Performance Optimization

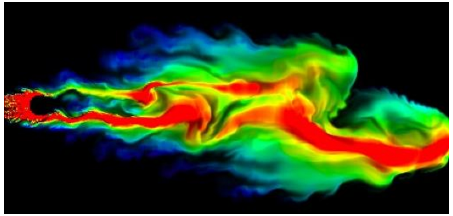
ALGORITHMS FOR GPGPU

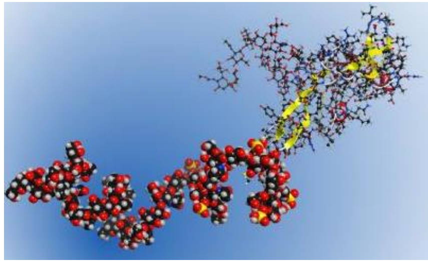
Algorithm Requirements

- High degree of parallelism
  - Thousands of threads
  - In multiple stages
  - Data parallelism
  - Fixed parallelism in each stage
- High arithmetic load
- Groups of threads work coherently
  - little branching
- Little need for reductions
- Similar execution time for all threads
- Few dependencies between successive kernels

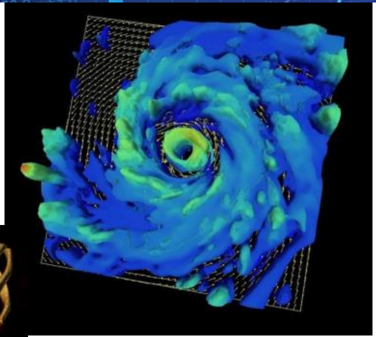
Well fitting algorithms 1

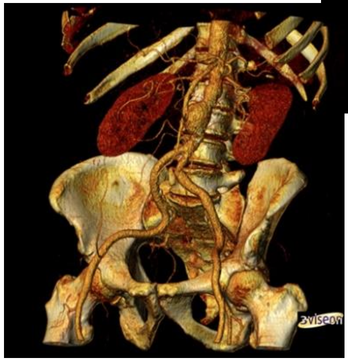
- Complex image filtering: 1 000 000 threads
  - per thread: a few inputs, one output
  - well aligned memory access
  - no dependencies
- Computational Fluid Dynamics: 10 000 threads
  - many iterations of complex math
  - well aligned memory access
- Protein sequencing: 8 000 threads
  - compute intensive Viterbi algorithm
  - well aligned memory access
  - threads finish at the same time





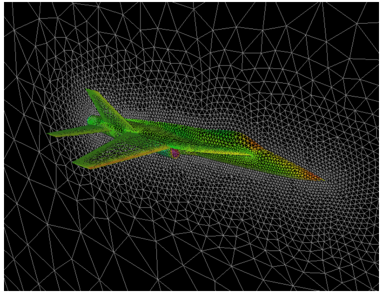
Well fitting algorithms 2



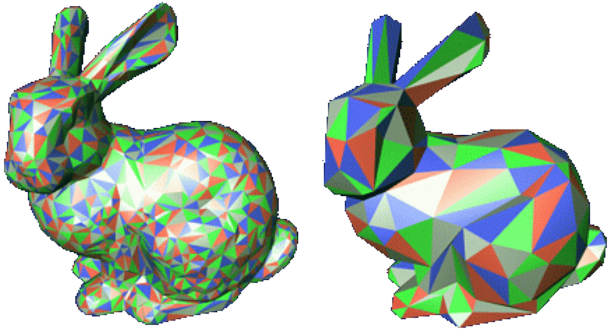


- Finance risk management: 100 000 threads
  - parallel Monte-Carlo simulations
  - independent execution paths
  - parallel reduction
- Weather Prediction: 1 000 000 threads
  - many iterations of complex math (2000+ ops)
  - well aligned memory access
- MRI reconstruction: 1 000 000 threads
  - use of FFT
  - many operations
  - well aligned memory access

Not so well fitting algorithms



- Volume rendering of unstructured grids: 1 000 000 threads
  - divergent paths based on hit cell
  - unstructured memory access patterns
  - varying execution durations
- Octree-based mesh simplification: 1 – 10 000 000 threads
  - varying parallelism throughout the algorithm
  - unordered writes, unknown dependencies between threads
  - low parallelism for low depths
  - execution configuration of next level depends on results from last
  - mapping threads to nodes complex



ARITHMETICS AND LATENCY

Average Instruction Latency per Core in Clock Cycles

	Instruction Latency						
Compute Capability	1.0–1.3	2.0	2.1	3.0	3.5	5.0, 5.2	5.3
Cores per SM	8	32	48	192	192	128	128
16-bit floating point add, mul, mad	-	-	-	-	-	-	½
32-bit floating point add, mul, mad	1	1	1	1	1	1	1
64-bit floating point add, mul, mad	8	2	6	24	3	32	32
32-bit integer add	0.8	1	1	1.2	1.2	1	1
32-bit integer compare	0.8	1	1	1.2	1.2	2	2
32-bit integer shift	1	2	3	6	3	2	2
32-bit bitwise logical operations	1	1	1	1.2	1.2	1	1
32-bit integer mul, mad	multiple	2	2	6	6	multiple	multiple
32-bit float rcp, sqrt, log, exp	4	8	6	6	6	4	4
type conversions	8	2	3	6	6	4	4

Instruction Throughput

	Device Throughput per cycle/per second				
Model/Compute Capability	GTX 280/1.3	GTX 580/2.0	GTX 680/3.0	GTX Titan Black/3.5	GTX Titan X
32-bit floating point add, mul, mad	240 / 305G	512 / 791G	1536 / 1625G	2880 / 2822G	3072 / 3072G
64-bit floating point add, mul, mad	30 / 38G	256 / 395G	64 / 68G	960 / 941G	96 / 96G
32-bit integer add	300 / 381G	512 / 791G	1344 / 1354G	2400 / 2352G	3072 / 3072G
32-bit integer compare	300 / 381G	512 / 791G	1344 / 1354G	2400 / 2352G	1536 / 1536G
32-bit integer shift	240 / 305G	256 / 395G	256 / 271G	960 / 941G	1536 / 1536G
32-bit bitwise logical operations	240 / 305G	512 / 791G	1280 / 1354G	2400 / 2352G	3072 / 3072G
32 bit integer mul, mad	?	256 / 395G	256 / 270G	480 / 470G	?
32 bit float recip, squrt, log, exp	60 / 76G	64 / 99G	256 / 270G	480 / 470G	768 / 768G
type conversions	30 / 38G	256 / 395G	256 / 270G	480 / 470G	768 / 768G

Latency hiding

- How many threads should be launched?
- Background
  - Instructions are issued in order
  - A thread is stalled when operands are not ready
  - Global memory latency: 20 – 400 cycles
  - Latency for arithmetics (pipeline depth): 18 - 22 cycles
  - Latency is hidden by switching threads
- Conclusion
  - We want enough threads to hide latency
  - Truth is > 90% of all implementations are latency-bound

Latency hiding: Occupancy

$$Occupancy = \frac{\#warps\ per\ mp}{\#max\ warps\ per\ mp}$$

- Higher occupancy the scheduler can choose from more threads → can hide latency better
- Use occupancy calculator to check what limits the number of warps per mp and adjust block size or algorithm
- Hide arithmetic latency (32 bit float)
  - Need ~18 warps (576 threads) per Fermi SM
  - Need ~108 warps (3456 threads) per Kepler SMX **(??)**
  - Max 64 warps (2048 threads) per Kepler SMX

Occupancy calculator

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	**3.5**
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	64	8
Registers	(Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)		4096	49152	12

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	8	8	64
Limited by Shared Memory per Multiprocessor	12		

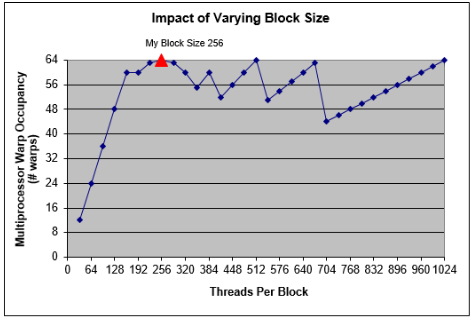
Note: Occupancy limiter is shown in orange

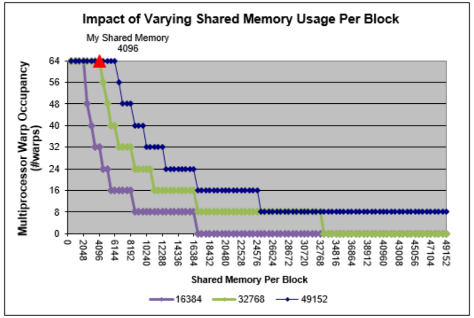
Physical Max Warps/SM = 64

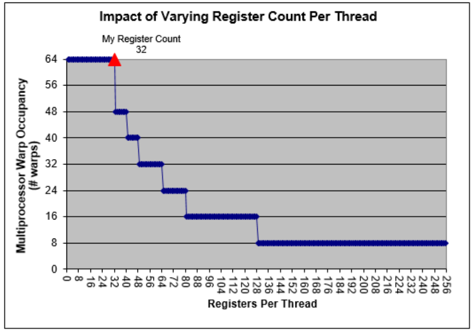
Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Interactive Demo

Latency hiding cont’d

- Latency hiding with independent instructions (instruction-level parallelism)

```
__global__ void memcpy(float *dst, float *src)
{
  uint block = blockIdx.x + blockIdx.y * gridDim.x;
  uint index = threadIdx.x + block * blockDim.x;

  float a0 = src[index];
  dst[index] = a0;
}
```

Latency hiding cont'd

- Latency hiding with independent instructions (instruction-level parallelism)

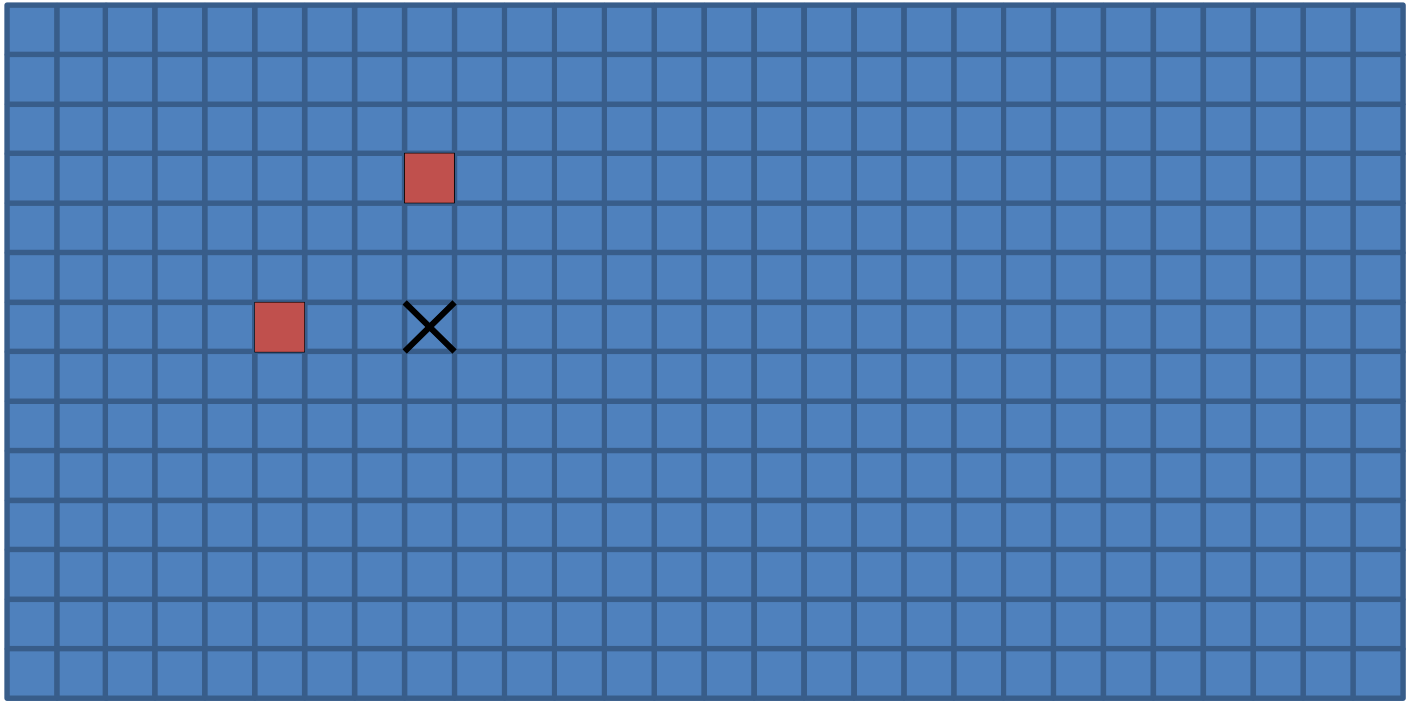
```
__global__ void memcpy(float *dst, float *src)
{
  uint block = blockIdx.x + blockIdx.y * gridDim.x;
  uint index = threadIdx.x + 2 * block * blockDim.x;

  float a0 = src[index];
  float a1 = src[index + blockDim.x];

  dst[index] = a0;
  dst[index + blockDim.x] = a1;
}
```

Latency hiding cont'd

- Occupancy is just one factor influencing performance
- Using more threads for the same work does not necessarily mean higher performance
- Always experiment with block size, grid size and algorithm layout
- General Guidelines
  - Really small threadblocks will reduce occupancy
  - Really large threadblocks are less flexible (especially interesting for varying execution time)
  - Start out with 128-256 theads/block and vary

SEPARABLE FILTERING

Filter X + Filter Y



Filtering Setup

```
texture<uchar4, 2, cudaReadModeNormalizedFloat> gaussInputTex;

const int gauss_kernel_size = 33;
const float gauss_kernel[] = {
  0.00288204f,  0.00418319f,  0.00592754f,  0.00819980f,  0.01107369f,  0.01459965f,
  0.01879116f,  0.02361161f,  0.02896398f,  0.03468581f,  0.04055144f,  0.04628301f,
  0.05157007f,  0.05609637f,  0.05957069f,  0.06175773f,  0.06250444f,  0.06175773f,
  0.05957069f,  0.05609637f,  0.05157007f,  0.04628301f,  0.04055144f,  0.03468581f,
  0.02896398f,  0.02361161f,  0.01879116f,  0.01459965f,  0.01107369f,  0.00819980f,
  0.00592754f,  0.00418319f,  0.00288204f };
const float gauss_kernel_offset[] = {
  -16.0f,  -15.0f,  -14.0f,  -13.0f,  -12.0f,  -11.0f,
  -10.0f,   -9.0f,   -8.0f,   -7.0f,   -6.0f,   -5.0f,
   -4.0f,   -3.0f,   -2.0f,   -1.0f,    0.0f,    1.0f,
    2.0f,    3.0f,    4.0f,    5.0f,    6.0f,    7.0f,
    8.0f,    9.0f,   10.0f,   11.0f,   12.0f,   13.0f,
   14.0f,  15.0f,  16.0f };

__constant__ float c_gauss_kernel[] = {
  0.00288204f,  0.00418319f,  0.00592754f,  0.00819980f,  0.01107369f,  0.01459965f,
  0.01879116f,  0.02361161f,  0.02896398f,  0.03468581f,  0.04055144f,  0.04628301f,
  0.05157007f,  0.05609637f,  0.05957069f,  0.06175773f,  0.06250444f,  0.06175773f,
  0.05957069f,  0.05609637f,  0.05157007f,  0.04628301f,  0.04055144f,  0.03468581f,
  0.02896398f,  0.02361161f,  0.01879116f,  0.01459965f,  0.01107369f,  0.00819980f,
  0.00592754f,  0.00418319f,  0.00288204f };
```

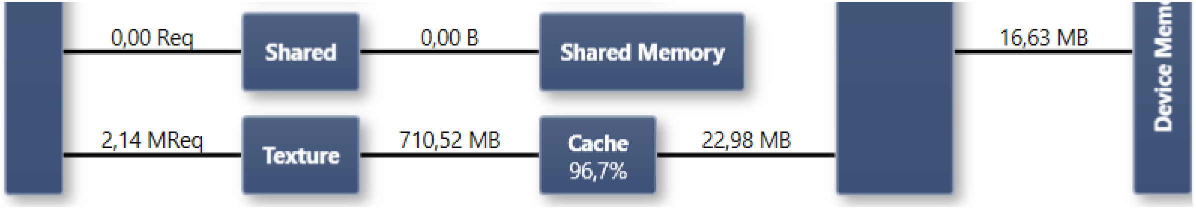
Straight Forward

```
__global__ void filterStraightForward(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  math::int2 pos(blockIdx.x * blockDim.x + threadIdx.x, blockIdx.y * blockDim.y + threadIdx.y);

  if (pos.x >= dimensions.x || pos.y >= dimensions.y)
    return;

  math::float3 val(0.0f);
  math::float2 filterpos(pos.x, pos.y);

  for (int i = 0; i < gauss_kernel_size; ++i)
  {
    math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
    math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
    val += pixel.xyz() * gauss_kernel[i];
  }

  output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01		1.285 4.936	81.89	

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01	0.273	1.285 4.936	81.89	0.411
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1

Straight Forward



straightForward<(120x68),(16x16)>

straightForward<(4x1080),(512x1)>

straightForward<(1920x3),(1x512)>

const

```
for (int i = 0; i < gauss_kernel_size; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
  val += pixel.xyz() * gauss_kernel[i];
}
```

```
math::float3 val(0.0f);
math::float2 filterpos = math::float2(pos.x, pos.y) - direction*static_cast<float>(gauss_kernel_size/2);
for (int i = 0; i < gauss_kernel_size; ++i, filterpos += direction)
{
  math::float4 pixel = toMathVec(tex2D(gaussInputTex, filterpos.x, filterpos.y));
  val += pixel.xyz() * c_gauss_kernel[i];
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01	0.273	1.285 4.936	81.89	0.411
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Const<(120,68),(16,16)> Const<(120,68),(16,16)>	0.767 0.762	333.3	0.939	1.278 1.704	170.90	0.858

const

```
tex.2d.v4.u32.f32 {%r120, %r121, %r122, %r123}, [gaussInputTex, {%f57, %f58}];
// inline asm
.loc 4 1778 5
mov.b32   %f235, %r120;
mov.b32   %f236, %r121;
mov.b32   %f237, %r122;
.loc 3 346 1
fma.rn.f32  %f238, %f235, 0f3C6F335F, %f232;
fma.rn.f32  %f239, %f236, 0f3C6F335F, %f233;
fma.rn.f32  %f240, %f237, 0f3C6F335F, %f234;
```

Kernel weights in opcode!

```
tex.2d.v4.u32.f32 {%r12, %r13, %r14, %r15}, [gaussInputTex, {%f50, %f49}];
// inline asm
.loc 4 1778 5
mov.b32   %f28, %r12;
mov.b32   %f29, %r13;
mov.b32   %f30, %r14;
.loc 2 78 1
ld.const.f32  %f31, [%rd17];
.loc 3 346 1
fma.rn.f32  %f32, %f31, %f28, %f53;
fma.rn.f32  %f33, %f31, %f29, %f52;
fma.rn.f32  %f34, %f31, %f30, %f51;
```

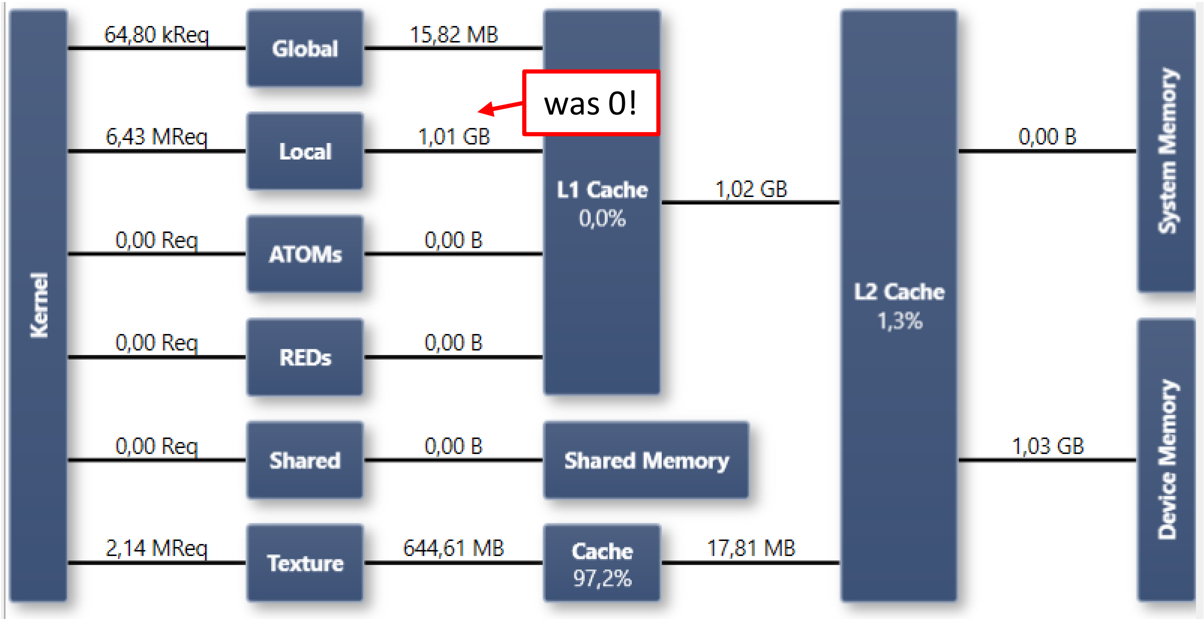
Checked

```
for (int i = 0; i < gauss_kernel_size; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
  val += pixel.xyz() * gauss_kernel[i];
}
```

```
for (int i = 0; i < gauss_kernel_size; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  if(tpos.x >= 0 && tpos.x < dimensions.x &&
     tpos.y >= 0 && tpos.y < dimensions.y)
  {
    math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
    val += pixel.xyz() * gauss_kernel[i];
  }
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01	0.273	1.285 4.936	81.89	0.411
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Const<(120,68),(16,16)> Const<(120,68),(16,16)>	0.767 0.762	333.3	0.939	1.278 1.704	170.90	0.858
Checked<(120,68),(16,16)> Checked<(120,68),(16,16)>	9.791 10.134	25.58	0.072	7.222 7.202	35.34	0.177

Checked



Kernel
64,80 kReq
Global
15,82 MB
was 0!
6,43 MReq
Local
1,01 GB
L1 Cache
0,0%
1,02 GB
L2 Cache
1,3%
0,00 B
System Memory
0,00 Req
ATOMs
0,00 B
0,00 Req
REDs
0,00 B
0,00 Req
Shared
0,00 B
Shared Memory
1,03 GB
Device Memory
2,14 MReq
Texture
644,61 MB
Cache
97,2%
17,81 MB

Checked

```
math::int2 pos(blockIdx.x * blockDim.x + threadIdx.x, blockIdx.y * blockDim.y + threadIdx.y);

if (pos.x >= dimensions.x || pos.y >= dimensions.y)
  return;

math::float3 val(0.0f);
math::float2 filterpos(pos.x, pos.y);

for (int i = 0; i < gauss_kernel_size; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  if(tpos.x >= 0 && tpos.x < dimensions.x &&
     tpos.y >= 0 && tpos.y < dimensions.y)
  {
    math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
    val += pixel.xyz() * gauss_kernel[i];
  }
}
```

2. Loop cannot be unrolled

1. if dependes on thread id

3. Indexing into array → local memory

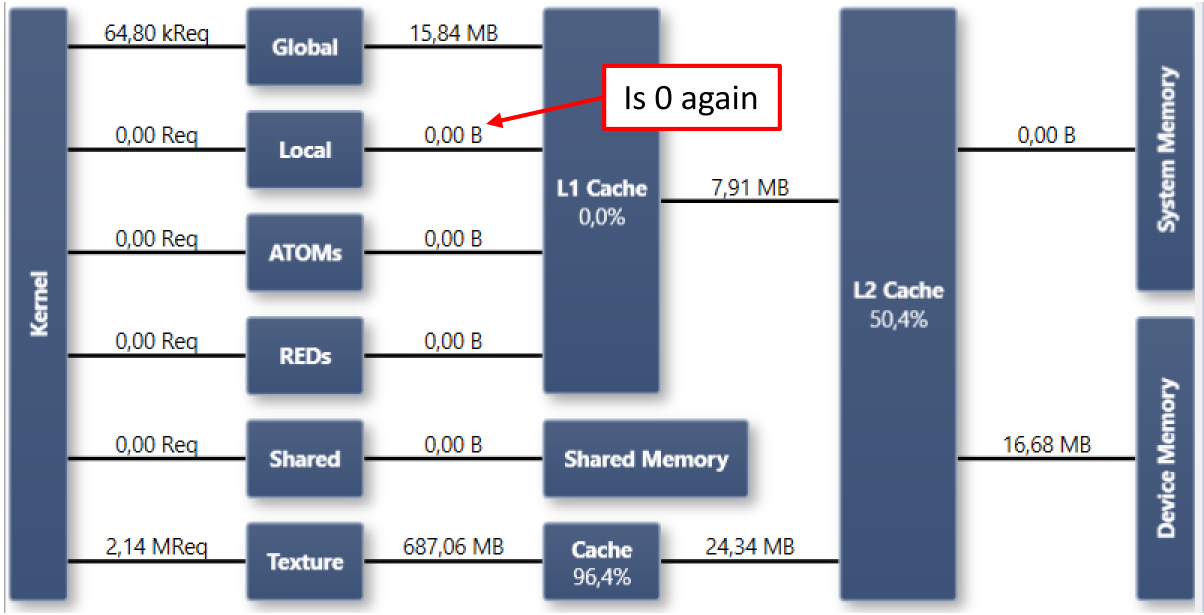
CheckedConst

```
for (int i = 0; i < gauss_kernel_size; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  if(tpos.x >= 0 && tpos.x < dimensions.x &&
     tpos.y >= 0 && tpos.y < dimensions.y)
  {
    math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
    val += pixel.xyz() * gauss_kernel[i];
  }
}
```

```
for (int i = 0; i < gauss_kernel_size; ++i, filterpos += direction)
{
  if(filterpos.x >= 0 && filterpos.x < dimensions.x &&
     filterpos.y >= 0 && filterpos.y < dimensions.y)
  {
    math::float4 pixel = toMathVec(tex2D(gaussInputTex, filterpos.x, filterpos.y));
    val += pixel.xyz() * c_gauss_kernel[i];
  }
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01	0.273	1.285 4.936	81.89	0.411
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Const<(120,68),(16,16)> Const<(120,68),(16,16)>	0.767 0.762	333.3	0.939	1.278 1.704	170.90	0.858
Checked<(120,68),(16,16)> Checked<(120,68),(16,16)>	9.791 10.134	25.58	0.072	7.222 7.202	35.34	0.177
CheckedConst<(120,68),(16,16)> CheckedConst<(120,68),(16,16)>	1.901 1.894	134.3	0.378	1.776 1.776	143.50	0.720

CheckedConst



Symmetric

```
for (int i = 0; i < gauss_kernel_size; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
  val += pixel.xyz() * gauss_kernel[i];
}
```

```
math::float2 filterpos(pos.x, pos.y);
math::float4 pixel = toMathVec(tex2D(gaussInputTex, filterpos.x, filterpos.y));
math::float3 val = pixel.xyz() * gauss_kernel[gauss_kernel_size/2];

#pragma unroll
for (int i = 0; i < gauss_kernel_size/2; ++i)
{
  math::float2 tpos = filterpos + direction*gauss_kernel_offset[i];
  math::float4 pixel = toMathVec(tex2D(gaussInputTex, tpos.x, tpos.y));
  math::float2 tpos2 = filterpos + direction*gauss_kernel_offset[gauss_kernel_size-1-i];
  math::float4 pixel2 = toMathVec(tex2D(gaussInputTex, tpos2.x, tpos2.y));
  val += (pixel.xyz() + pixel2.xyz()) * gauss_kernel[i];
}

output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01	0.273	1.285 4.936	81.89	0.411
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Const<(120,68),(16,16)> Const<(120,68),(16,16)>	0.767 0.762	333.3	0.939	1.278 1.704	170.90	0.858
Checked<(120,68),(16,16)> Checked<(120,68),(16,16)>	9.791 10.134	25.58	0.072	7.222 7.202	35.34	0.177
CheckedConst<(120,68),(16,16)> CheckedConst<(120,68),(16,16)>	1.901 1.894	134.3	0.378	1.776 1.776	143.50	0.720
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001

Shared float4X

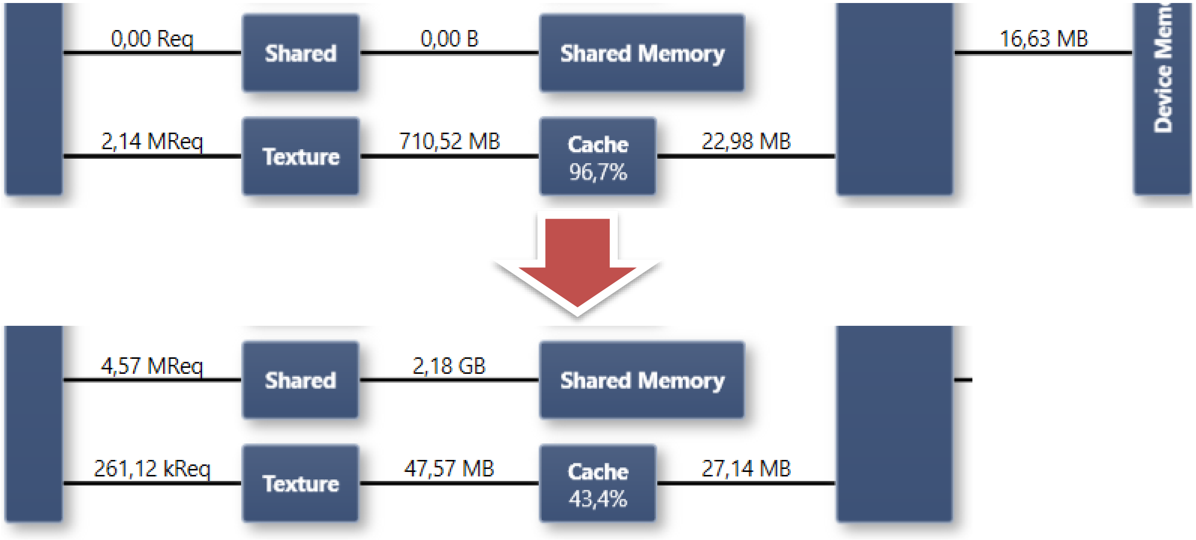
```
__global__ void filterSharedX4(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  extern __shared__ math::float4 f4_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int x = threadIdx.x; x < gauss_kernel_size + 16; x+=blockDim.x)
    f4_values[x + threadIdx.y*(gauss_kernel_size + 16)] =
      toMathVec(tex2D(gaussInputTex, blockpos.x+x-gauss_kernel_size/2, blockpos.y+threadIdx.y));
  __syncthreads();

  math::float3 val(0.0f);
  for (int i = 0; i < gauss_kernel_size; ++i)
    val += f4_values[threadIdx.x+i + threadIdx.y*(gauss_kernel_size + 16)].xyz() * gauss_kernel[i];

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

Shared float4Y

```
__global__ void filterSharedY4(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  extern __shared__ math::float4 f4_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int y = threadIdx.y; y < (gauss_kernel_size + 16); y+=blockDim.y)
    f4_values[threadIdx.x + y*16] =
      toMathVec(tex2D(gaussInputTex, blockpos.x+threadIdx.x, blockpos.y+y-gauss_kernel_size/2));
  __syncthreads();

  math::float3 val(0.0f);
  for (int i = 0; i < gauss_kernel_size; ++i)
    val += f4_values[threadIdx.x + (threadIdx.y+i)*16].xyz() * gauss_kernel[i];

   math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if (pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

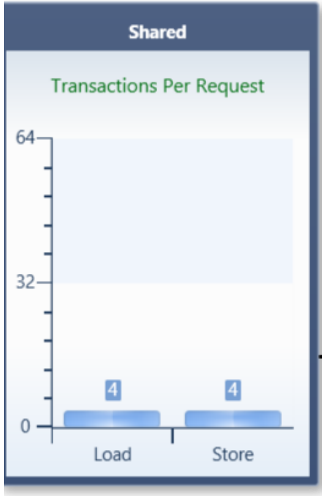
	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(4,1080),(512x1)> straightForward<(1920,3),(1x512)>	0.820 4.440	97.01	0.273	1.285 4.936	81.89	0.411
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Const<(120,68),(16,16)> Const<(120,68),(16,16)>	0.767 0.762	333.3	0.939	1.278 1.704	170.90	0.858
Checked<(120,68),(16,16)> Checked<(120,68),(16,16)>	9.791 10.134	25.58	0.072	7.222 7.202	35.34	0.177
CheckedConst<(120,68),(16,16)> CheckedConst<(120,68),(16,16)>	1.901 1.894	134.3	0.378	1.776 1.776	143.50	0.720
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797

Shared float4

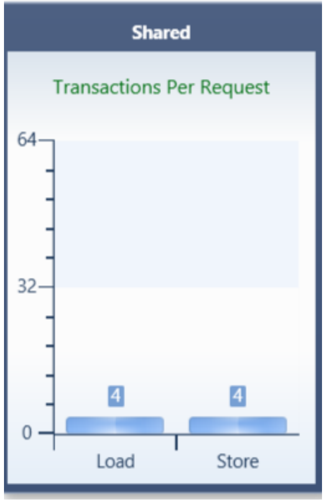


Shared memory is a limited resource and with this setup we can only start half the number of blocks concurrently

Shared float4



along
X



along
Y

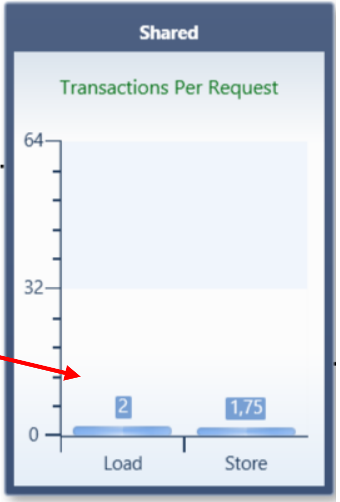
4x bank conflicts for x and y kernel!

Shared float3

```
__global__ void filterSharedX3(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  extern __shared__ math::float3 f3_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int x = threadIdx.x; x < (gauss_kernel_size + 16); x+=blockDim.x)
    f3_values[x + threadIdx.y*(gauss_kernel_size + 16)] =
      toMathVec(tex2D(gaussInputTex, blockpos.x+x-gauss_kernel_size/2, blockpos.y+threadIdx.y)).xyz();
  __syncthreads();

  math::float3 val(0.0f);
  for (int i = 0; i < gauss_kernel_size; ++i)
    val += f3_values[threadIdx.x+i + threadIdx.y*(gauss_kernel_size + 16)] * gauss_kernel[i];

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

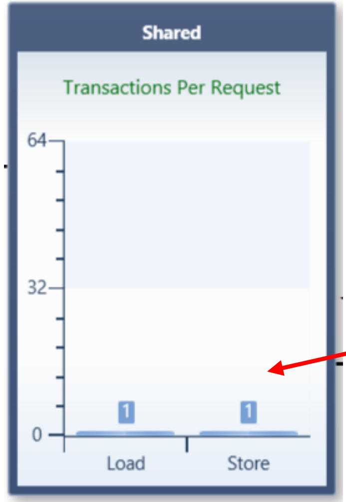
	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797
SharedX3<(120,68),(16,16)> SharedY3<(120,68),(16,16)>	1.850 1.131	171.0	0.481	1.167 0.723	269.59	1.354

Shared Float3



was 4

along
X



was 4

along
Y

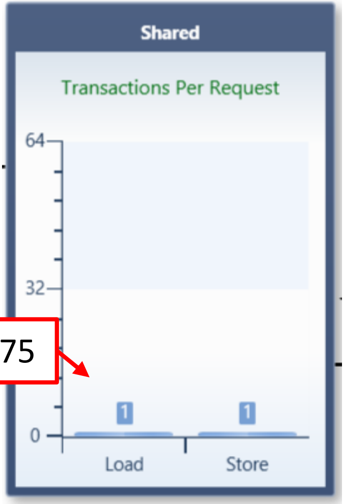
Shared float3X Separate

```
__global__ void filterSharedX3Separate(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  extern __shared__ float f_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int x = threadIdx.x; x < (gauss_kernel_size + 32); x+=blockDim.x)
  {
    math::float3 in = toMathVec(tex2D(gaussInputTex, blockpos.x+x-gauss_kernel_size/2, blockpos.y+threadIdx.y)).xyz();
    f_values[x + threadIdx.y*(gauss_kernel_size + 32)] = in.x;
    f_values[x + threadIdx.y*(gauss_kernel_size + 32) + (gauss_kernel_size + 32)*8] = in.y;
    f_values[x + threadIdx.y*(gauss_kernel_size + 32) + 2*(gauss_kernel_size + 32)*8] = in.z;
  }
  __syncthreads();

  math::float3 val(0.0f);
  #pragma unroll
  for (int i = 0; i < gauss_kernel_size; ++i)
  {
    math::float3 tval(f_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 32)],
                      f_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 32)+ (gauss_kernel_size + 32)*8],
                      f_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 32)+ 2*(gauss_kernel_size + 32)*8]);
    val += tval * gauss_kernel[i];
  }

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

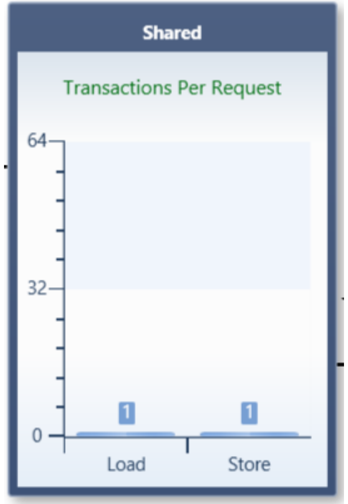
	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797
SharedX3<(120,68),(16,16)> SharedY3<(120,68),(16,16)>	1.850 1.131	171.0	0.481	1.167 0.723	269.59	1.354
SharedX3Separate<(60,135),(32,8)> SharedY3<(120,68),(16,16)>	0.935 1.136	246.0	0.693	0.702 0.722	357.82	1.797

Shared float3X Separate

was 2/1,75



along
X



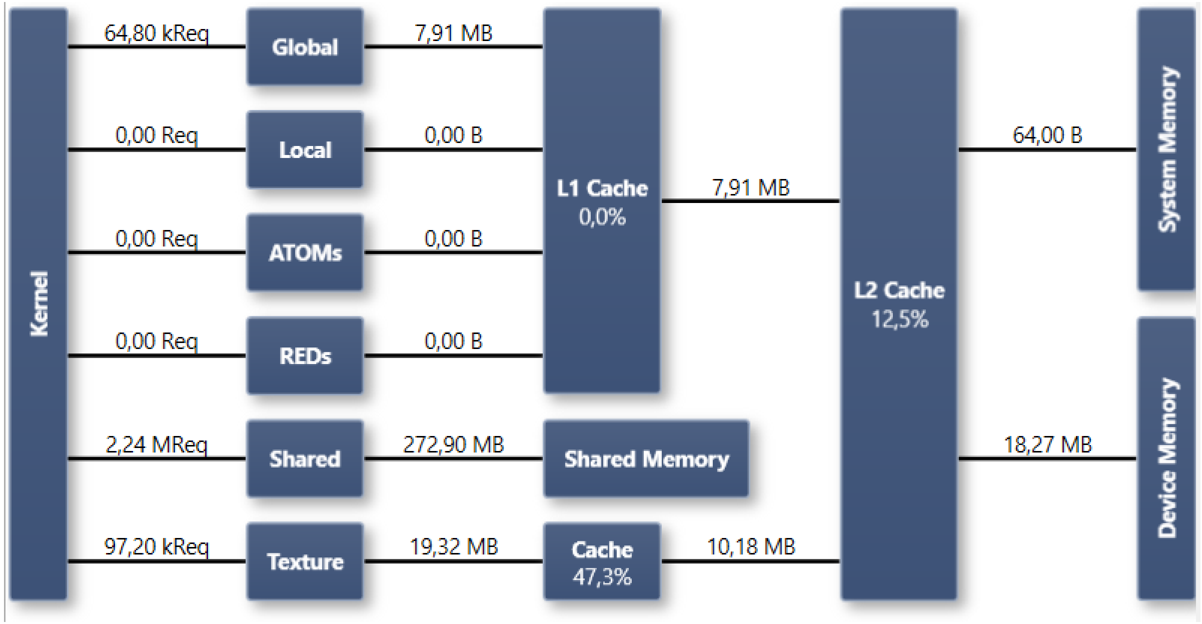
along
Y

Shared uchar4

```
__global__ void filterSharedXUchar4(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  extern __shared__ uchar4 uchar4_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int x = threadIdx.x; x < (gauss_kernel_size + 128); x+=blockDim.x)
    uchar4_values[x + threadIdx.y*(gauss_kernel_size + 128)] =
      tex2D(gaussInputTexElement, blockpos.x+x-gauss_kernel_size/2, blockpos.y+threadIdx.y);

  __syncthreads();

  math::float3 val(0.0f);
  #pragma unroll
  for (int i = 0; i < gauss_kernel_size; ++i)
  {
    uchar4 tuchar4 = uchar4_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 128)];
    math::float3 tval(tuchar4.x/255.0f,tuchar4.y/255.0f,tuchar4.z/255.0f);
    val += tval * gauss_kernel[i];
  }

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward <(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797
SharedX3<(120,68),(16,16)> SharedY3<(120,68),(16,16)>	1.850 1.131	171.0	0.481	1.167 0.723	269.59	1.354
SharedX3Separate<(60,135),(32,8)> SharedY3<(120,68),(16,16)>	0.935 1.136	246.0	0.693	0.702 0.722	357.82	1.797
SharedXUchar4<(15,135),(128,8)> SharedYUchar4<(240,9),(8,128)>	8.322 7.314	32.60	0.091	9.675 8.505	28.04	0.140

Shared uchar4



same as with float3 + no bank conflicts either …?

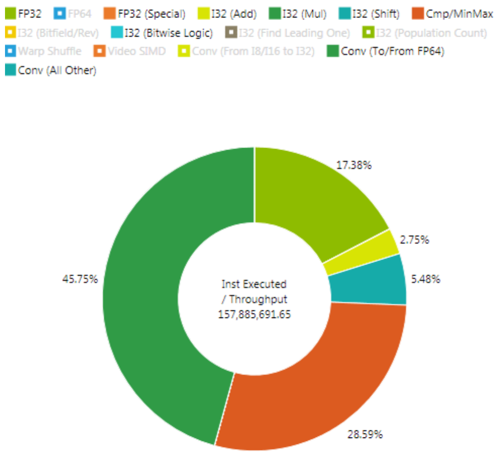
Shared uchar4

```
__global__ void filterSharedXUchar4(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL;
  extern __shared__ uchar4 uchar4_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int x = threadIdx.x; x < (gauss_kernel_size + 128); x+=blockDim.x)
    uchar4_values[x + threadIdx.y*(gauss_kernel_size + 128)] =
      tex2D(gaussInputTexElement, blockpos.x+x-gauss_kernel_size/2, blockpos.y+threadIdx.y);

  __syncthreads();

  math::float3 val(0.0f);
  #pragma unroll
  for (int i = 0; i < gauss_kernel_size; ++i)
  {
    uchar4 tuchar4 = uchar4_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 128)];
    math::float3 tval(tuchar4.x/255.0f,tuchar4.y/255.0f,tuchar4.z/255.0f);
    val += tval * gauss_kernel[i];
  }

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

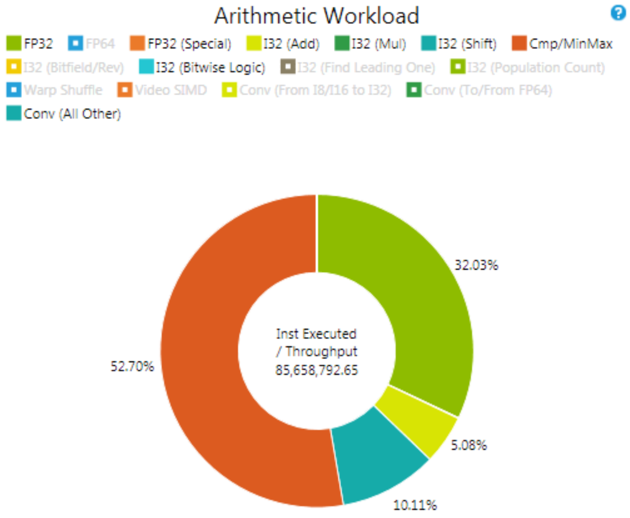
conversion is very slow!

Intermezzo: Conversion Creep

```
if (tmax < 10e-18)
{
  return FLT_MAX;
}
```



add missing 'f'



Shared uchar4Int

```
#define GAUSS_KERNEL_UINT \
  const unsigned int gauss_kernel_uint[] = { \
    3022, 4386, 6215, 8598, 11612, 15309, 19704, 24759, 30371, 36371, 42521, 48531, \
    54075, 58821, 62464, 64758, 65541, 64758, 62464, 58821, 54075, 48531, 42521, 36371, \
    30371, 24759, 19704, 15309, 11612, 8598, 6215, 4386, 3022 };
```

gauss_kernel *1024*1024

```
  math::uint3 val(0);
  #pragma unroll
  for (int i = 0; i < gauss_kernel_size; ++i)
  {
    uchar4 tuchar4 = uchar4_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 128)];
    val.x += tuchar4.x * gauss_kernel_uint[i];
    val.y += tuchar4.y * gauss_kernel_uint[i];
    val.z += tuchar4.z * gauss_kernel_uint[i];
  }

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x>>20, val.y>>20, val.z>>20, 255u);
}
```

no int to float conversion

no float to int conversion

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797
SharedX3<(120,68),(16,16)> SharedY3<(120,68),(16,16)>	1.850 1.131	171.0	0.481	1.167 0.723	269.59	1.354
SharedX3Separate<(60,135),(32,8)> SharedY3<(120,68),(16,16)>	0.935 1.136	246.0	0.693	0.702 0.722	357.82	1.797
SharedXUchar4<(15,135),(128,8)> SharedYUchar4<(240,9),(8,128)>	8.322 7.314	32.60	0.091	9.675 8.505	28.04	0.140
SharedXUchar4Int<(15,135),(128,8)> SharedYUchar4Int<(240,9),(8,128)>	1.351 1.512	177.97	0.501	1.187 1.283	206.32	1.036

still char to int conversion;
floating point fused multiple add fastest instruction

Shared uchar4 Load Mem

```
__global__ void filterSharedXUchar4IntLoadMem(const uchar4* input, uchar4* output,
  int adjustedPitch, math::int2 dimensions, math::float2 invtexsize, math::float2 direction)
{
  GAUSS_KERNEL_UINT;
  extern __shared__ uchar4 uchar4_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);


  for(int x = threadIdx.x; x < (gauss_kernel_size + 128); x+=blockDim.x)
  {
    int px = blockpos.x+x-gauss_kernel_size/2;
    int py = blockpos.y+threadIdx.y;
    px = min(max(px,0),dimensions.x-1);
    py = min(max(py,0),dimensions.y-1);
    uchar4_values[x + threadIdx.y*(gauss_kernel_size + 128)] = input[py * dimensions.x + px];
  }

  __syncthreads();

  math::uint3 val(0);
  #pragma unroll
  for (int i = 0; i < gauss_kernel_size; ++i)
  {
    uchar4 tuchar4 = uchar4_values[threadIdx.x+i+ threadIdx.y*(gauss_kernel_size + 128)];
    val.x += tuchar4.x * gauss_kernel_uint[i];
    val.y += tuchar4.y * gauss_kernel_uint[i];
    val.z += tuchar4.z * gauss_kernel_uint[i];
  }

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if(pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x>>20, val.y>>20, val.z>>20, 255u);
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797
SharedX3<(120,68),(16,16)> SharedY3<(120,68),(16,16)>	1.850 1.131	171.0	0.481	1.167 0.723	269.59	1.354
SharedX3Separate<(60,135),(32,8)> SharedY3<(120,68),(16,16)>	0.935 1.136	246.0	0.693	0.702 0.722	357.82	1.797
SharedXUchar4<(15,135),(128,8)> SharedYUchar4<(240,9),(8,128)>	8.322 7.314	32.60	0.091	9.675 8.505	28.04	0.140
SharedXUchar4Int<(15,135),(128,8)> SharedYUchar4Int<(240,9),(8,128)>	1.351 1.512	177.97	0.501	1.187 1.283	206.32	1.036
SharedXUchar4IntLoadMem<(15,135),(128,8)> SharedYUchar4IntLoadMem<(240,9),(8,128)>	1.364 1.554	174.68	0.492	1.179 1.386	198.68	0.997

Shared Fused

- avoid kernel call overhead
- avoid intermediate read and write
- while loading data to shared perform filtering along one dimension
- for big filter sizes, many values or computed multiple times

```
template<int BlockDimX, int BlockDimY>
__global__ void filterFused(const uchar4* input, uchar4* output, int adjustedPitch,
  math::int2 dimensions, math::float2 invtexsize)
{
  GAUSS_KERNEL;
  const int YElements = BlockDimY+gauss_kernel_size;

  extern __shared__ math::float3 f3_values[];
  math::int2 blockpos(blockIdx.x*blockDim.x, blockIdx.y*blockDim.y);

  for(int y = threadIdx.y; y < YElements; y+=BlockDimY)
  {
    math::int2 filterpos(blockpos.x+threadIdx.x-gauss_kernel_size/2, blockpos.y+y-gauss_kernel_size/2);
    math::float3 in_val(0.0f);

    #pragma unroll
    for (int i = -gauss_kernel_size/2; i <= gauss_kernel_size/2; ++i)
    {
      math::float4 pixel = toMathVec(tex2D(gaussInputTex, filterpos.x+i, filterpos.y));
      in_val += pixel.xyz() * gauss_kernel[i];
    }

    f3_values[threadIdx.x + y*BlockDimX] = in_val;
  }
  __syncthreads();

  math::float3 val(0.0f);
  #pragma unroll
  for (int i = 0; i < gauss_kernel_size; ++i)
    val += f3_values[threadIdx.x + (i+threadIdx.y)*BlockDimX] * gauss_kernel[i];

  math::int2 pos = blockpos + math::int2(threadIdx.x, threadIdx.y);
  if (pos.x < dimensions.x && pos.y < dimensions.y)
    output[pos.y*adjustedPitch + pos.x] = make_uchar4(val.x*255.0f, val.y*255.0f, val.z*255.0f, 255);
}
```

	ms 680	GB/s 680	speed up	ms 580	GB/s 580	speed up
straightForward<(120,68),(16x16)> straightForward<(120,68),(16x16)>	0.720 0.716	354.9	1	1.282 1.277	199.15	1
Const<(120,68),(16,16)> Const<(120,68),(16,16)>	0.767 0.762	333.3	0.939	1.278 1.704	170.90	0.858
Checked<(120,68),(16,16)> Checked<(120,68),(16,16)>	9.791 10.134	25.58	0.072	7.222 7.202	35.34	0.177
CheckedConst<(120,68),(16,16)> CheckedConst<(120,68),(16,16)>	1.901 1.894	134.3	0.378	1.776 1.776	143.50	0.720
Symmetric<(120,68),(16,16)> Symmetric<(120,68),(16,16)>	0.693 0.690	368.4	1.038	1.278 1.277	199.43	1.001
SharedX4<(120,68),(16,16)> SharedY4<(120,68),(16,16)>	2.433 1.832	119.5	0.336	1.624 1.586	158.78	0.797
SharedX3<(120,68),(16,16)> SharedY3<(120,68),(16,16)>	1.850 1.131	171.0	0.481	1.167 0.723	269.59	1.354
SharedX3Separate<(60,135),(32,8)> SharedY3<(120,68),(16,16)>	0.935 1.136	246.0	0.693	0.702 0.722	357.82	1.797
SharedXUchar4<(15,135),(128,8)> SharedYUchar4<(240,9),(8,128)>	8.322 7.314	32.60	0.091	9.675 8.505	28.04	0.140
SharedXUchar4Int<(15,135),(128,8)> SharedYUchar4Int<(240,9),(8,128)>	1.351 1.512	177.97	0.501	1.187 1.283	206.32	1.036
filterFused2<fusedX,fusedY><(240,17),(8,64)>	2.267	224.87	0.633	3.486	146.23	0.734

Questions