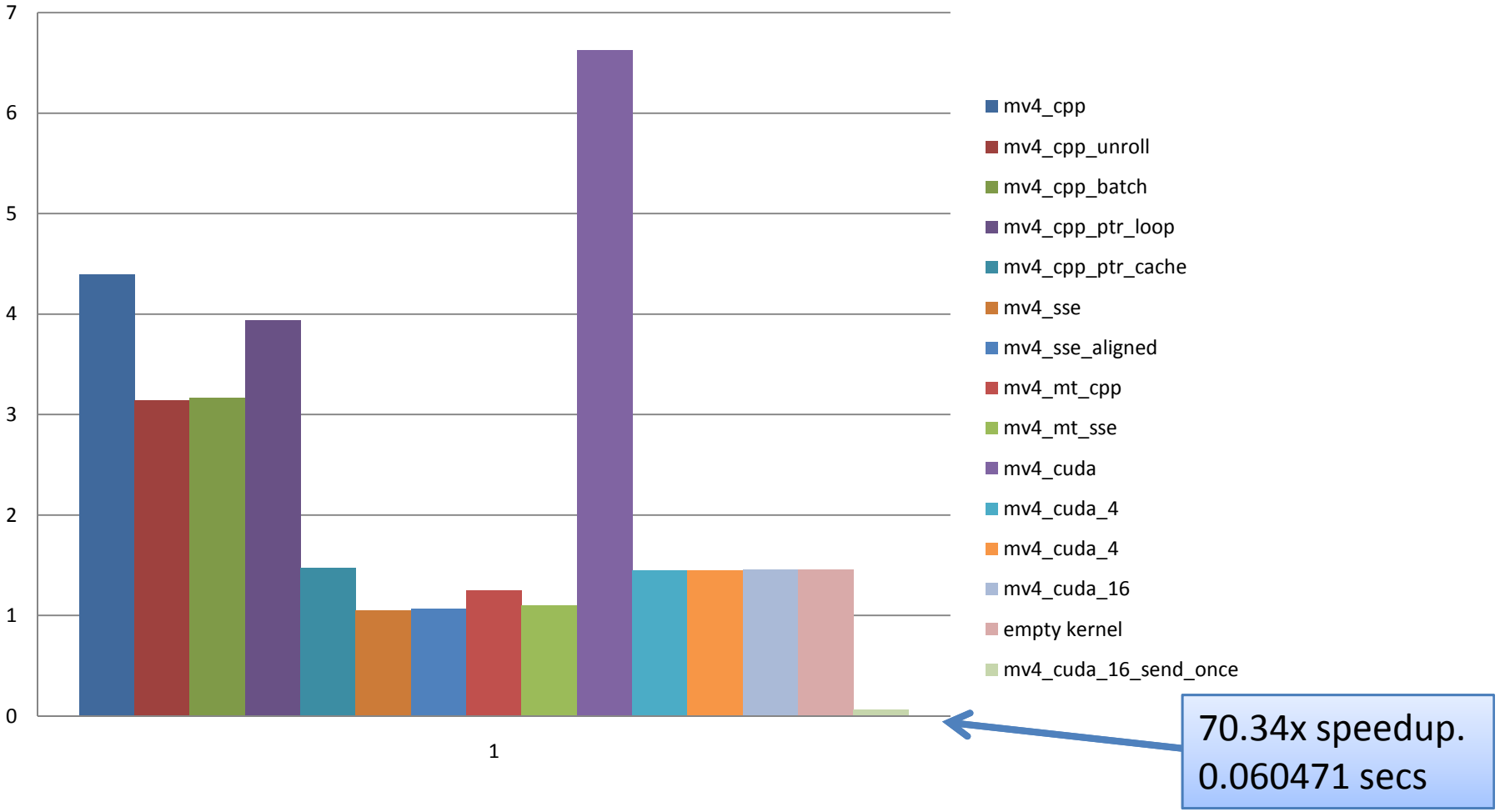
CUDA Approach

512K Vertices, 100 repetitions



CUDA Approach

1M Vertices, 100 repetitions

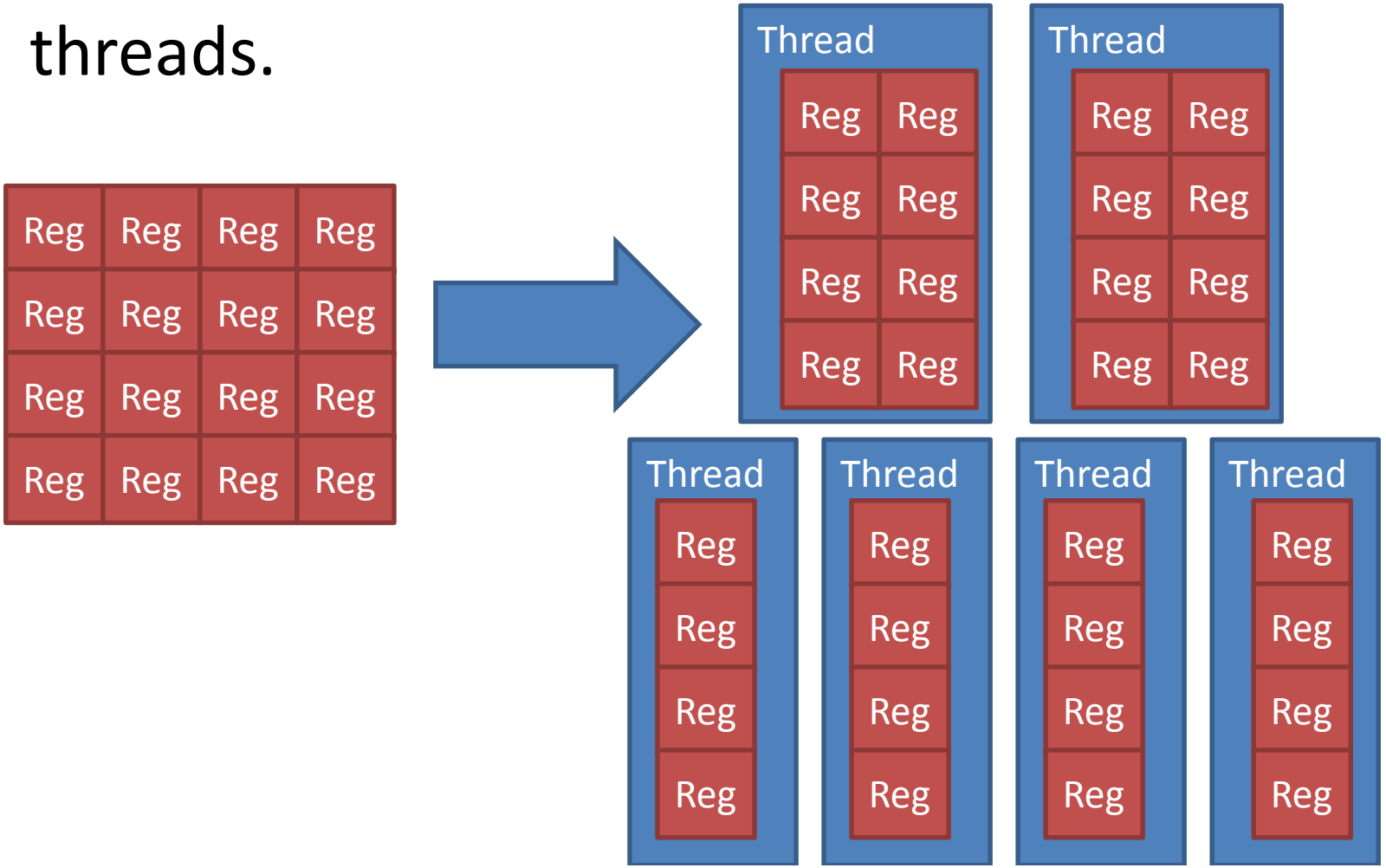


CUDA Limitations

- Memory Wall.
- Register Allocation.
- Memory Coalescing.
- Double Indirections.
- Limited Documentation.

CUDA Limitations

- Fixed # of registers get allocated for each threads.



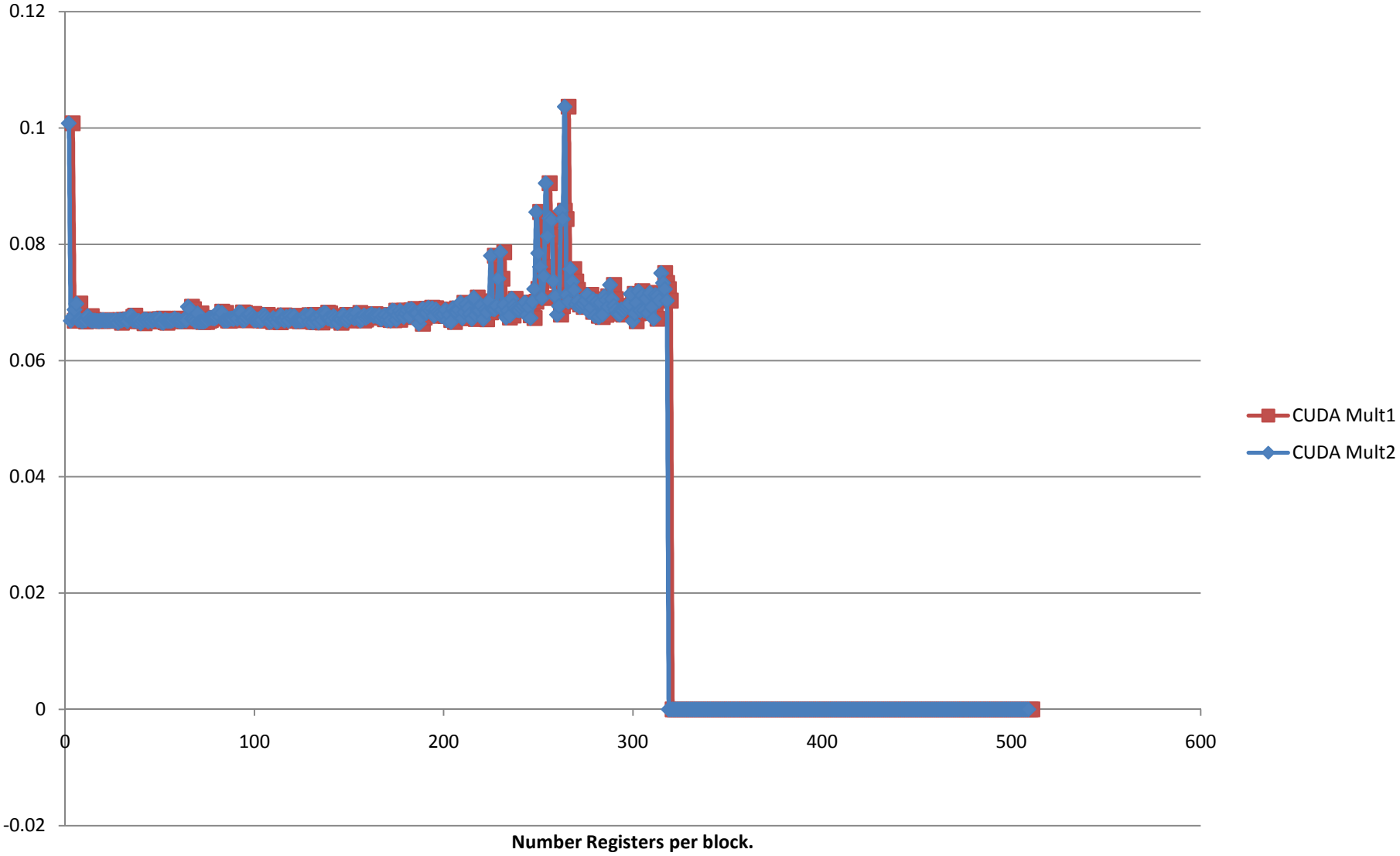
CUDA Limitations

Compiling entry function
'_Z16mv4_mult3_kernelP8svector4PK8smatrix4PKS_j' for 'sm_10‘
Used 24 registers, 16+16 bytes smem, 4 bytes cmem[1]

Compiling entry function
'_Z16mv4_mult2_kernelP8svector4PK8smatrix4PKS_j' for 'sm_10‘
Used 24 registers, 16+16 bytes smem, 4 bytes cmem[1]

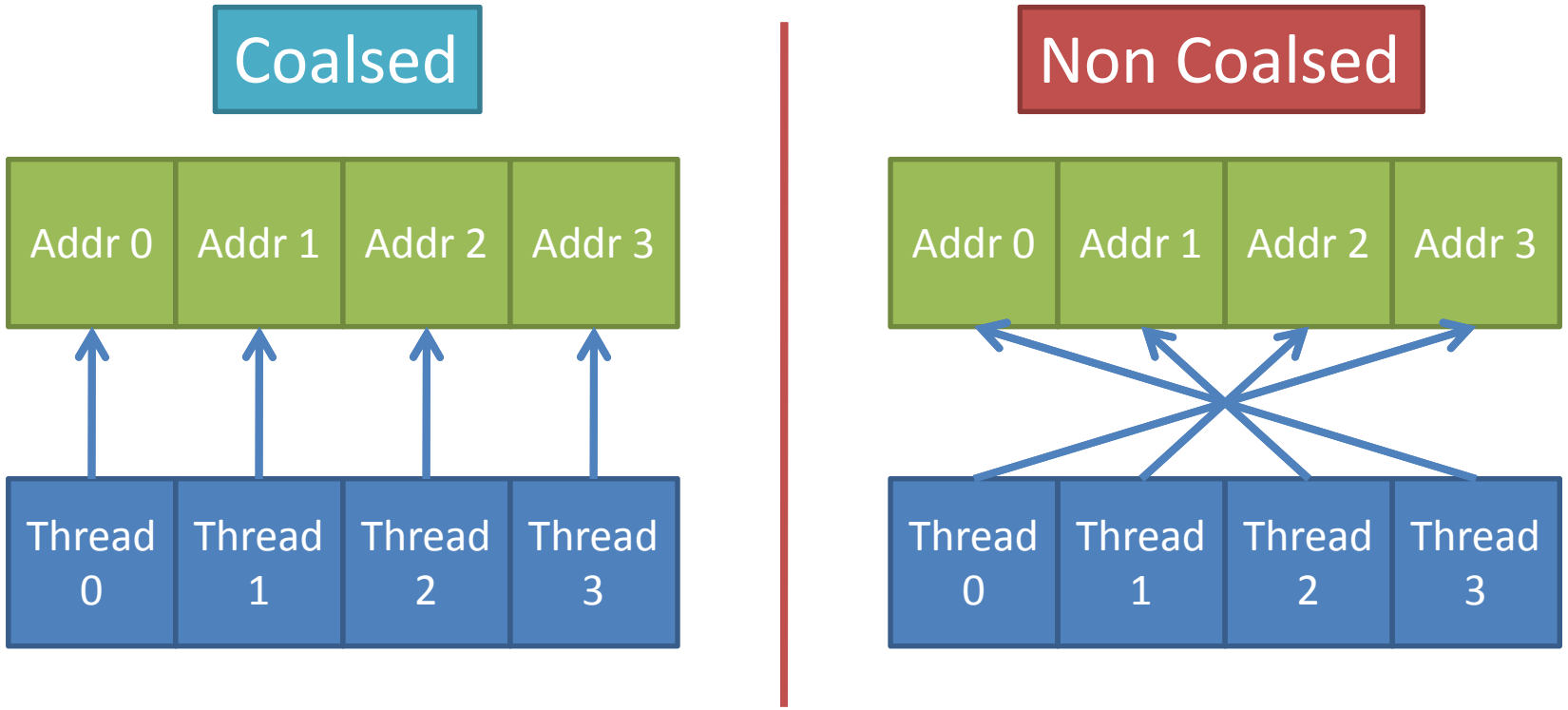
Compiling entry function
'_Z16mv4_mult1_kernelP8svector4PK8smatrix4PKS_j' for 'sm_10‘
Used 30 registers, 16+16 bytes smem, 4 bytes cmem[1]

CUDA Limitations



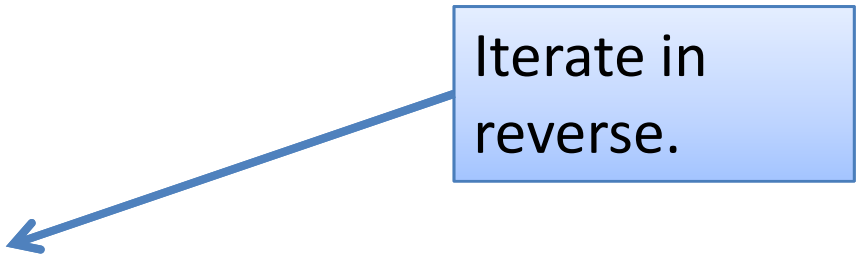
CUDA Limitations

- Hardware serializes non-coalesced memory reads.

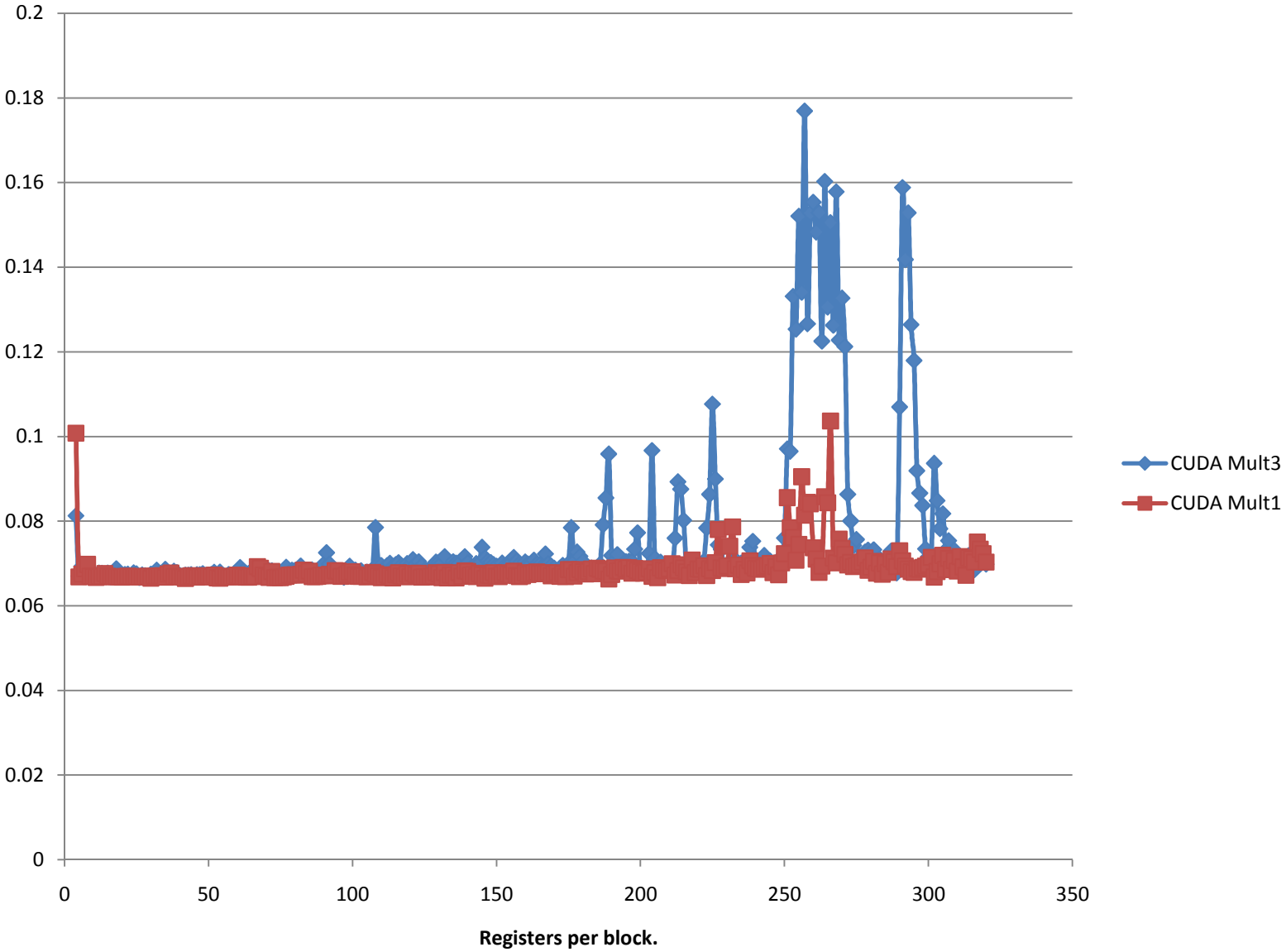


CUDA Limitations

```
unsigned int index = blockIdx.x * blockDim.x + threadIdx.x * 4;

svector4 cb;
svector4 cdst;
const smatrix4 cm = *a;

index += 3;

cb = b[size - index];
mv4_mat_vec_mult(cdst, cm, cb);
dst[size - index] = cdst;
--index;
```

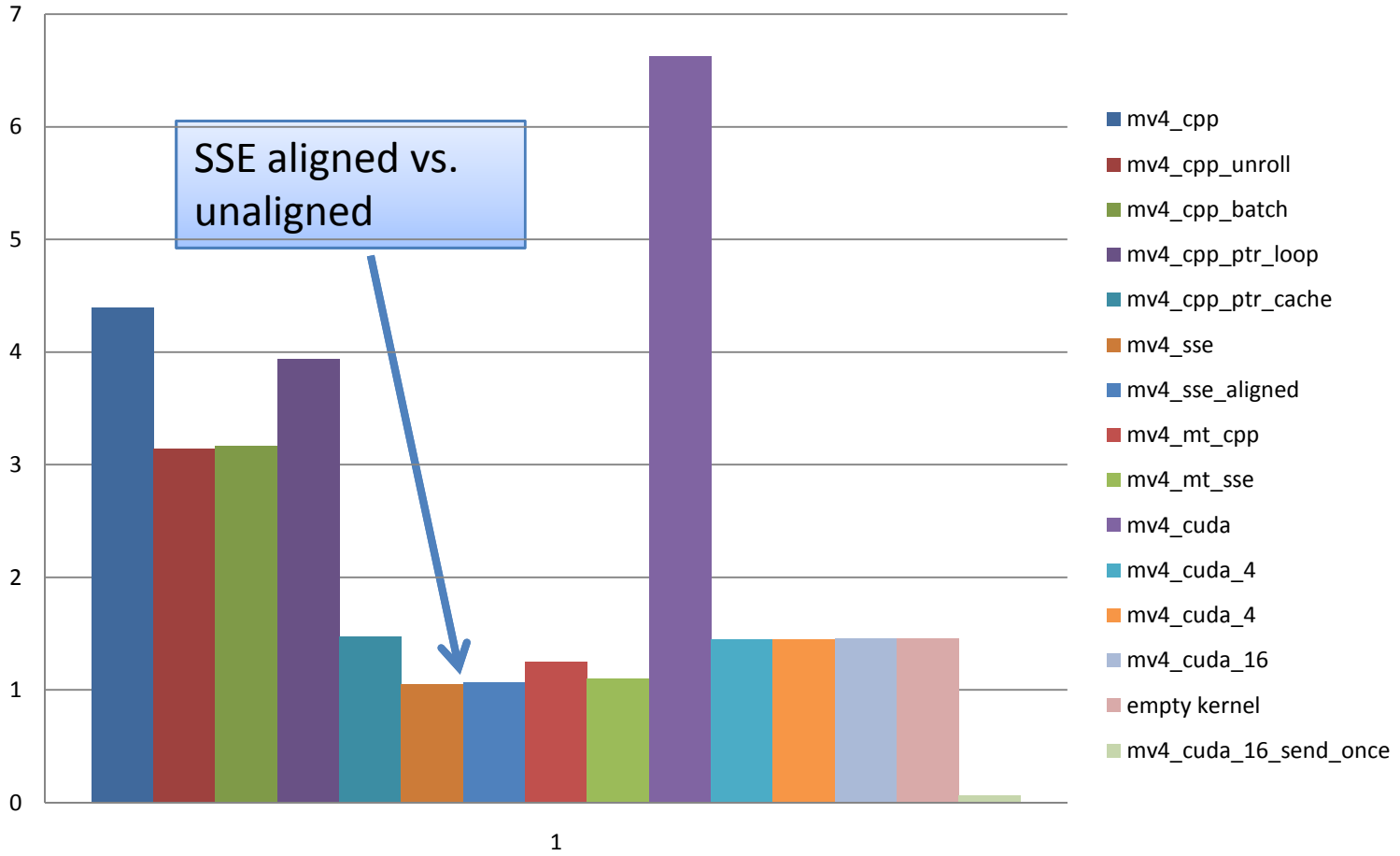


CUDA Limitations



Other Notes

1M Vertices, 100 repetitions



Other Notes

1M Vertices, 100 repetitions