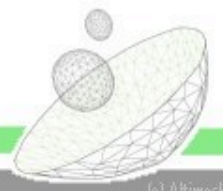
Image Processing Optimization
C# on GPU with Hybridizer™

regis.portalez@altimesh.com

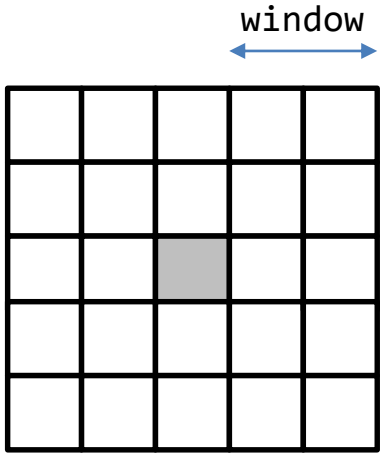


Median Filter
Denoising

Noisy image
(lena 1960x1960)

Denoised image
window = 3

Median Filter
Denoising



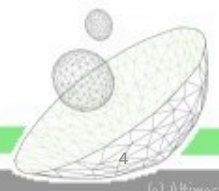
$$\text{Output}[i,j] = Median\{input[p,q],\ p \in [i-window, i+window],\ q \in [j-window, j+window]\}$$

For each pixel, we read (2 * window + 1)² pixels of input

Optimization Steps
An Overview

1. Enable C# parallelization (remove loop side effects) 2. Use Parallel.For	Necessary	x5
3. Run on GPU (Hybridizer) – 3.1 Decorate methods – 3.2 Allocate memory – 3.3 Feed our 50k threads	Low cost	x78
4. Implement Advanced Optimizations – 4.1 Shared memory – 4.2 Texture memory	Bonus	x92
5. More Optimizations	Expertise	x?

Median Filter is not easy.
On easier code, steps 3 and 4 would be sufficient



AForge code

```
ushort* src, dst;
for (int y = startY; y < stopY; y++)
{
    for (int x = startX; x < stopX; x++, src++, dst++)
    {
        int c = 0;
        for (i = -radius; i <= radius; i++)
        {
            for (j = -radius; j <= radius; j++)
            {
                g[c++] = src[i * srcStride + j];
            }
        }

        Array.Sort(g, 0, c);
        *dst = g[c >> 1];
    }
    src += srcOffset;
    dst += dstOffset;
}
```

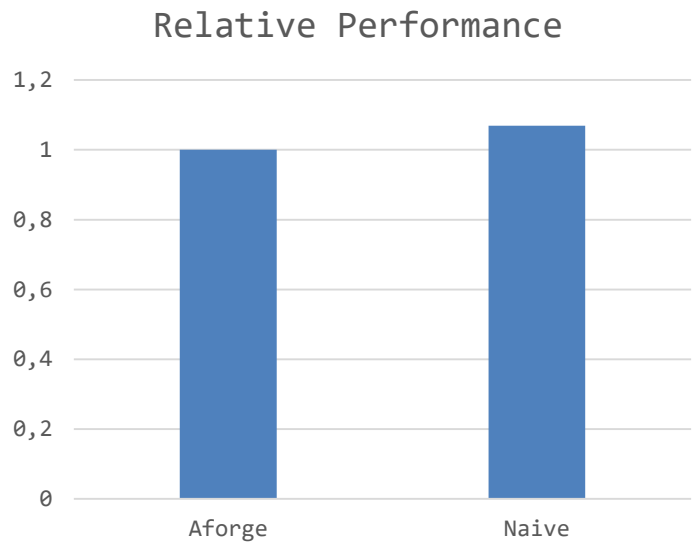
AForge code

```
ushort* src, dst;
for (int y = startY; y < stopY; y++)
{
    for (int x = startX; x < stopX; x++, src++, dst++)
    {
        int c = 0;
        for (i = -radius; i <= radius; i++)
        {
            for (j = -radius; j <= radius; j++)
            {
                g[c++] = src[i * srcStride + j];
            }
        }

        Array.Sort(g, 0, c);
        *dst = g[c >> 1];
    }
    src += srcOffset;
    dst += dstOffset;
}
```

Old-school optimizations
Inner loops have side-effects
Requires unsafe

1. Enable Parallelization
Remove loop side-effects

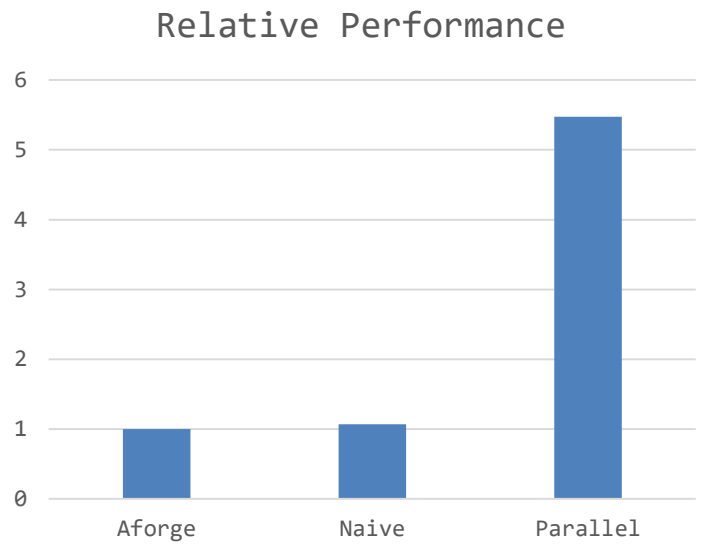
```
var buffer1 = new ushort[windowCount * windowCount];
for (int j = window; j < height - window; ++j)
{
    for (int i = window; i < width - window; ++i)
    {
        for (int k = -window; k <= window; ++k)
        {
            for (int p = -window; p <= window; ++p)
            {
                int bufferIndex = (k + window) * windowCount + p + window;
                int pixelIndex = (j + k) * width + (i + p);
                buffer1[bufferIndex] = input[pixelIndex];
            }
        }

        Array.Sort(buffer1, 0, windowCount * windowCount);
        output[j * width + i] = buffer1[(windowCount * windowCount) / 2];
    }
}
```



No performance penalty – Jitter is quite smart now!
Much more readable code
Inner loops are independant of the outer loops: possible to introduce parallelization

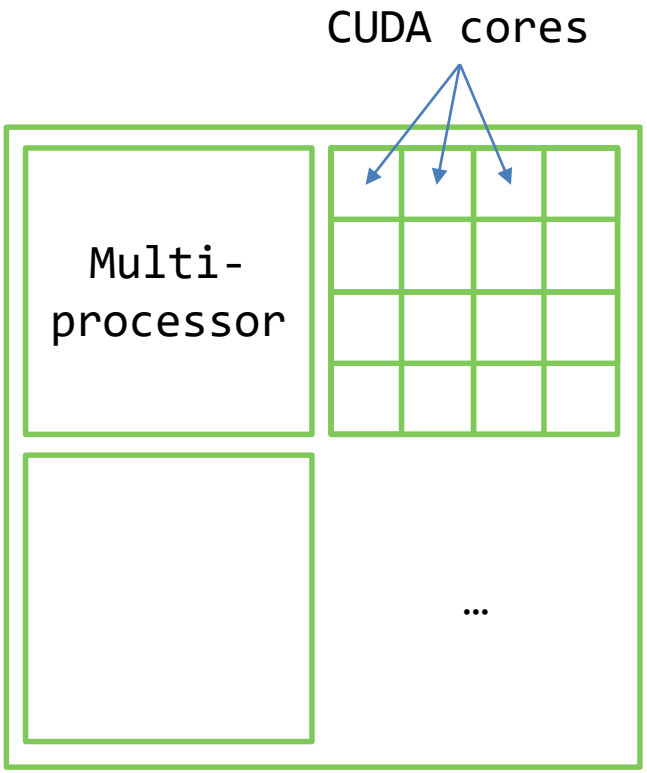
2. Use Parallel.For

```
Parallel.For(window, height - window, j =>
{
    var buffer1 = new ushort[windowCount * windowCount];
    for (int i = window; i < width - window; ++i)
    {
        for (int k = -window; k <= window; ++k)
        {
            for (int p = -window; p <= window; ++p)
            {
                int bufferIndex = (k + window) * windowCount + p + window;
                int pixelIndex = (j + k) * width + (i + p);
                buffer1[bufferIndex] = input[pixelIndex];
            }
        }

        Array.Sort(buffer1, 0, windowCount * windowCount);
        output[j * width + i] = buffer1[(windowCount * windowCount) / 2];
    }
});
```



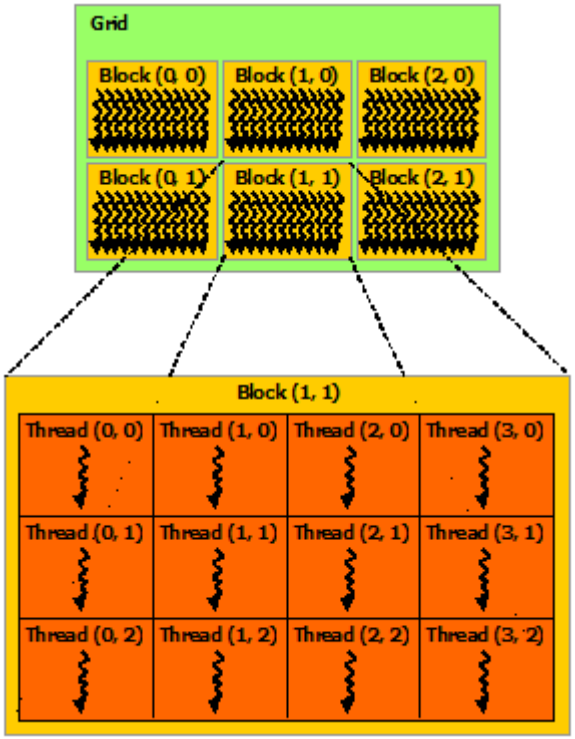
One line change yields a x5,5 speed-up

3. Run On GPU
CUDA & GPU: A Few Words



- Multiprocessors (SM) are similar to CPU Cores
- CUDA cores are similar to CPU SIMD lanes

3. Run On GPU
CUDA threading model

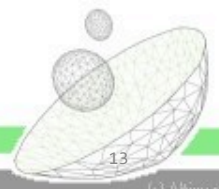


- Threads are grouped in blocks
- Blocks are grouped in a grid
- Grids and blocks have configurable shape (1, 2 or 3D)
- 1 block run on a single SM

3. Run on GPU
Hybridizer™: A Few Words

- Hybridizer™ is a compiler targeting CUDA-enabled GPUS from DotNet.
- Attribute-based (no runtime cost)
- Integrated with debugger and profiler
- Support of Generics and Virtual functions
- …

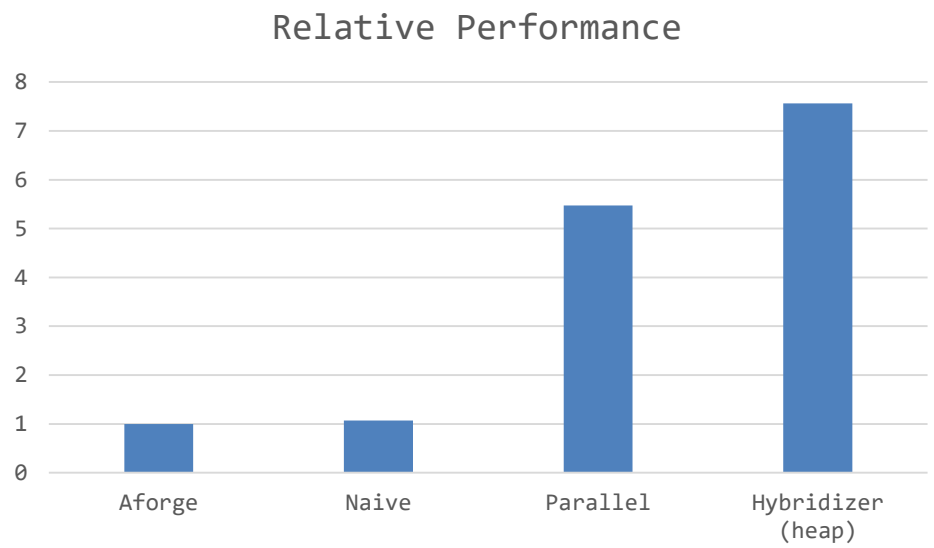
- Trial version downloadable from Visual Studio Marketplace
- Professional edition available in beta (Altimesh website)
- Full version already deployed in Investment Banks (upon request)



3.1 Run On GPU
Decorate Methods

One and only modification

```
[EntryPoint]
public static void ParallelCsharp(byte[] output, byte[] input, int width, int height)
{
    Parallel.For(window, height - window, j =>
    {
        var buffer1 = new byte[windowCount * windowCount];
        for (int i = window; i < width - window; ++i)
        {
            for (int k = -window; k <= window; ++k)
            {
                for (int p = -window; p <= window; ++p)
                {
                    int bufferIndex = (k + window) * windowCount + p + window;
                    int pixelIndex = (j + k) * width + (i + p);
                    buffer1[bufferIndex] = input[pixelIndex];
                }
            }

            Array.Sort(buffer1, windowCount * windowCount);
            output[j * width + i] = buffer1[(windowCount * windowCount) / 2];
        }
    });
}
```



Quite disappointing isn’t it? WHY??

3.2 Allocate Memory
Heap Allocation On GPU

Thread-local malloc is really slow on GPU

```
[EntryPoint]
public static void ParallelCsharp(ushort[] output, ushort[] input, int width, int height)
{
    Parallel.For(window, height - window, j =>
    {
        var buffer1 = new ushort[windowCount * windowCount];
        for (int i = window; i < width - window; ++i)
        {
            for (int k = -window; k <= window; ++k)
            {
                for (int p = -window; p <= window; ++p)
                {
                    int bufferIndex = (k + window) * windowCount + p + window;
                    int pixelIndex = (j + k) * width + (i + p);
                    buffer1[bufferIndex] = input[pixelIndex];
                }
            }

            Array.Sort(buffer1, windowCount * windowCount);
            output[j * width + i] = buffer1[(windowCount * windowCount) / 2];
        }
    });
}
```

3.2 Allocate Memory
Move To Stack

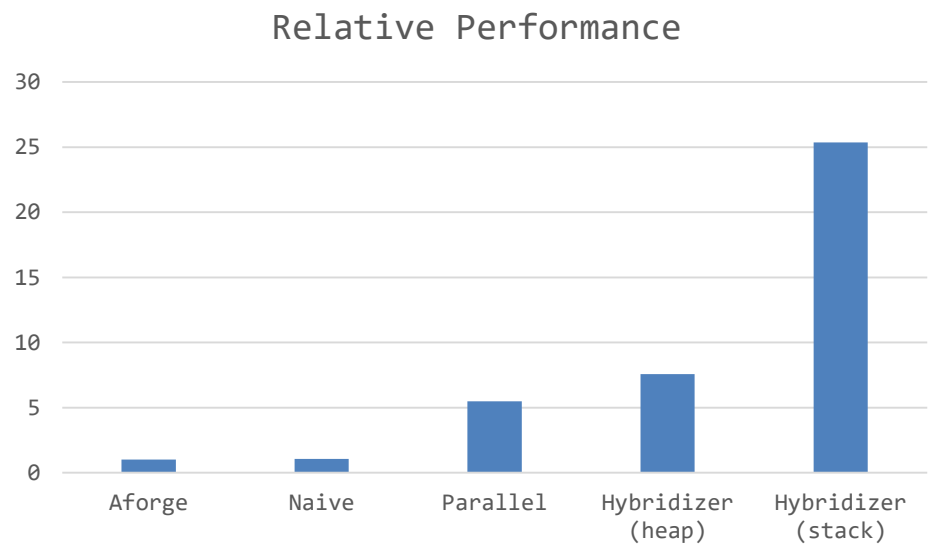
```
[EntryPoint]
public static void ParallelCsharp(ushort[] output, ushort[] input, int width, int height)
{
    Parallel.For(window, height - window, j =>
    {
        var buffer1 = new StackArray<ushort>(windowCount * windowCount);
        for (int i = window; i < width - window; ++i)
        {
            …
        }
    });
}
```

Mapped to:

unsigned short buffer1[size];

Allocated on stack :

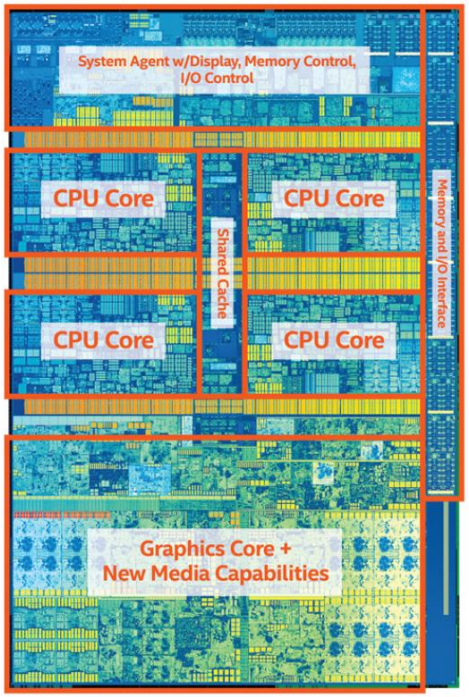
benefits from cache / registries if it fits



3.3 Feed the Beast

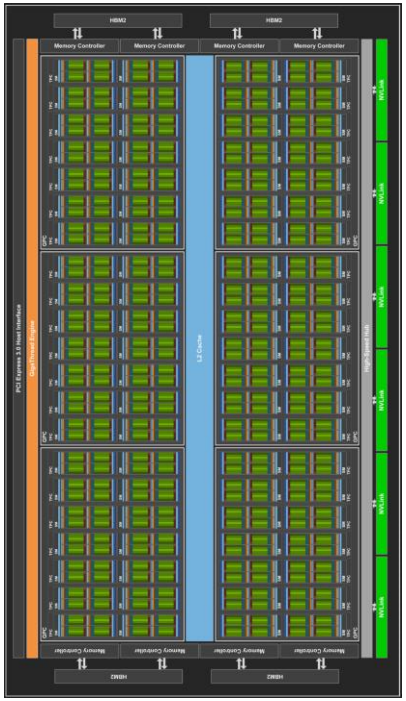
CPU

- Cores
  - Consumer : 8
  - Server : 22
- SIMD Lanes
  - AVX2 : 4 - 8
  - AVX512 : 8 – 16
- Hyperthreading
  - x2
- Parallelism



GPU

- SMs
  - GeForce : 28
  - Tesla : 80
- Cores per SM
  - GeForce : 128
  - Tesla : 64
- Context (to hide latency)
  - 32
- Parallelism

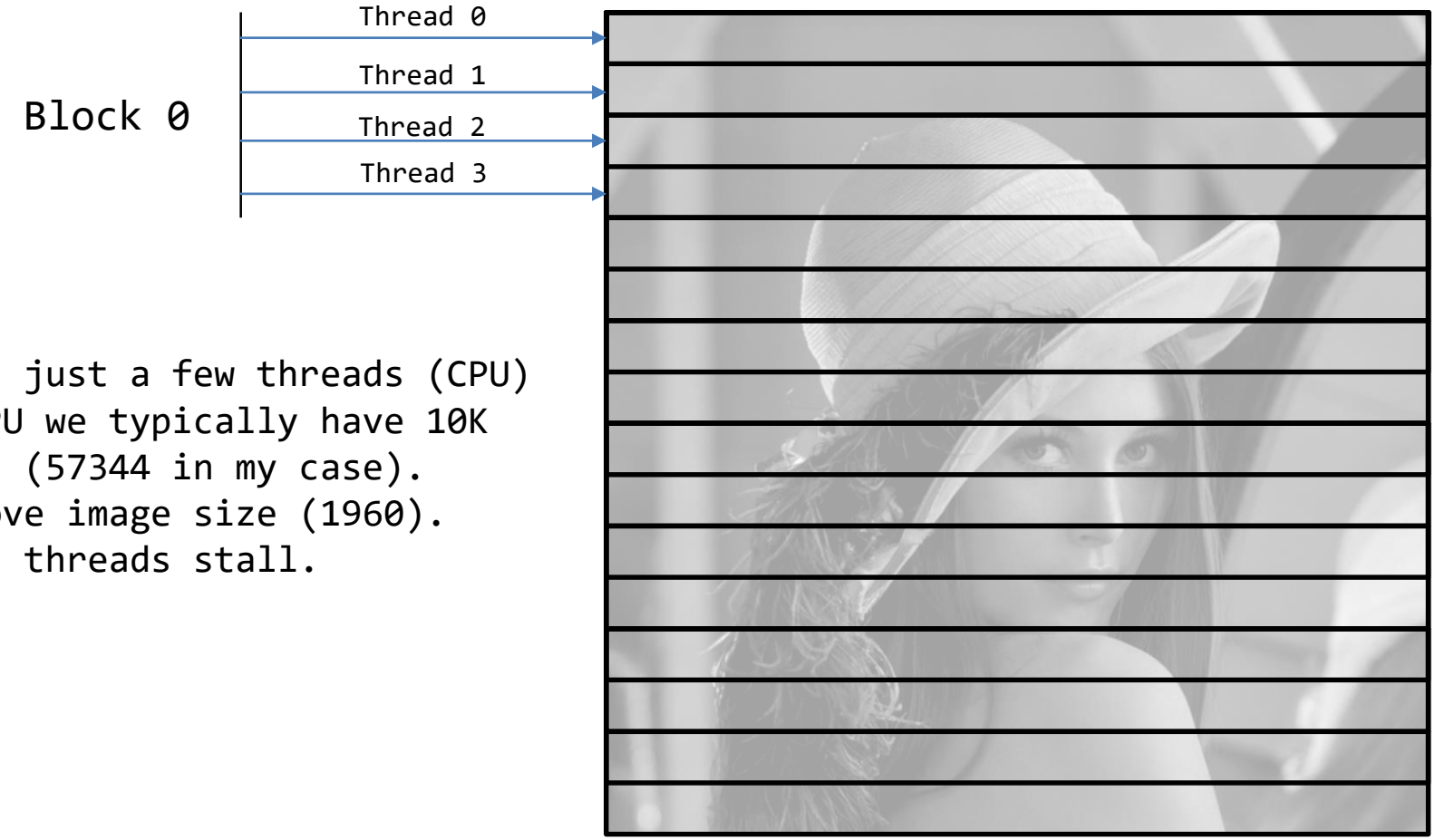


32 up to **704**

3,584 up to **164,000**

3.3 Feed the Beast

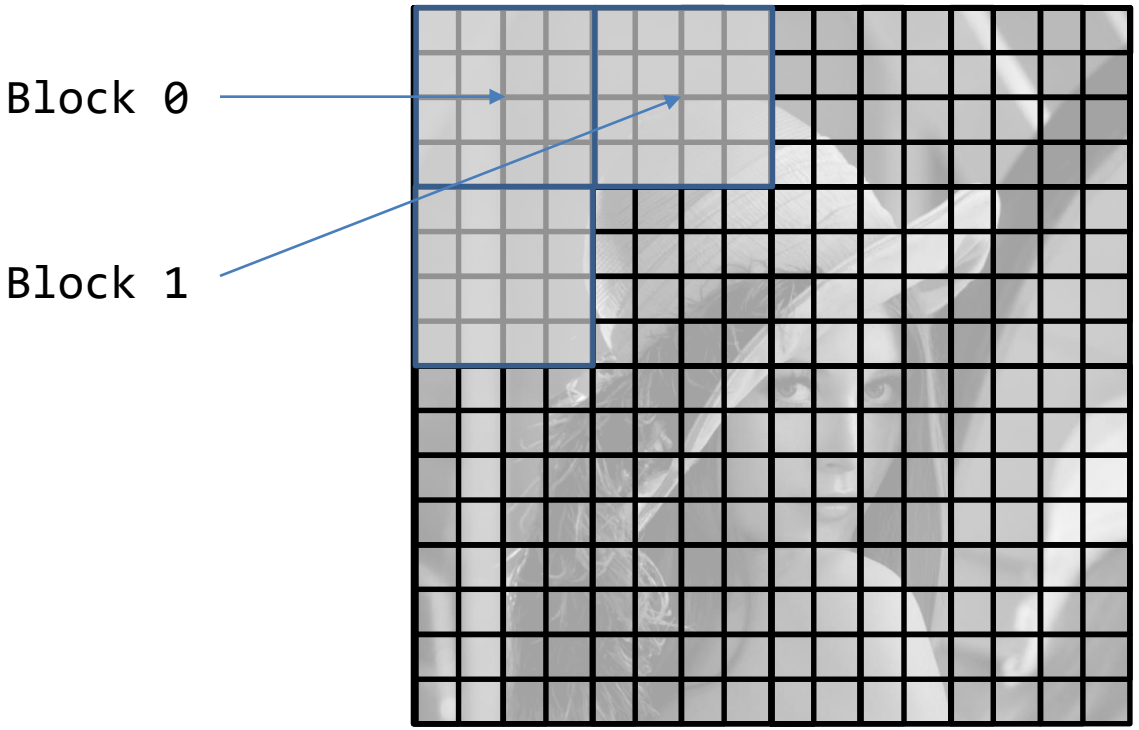
Not Enough Lines – Too Many Threads



Ok with just a few threads (CPU)
On a GPU we typically have 10K threads (57344 in my case).
Far above image size (1960).
=> Most threads stall.

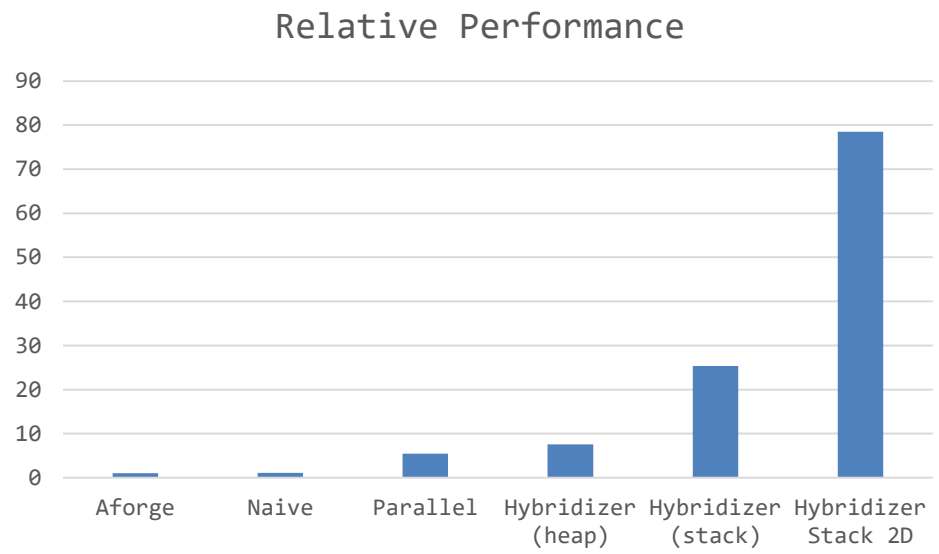
3.3 Feed the Beast
Use A 2D Grid

```
[EntryPoint]
public static void Parallel2DStack(ushort[] output, ushort[] input, int width, int height)
{
    Parallel2D.For(window, width - window, window, height - window, (i, j) =>
    {
        …
    });
}
```



Don’t slice the image, dice it !

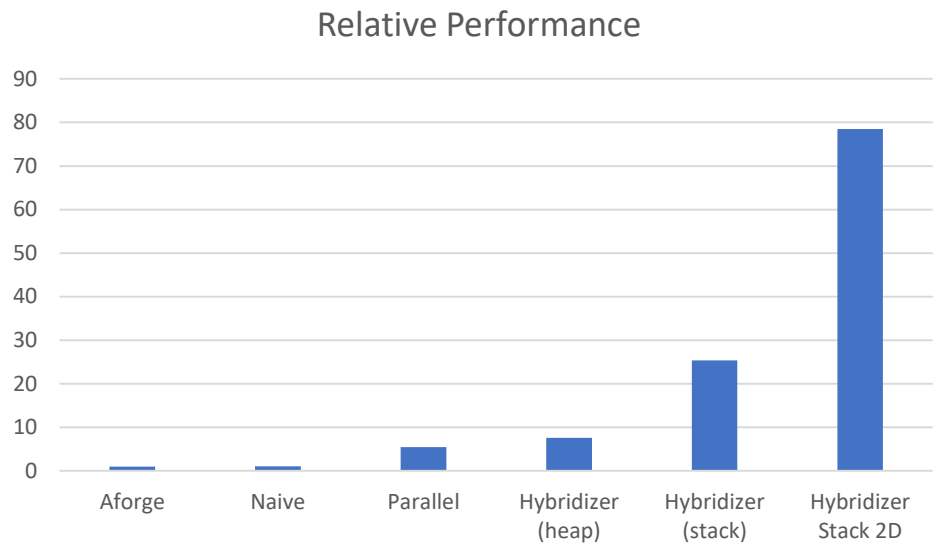
We have 4M pixels : enough to feed the GPU



Run time (seconds):

- AForge : 4,16
- Parallel C# : 0,76
- Hybridizer Stack 2D : 0,053

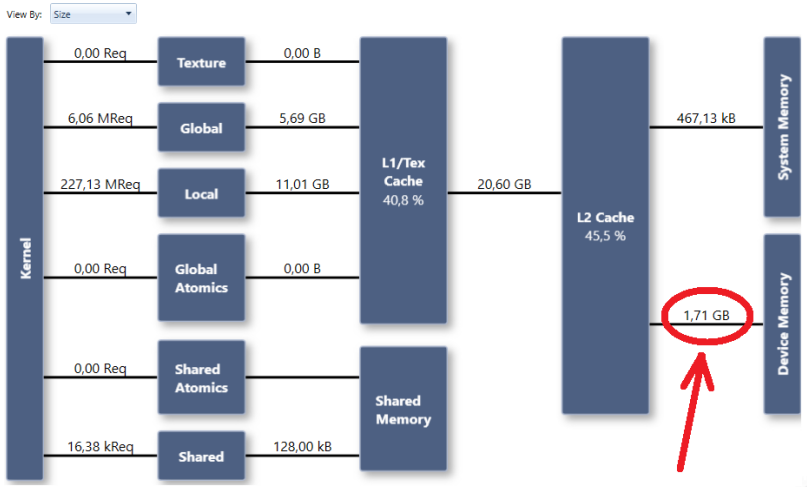
Can we do better?



Run time (seconds):

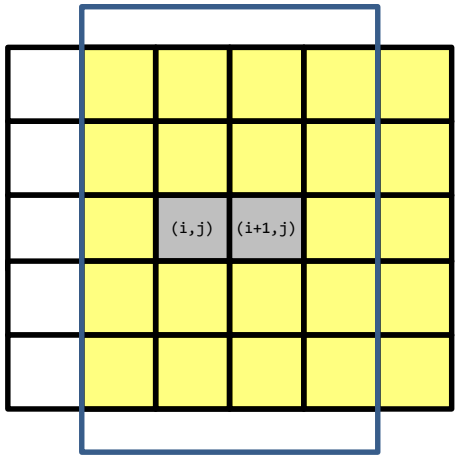
- AForge : 4,16
- Parallel C# : 0,76
- Hybridizer Stack 2D : 0,053

Can we do better?



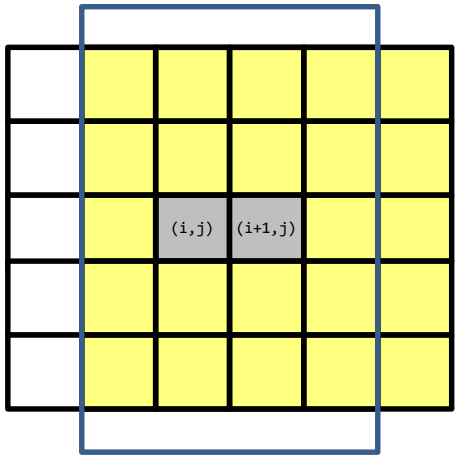
Seems we are reading too much data!

4.1 Implement Advanced Optimizations
Leverage On-Chip Cache (Shared memory)

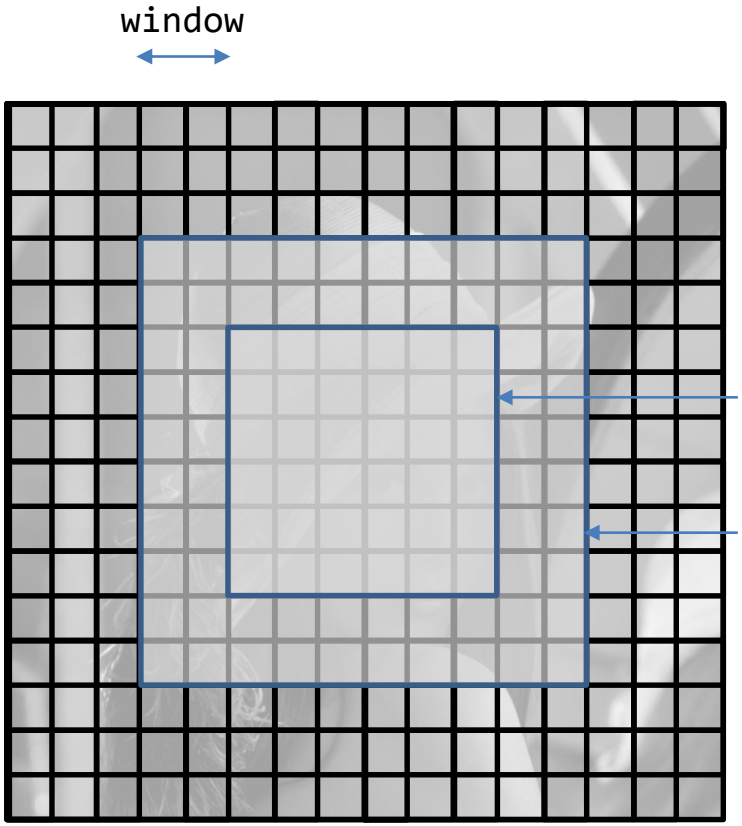


Common read zone

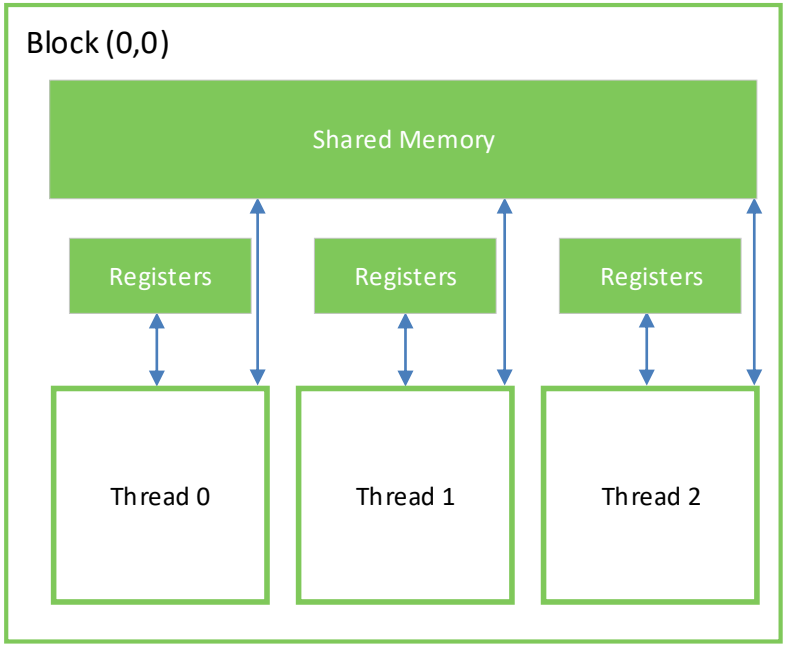
4.1 Implement Advanced Optimizations
Leverage On-Chip Cache (Shared memory)



Common read zone



4.1 Implement Advanced Optimizations
Leverage On-Chip Cache (Shared memory)



- On chip (Multiprocessor)
- Accessible by entire block
- 48KB per block

- See it as CPU L1-cache with explicit control

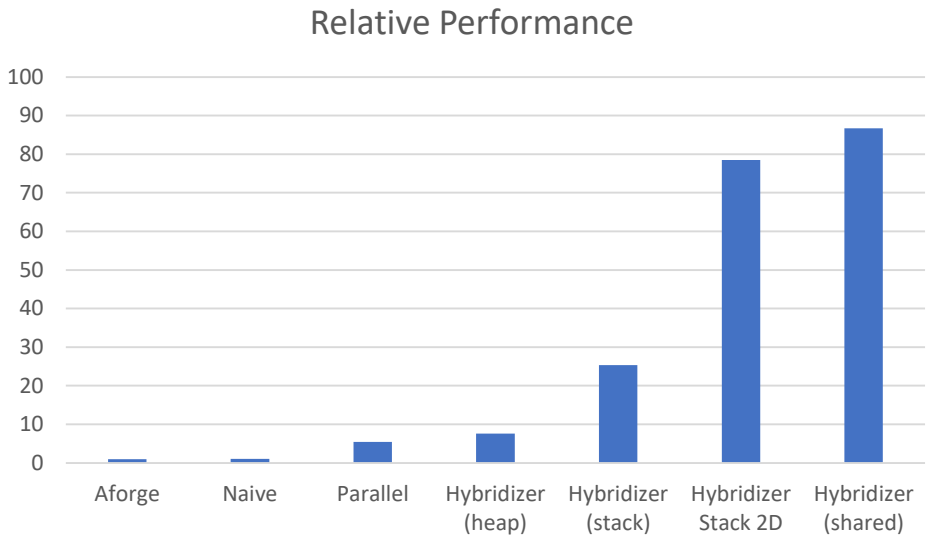
4.1 Implement Advanced Optimizations
Leverage On-Chip Cache (Shared memory)

```
[EntryPoint]
public static void Parallel2DShared(ushort[] output, ushort[] input, int width, int height)
{
    int cacheWidth = blockDim.x + 2 * window;
    ushort[] cache = new SharedMemoryAllocator<ushort>().allocate(cacheWidth* cacheWidth);
    for (int bid_j = blockIdx.y; bid_j < (height ) / blockDim.y; bid_j += gridDim.y)
    {
        for (int bid_i = blockIdx.x; bid_i < (width) / blockDim.x; bid_i += gridDim.x)
        {
            int bli = bid_i * blockDim.x;
            int blj = bid_j * blockDim.y;
            int i = threadIdx.x + bid_i * blockDim.x;
            int j = threadIdx.y + bid_j * blockDim.y;

            // … some code to fetch cache – put data in shared memory
            CUDAIntrinsics.__syncthreads();

            var buffer1 = new StackArray<ushort>(windowCount * windowCount);
            var buffer2 = new StackArray<ushort>(windowCount * windowCount);
            for (q = -window; q <= window; ++q)
            {
                for (p = -window; p <= window; ++p)
                {
                    int bufferIndex = (q + window) * windowCount + p + window;
                    int cacheIndex = (threadIdx.y + window + q) * cacheWidth + threadIdx.x + window + p;
                    buffer1[bufferIndex] = cache[cacheIndex];
                }
            }

            MergeSort(buffer1, buffer2, windowCount * windowCount);
            output[j * width + i] = buffer1[(windowCount * windowCount) / 2];
        }
    }
}
```

Cache « allocation »

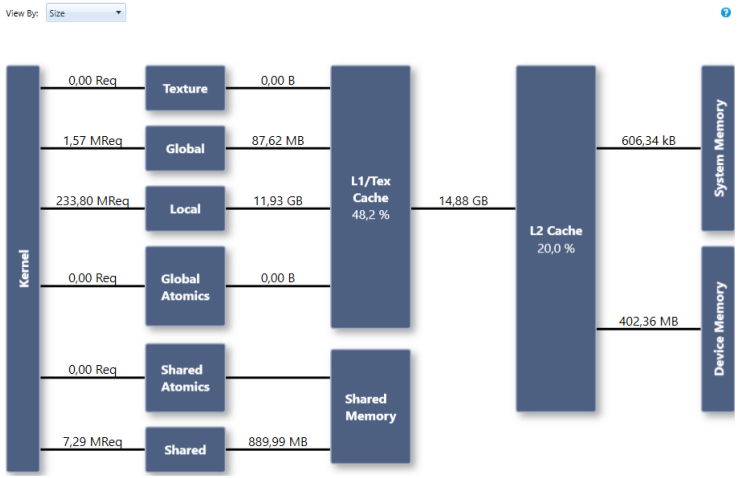
Synchronize threads in block

Read from cache



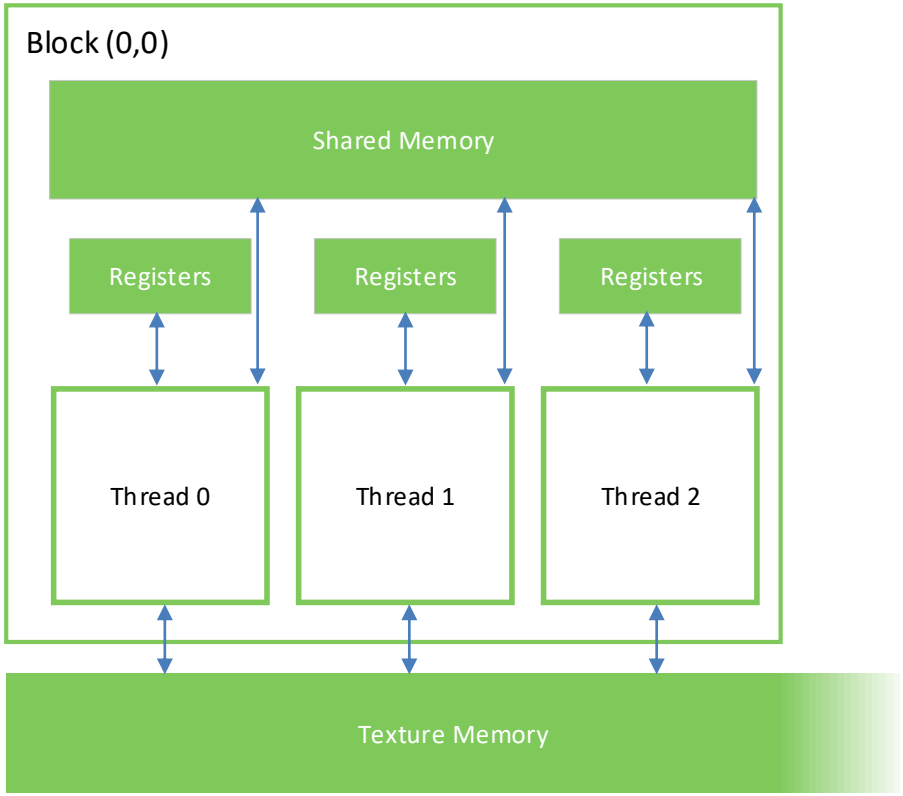
We have a x87 speed-up over initial single-threaded code.
Code still works in .Net

Can we do better?

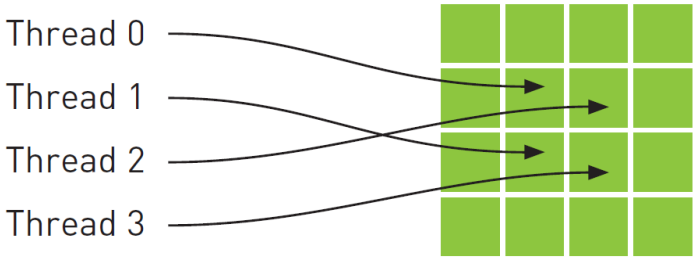


From 1.7 GB down to 402 MB

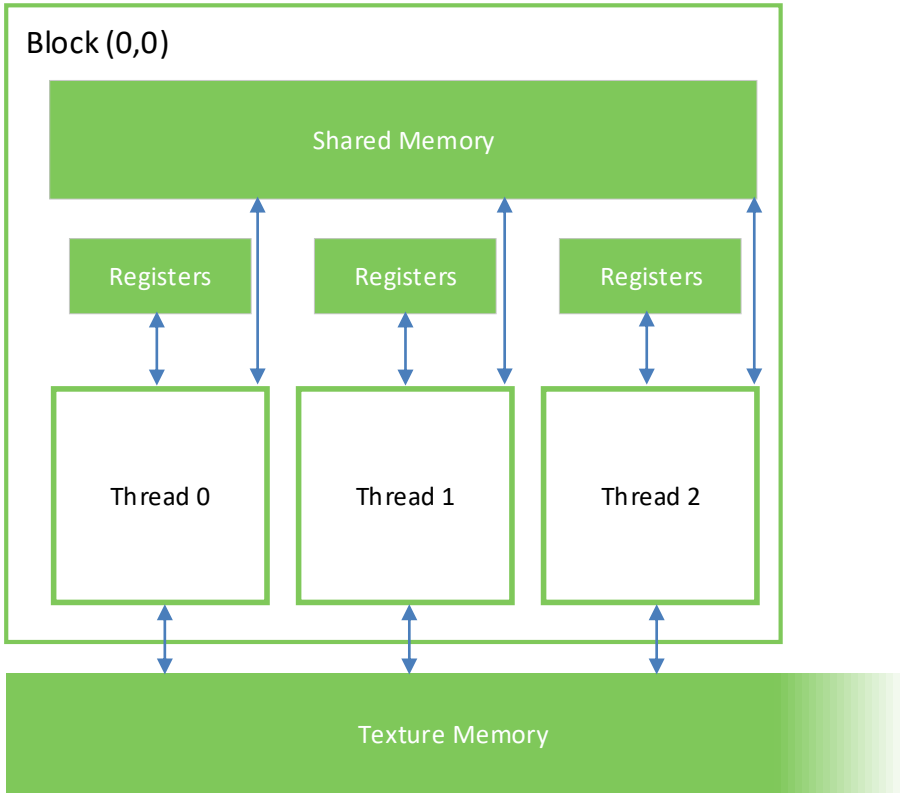
4.2 Implement Advanced Optimizations
Leverage Texture Cache



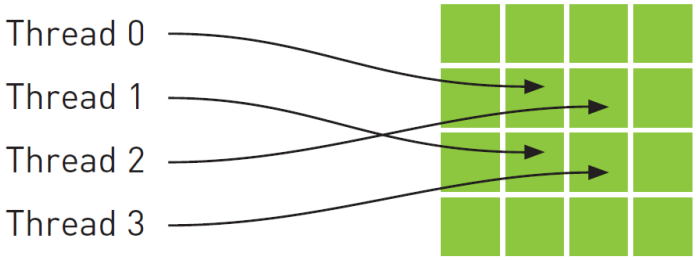
- Different memory cache
- Optimized for 2D spatial locality



4.2 Implement Advanced Optimizations
Leverage Texture Cache



- Different memory cache
- Optimized for 2D spatial locality



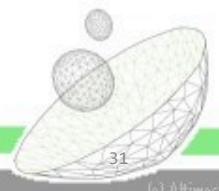
Bind input image to texture

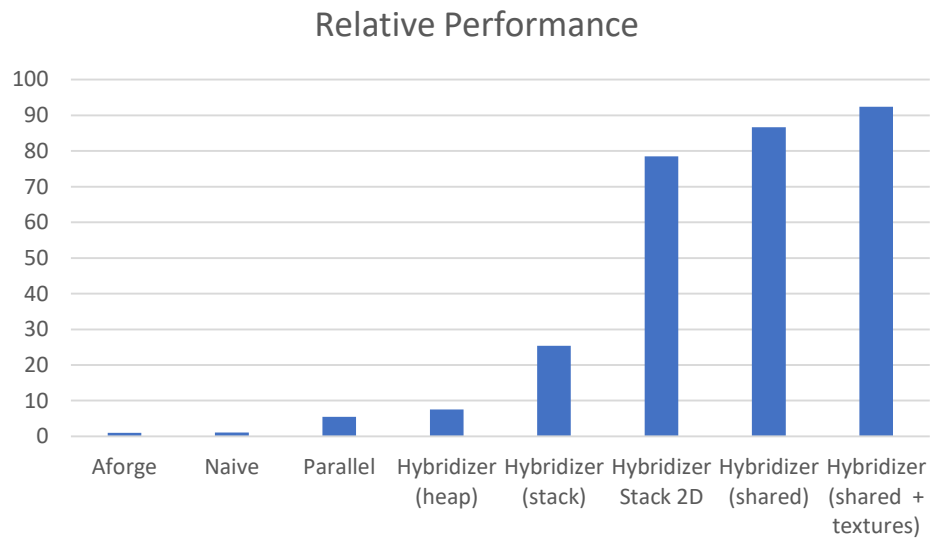
4.2 Implement Advanced Optimizations
Leverage Texture Memory

```
IntPtr src = runner.Marshaller.MarshalManagedToNative(pixels);
cudaChannelFormatDesc channelDescTex = TextureHelpers.cudaCreateChannelDesc(8, 0, 0, 0, cudaChannelFormatKind.cudaChannelFormatKindUnsigned);
cudaArray_t cuArrayTex = TextureHelpers.CreateCudaArray(channelDescTex, src, width, height);
cudaResourceDesc resDescTex = TextureHelpers.CreateCudaResourceDesc(cuArrayTex);

//create Texture descriptor
cudaTextureDesc texDesc = TextureHelpers.CreateCudaTextureDesc();

//create Texture object
cudaTextureObject_t texObj;
cuda.CreateTextureObject(out texObj, ref resDescTex, ref texDesc);
```

- CUDA API is fully available through a wrapper (P/Invoke)
- Texture and Surface API types are exposed and mapped (IntrinsicTypes)

- Resulting C# code for textures usage very similar to CUDA/C tutorials

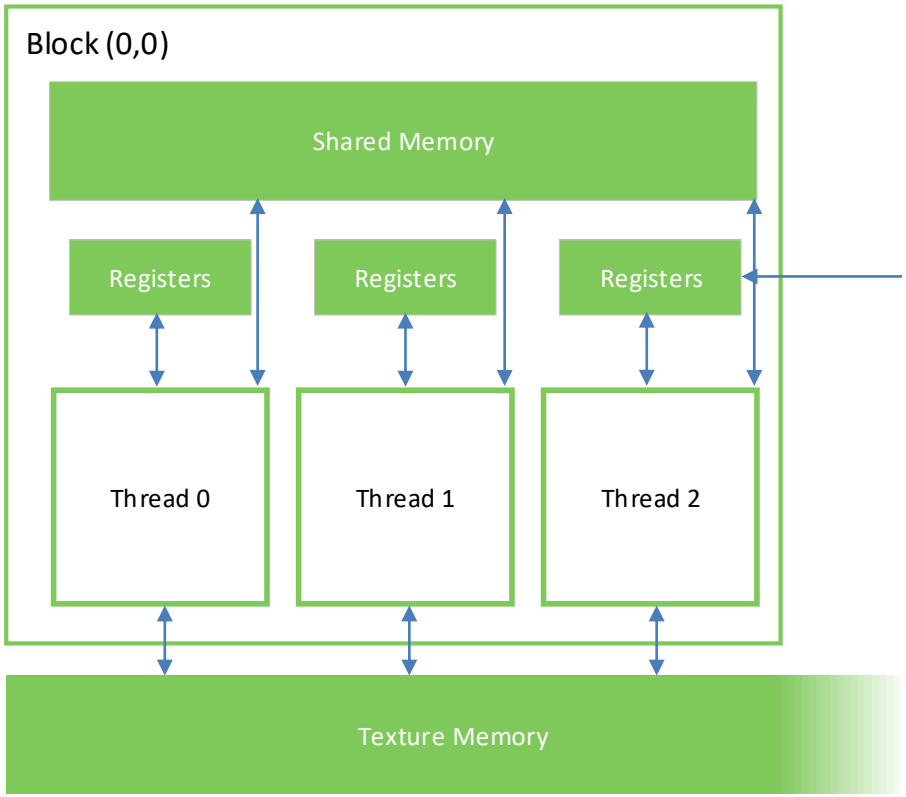




We accelerated AForge with a x92 speed-up.

Can we do better?

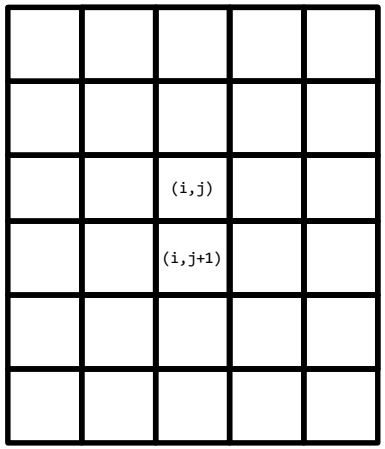
5. Implement Advanced Optimizations
What's next?



Put everything in register file

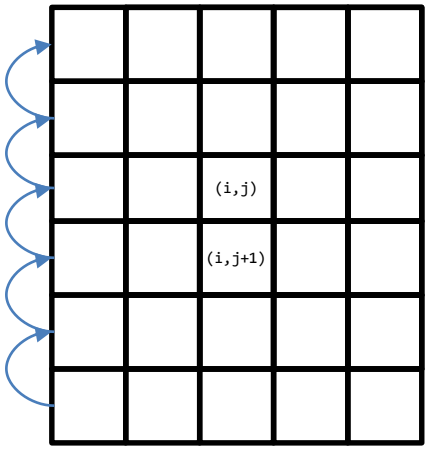
GPU SM have 32k registers for a Block – up to 255 by threads

5.1 Implement Advanced Optimizations
Rolling Buffer Of Registers



Load data in registers and process pixel (i,j)

5.1 Implement Advanced Optimizations
Rolling Buffer Of Registers



Load next line and roll buffer for pixel(i, j+1)

5.2 Implement Advanced Optimizations
Loop Unrolling

```
// preload window
for (int lj = -window; lj < window; ++lj)
{
    j = bj + lj;
    if (j < 0) j = 0;
    if (j >= height) j = height - 1;

    for (int li = -window; li <= window; ++li)
    {
        i = bi + li;
        if (i < 0) i = 0;
        if (i >= width) i = width - 1;

        filter.set_Item(index, input[j * width + i]);
    }
}
```

If window is a compile-time constant,
backend-compiler is able to completely unroll loop

(actually required for compiler to map arrays on registers)

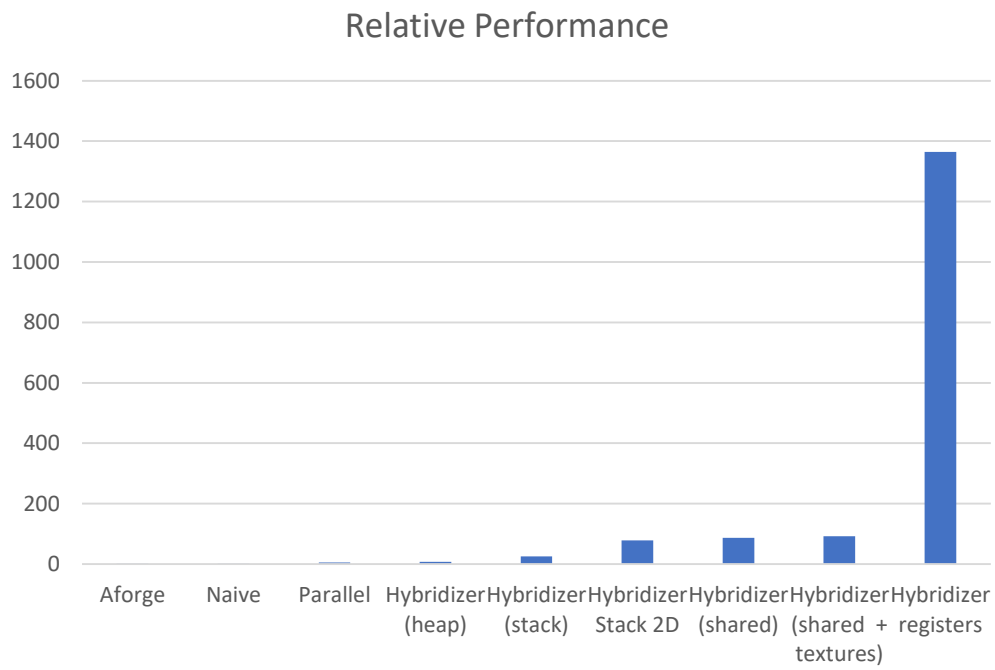
5.3 Implement Advanced Optimizations
Smart Sorting

- Sorting networks are optimal for known-size arrays.
- They are not capable of sorting arbitrary long arrays.
- Possible to implement in C++ meta-programming.

- Enabled with hand-written CUDA, called from C# using « IntrinsicType »

```
[IntrinsicInclude("intrinsics.cuh")]
[IntrinsicType("medianfilter<unsigned short, 3>")]
struct medianfilter_ushort_3
{
    public ushort apply() { … }
    public void rollbuffer() { … }
    public ushort get_Item(int i) { … }
    public void set_Item(int i, ushort val) { … }
}
```

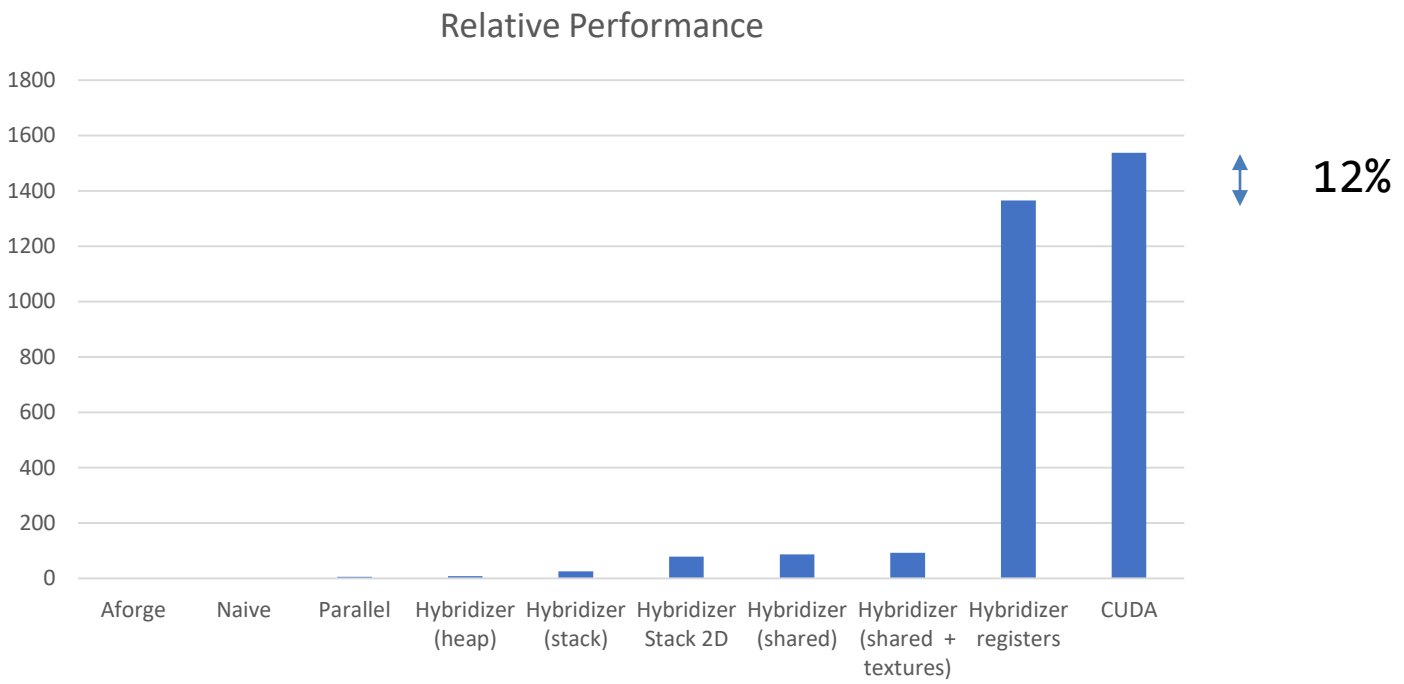
```
template <typename scalar, int window>
struct medianfilter
{
  static constexpr int size = (window * 2 + 1) * (window * 2 + 1);
  scalar buffer[size];
  scalar work[size];

__forceinline__ __device__ __host__ void set_Item(int i, scalar val) {
buffer[i] = val; }

__forceinline__  __device__ __host__ scalar apply()
{
  #pragma unroll
  for (int k = 0; k < size; ++k) {
    work[k] = buffer[k];
  }

  hybridizer::StaticSort<size> sort;
  sort(work);

  return work[size / 2];
}
```



Can we do better?

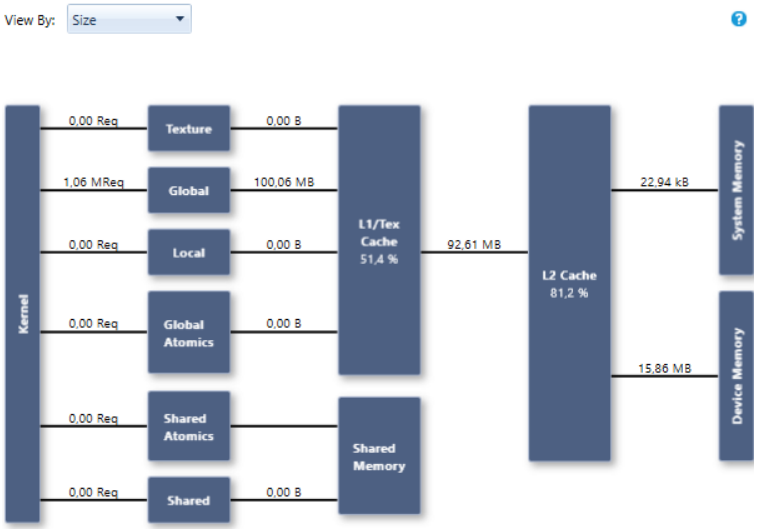
6. Write Plain CUDA

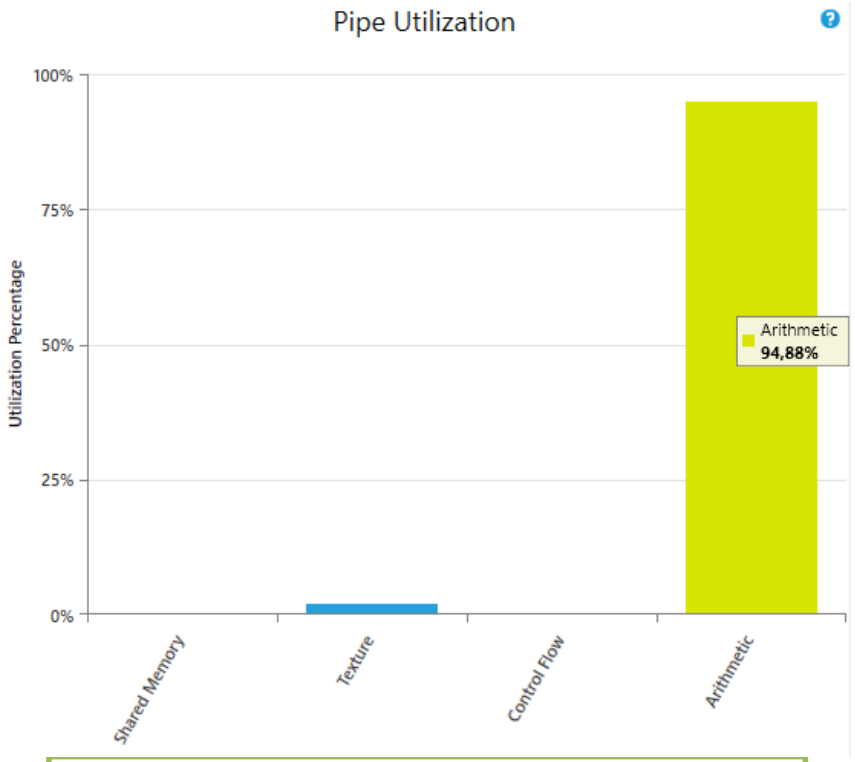
- Writing the entire application in CUDA/C leads to



Can we do better?

Maybe…

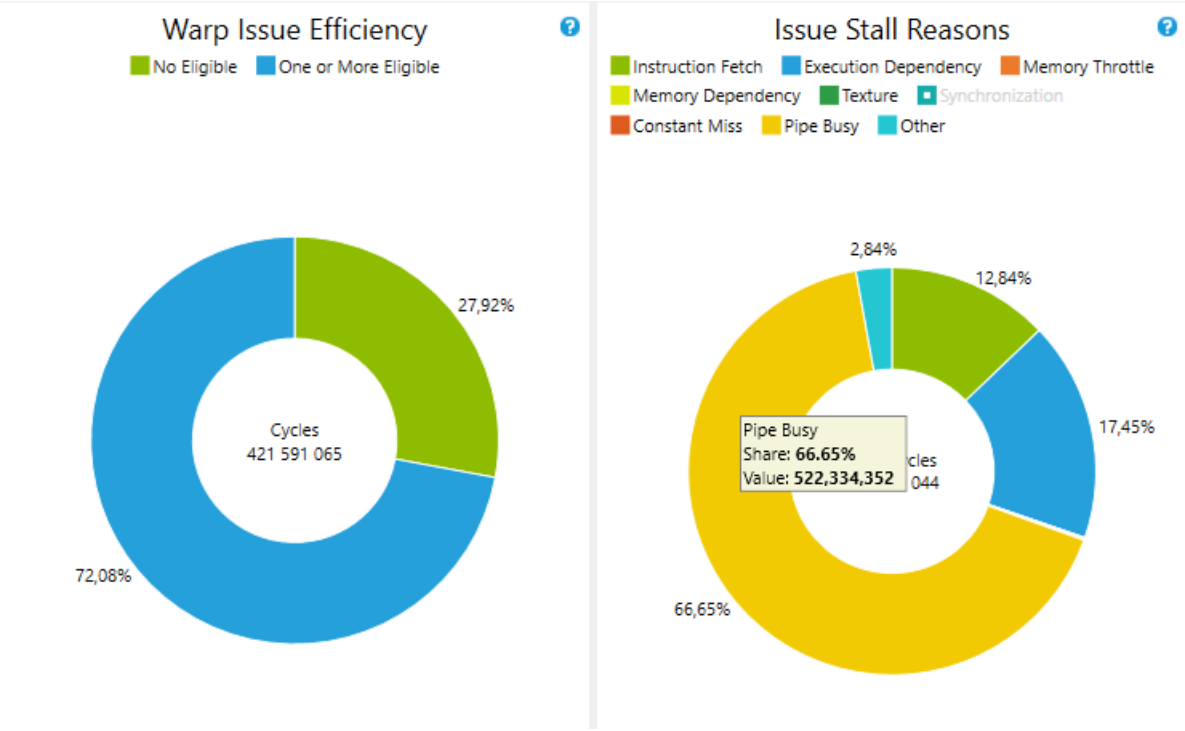




We barely read the image once
- 7.68 MB read
- 7.68 MB write
- 0.49 MB overhead

Room for improvement is 5%

Maybe…



Next in line : pipe busy…

7. Take Away

FLOPS double every

CPU: 1.8y
ACC: 1.9y

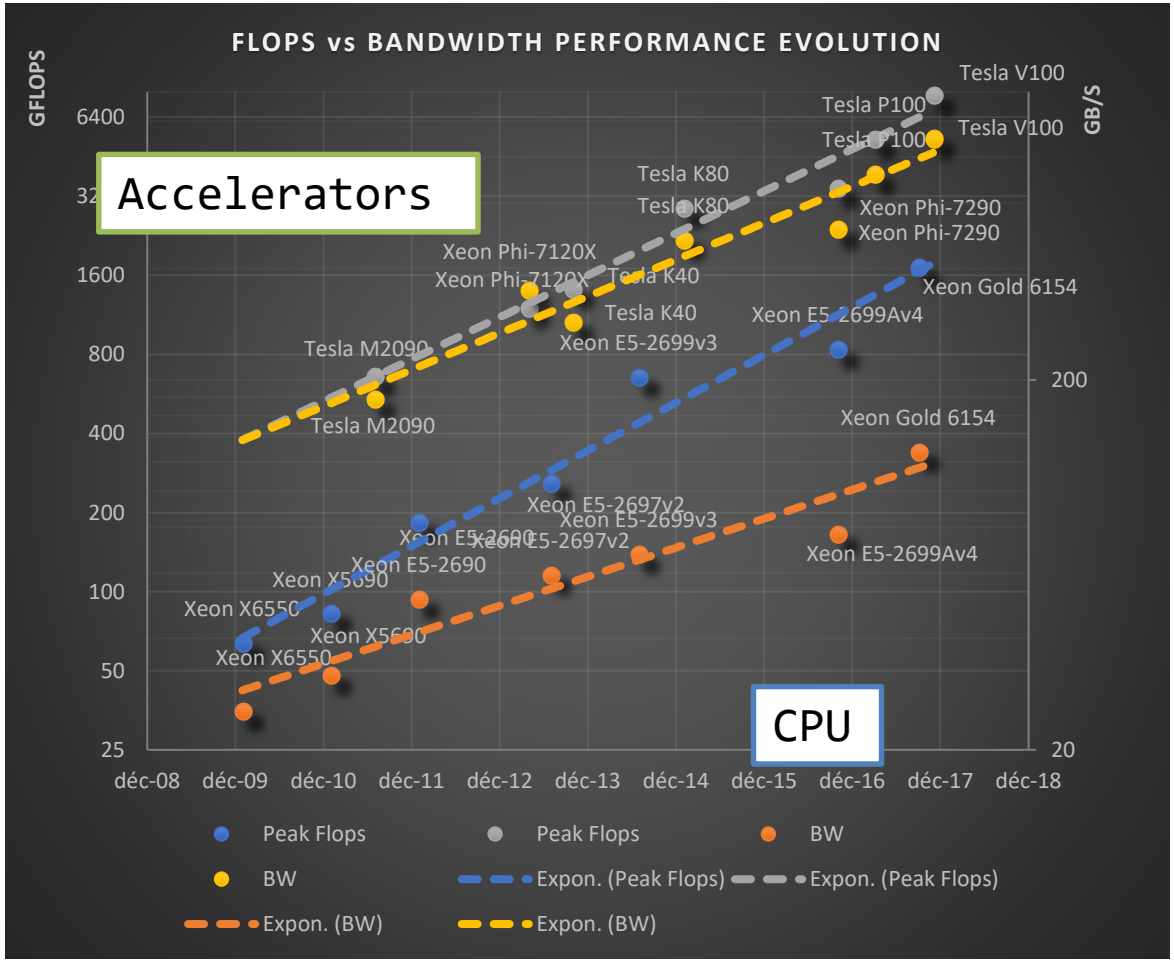
BANDWIDTH doubles every

CPU: 4.3y
ACC: 2.8y

Caching computations is not necessary anymore

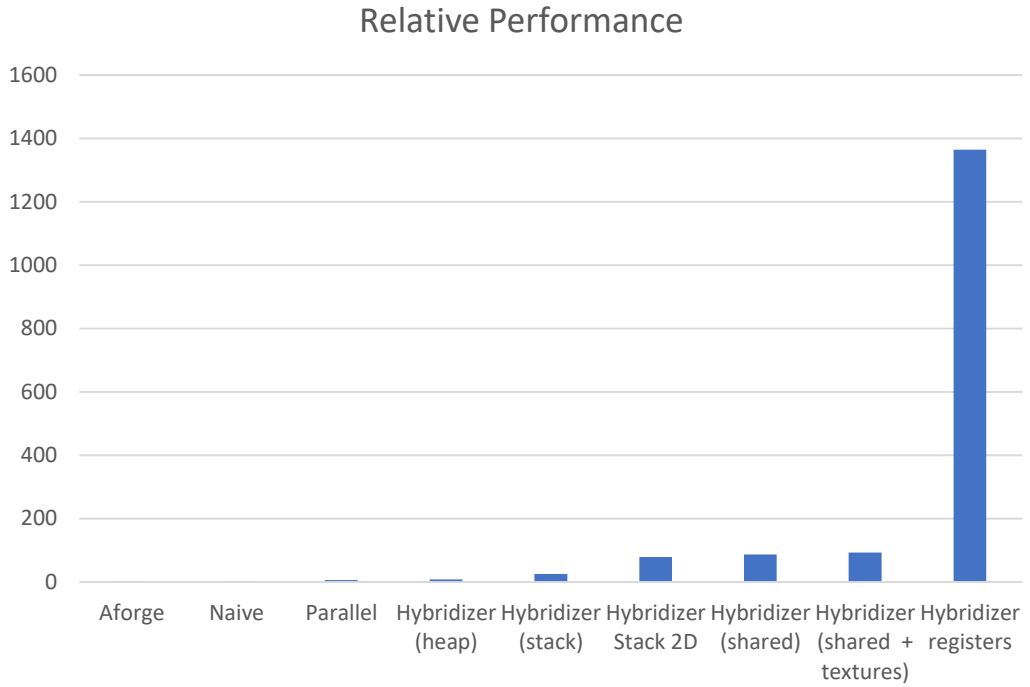
Caching memory operations is mandatory!

Always use the fastest memory available, the fastest of them all being registries



Memory interaction is the elephant in the room

Thank you



All performance measurements have been done on:
- Core I7 4770S@3.1 GHZ
- GeForce 1080 TI – 3584 cores @ 1.4GHz

Windows 10 x64

http://www.altimesh.com