Optimization Opportunities and Pitfalls when Implementing High Performance 2D Convolutions

S4297

Ian Wainwright

High Performance Consulting Sweden

ian.wainwright@hpcsweden.se

2D Convolutions
What are they?

Filter

3	3	1
1	1	2
1	2	3

Image

1	2	3	1
4	5	2	3
4	1	1	5
1	2	5	2

Output


```
For all image pixels
    For all filter elements
        output += In(x,y)*Fil(x,y)
```

2D Convolutions
What are they?

Filter

3	3	1
1	1	2
1	2	3

Image

1	2	3	1
4	5	2	3
4	1	1	5
1	2	5	2

Output


```
For all image pixels
    For all filter elements
        output += In(x,y)*Fil(x,y)
```

2D Convolutions
What are they?

Filter

3	3	1
1	1	2
1	2	3

Image

1 * 3	2 * 3	3 * 1	1
4 * 1	5 * 1	2 * 2	3
4 * 1	1 * 2	1 * 3	5
1	2	5	2

Output


```
For all image pixels
    For all filter elements
        output += In(x,y)*Fil(x,y)
```

2D Convolutions
What are they?

Filter

3	3	1
1	1	2
1	2	3

Image

1 * 3	2 * 3	3 * 1	1
4 * 1	5 * 1	2 * 2	3
4 * 1	1 * 2	1 * 3	5
1	2	5	2

Output

	34		

```
For all image pixels
    For all filter elements
        output += In(x,y)*Fil(x,y)
```

3	6	3
4	5	4
4	2	3

$\sum = 34$

2D Convolutions

Tesla K20
GFLOPS = 3521 GFLOPS
BW 208 GB/s = 52 GigaFLOAT/s
→ 3521 GFLOPS/ 52 GFLOAT/s = ***67*** FLOPS / FLOAT is the theoretical break-even between bandwidth bound and compute bound.

The number of computations per output element in a 2D convolution is filter size * filter size * 2. → filter size = √(2 * ***67***) = 11.5
→ break-even between compute and bandwidth bound.
A 13*13 2D convolution is in theory compute bound for the Tesla K20.
Smaller sizes are bandwidth bound.

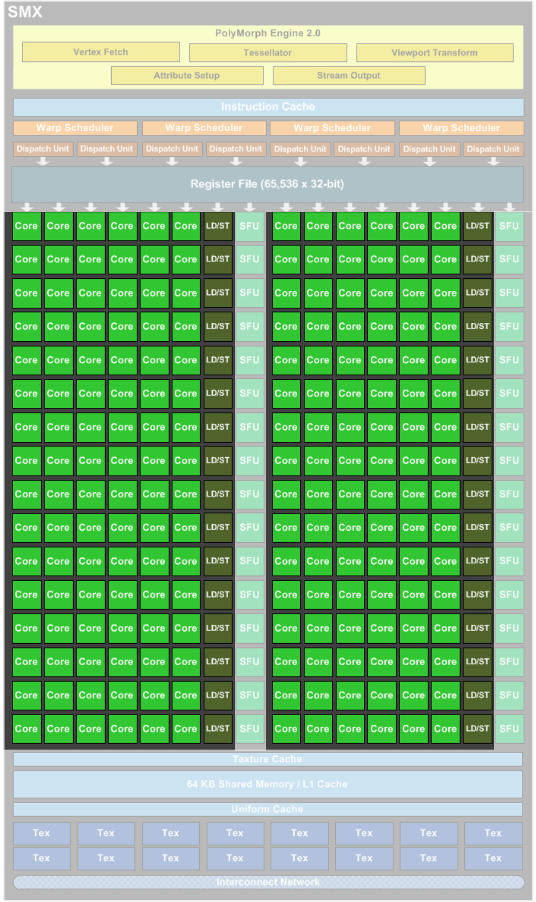
The Tesla K20

Kepler 110 Whitepaper

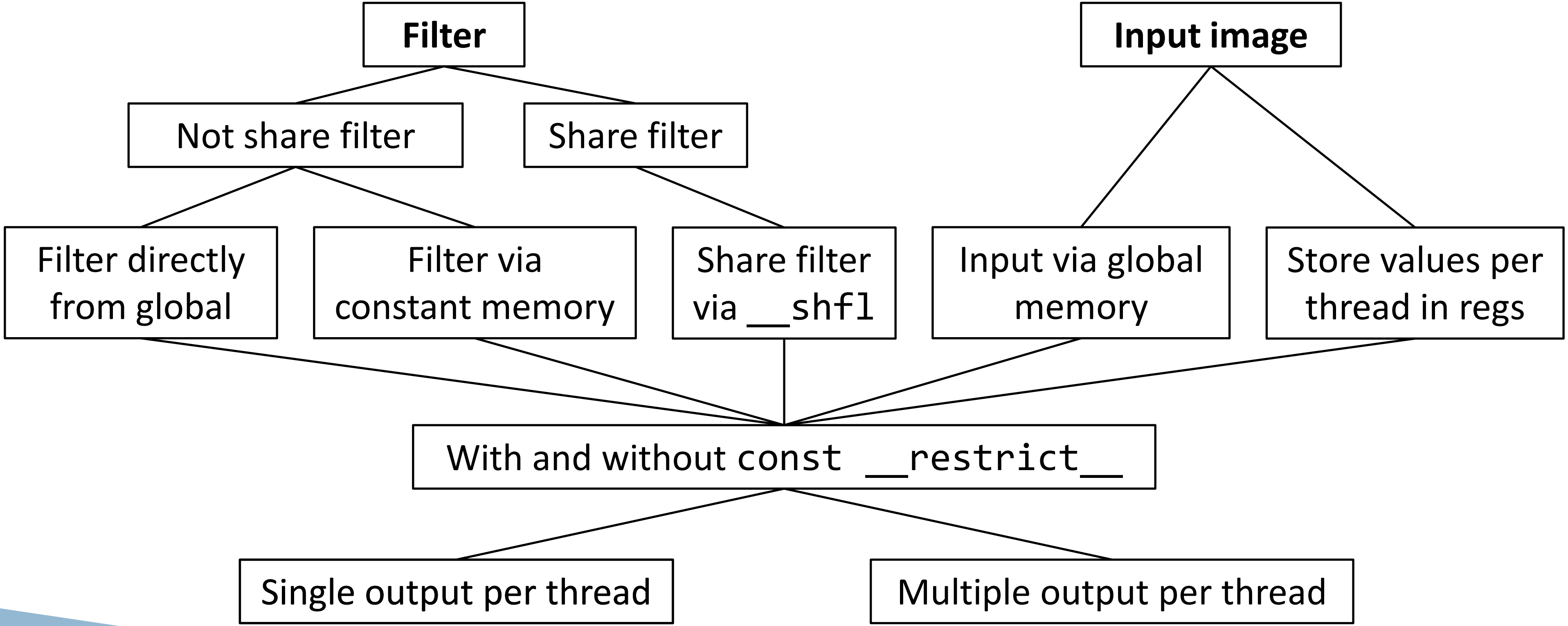
"192 single-precision floating point units"

"32 load/store units (LD/ST)"

→ 6 FMADs per LD/ST



The 2D Convolutions Implemented



The 2D Convolutions Implemented

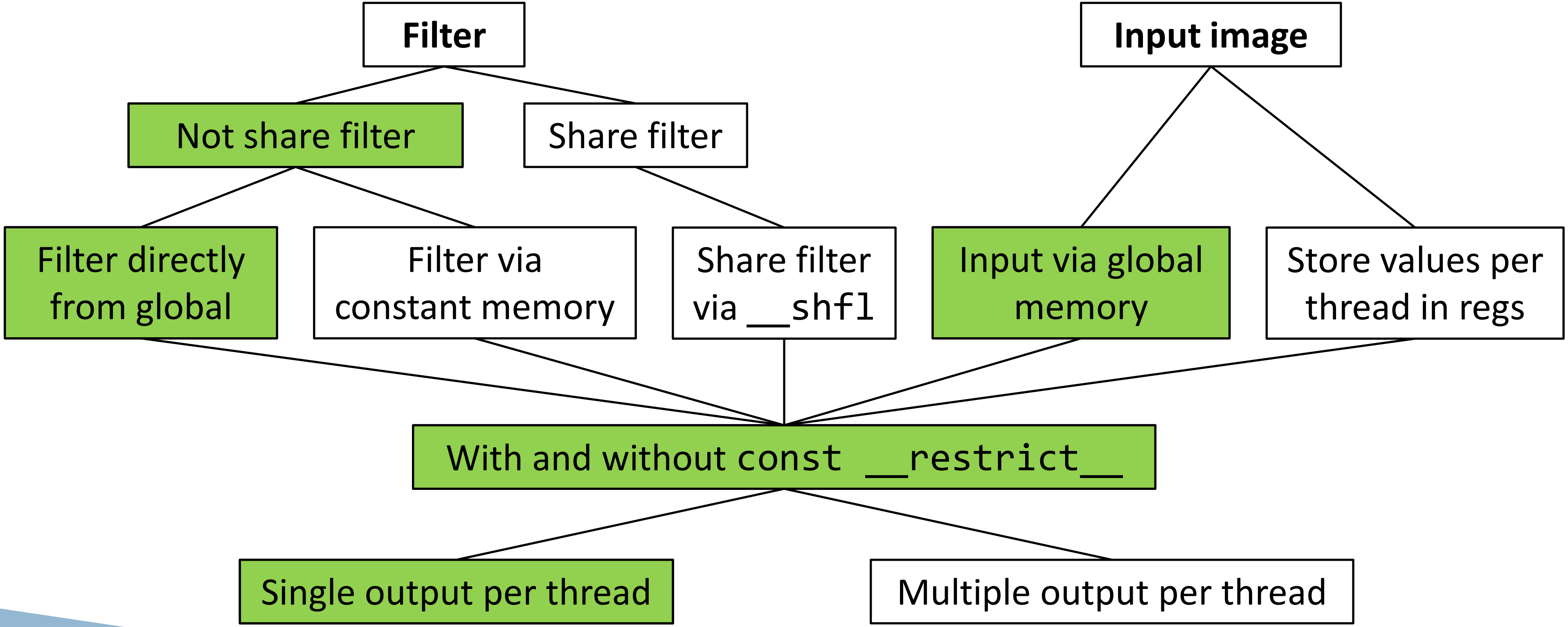
Filter

Input image

No const `__restrict__`	`const __restrict__` on filter only	`const __restrict__` on input only	`const __restrict__` on both
No restrict	Filter restrict	Input restrict	Both restrict

With and without `const __restrict__`

The 2D Convolutions Implemented



Everything in global memory

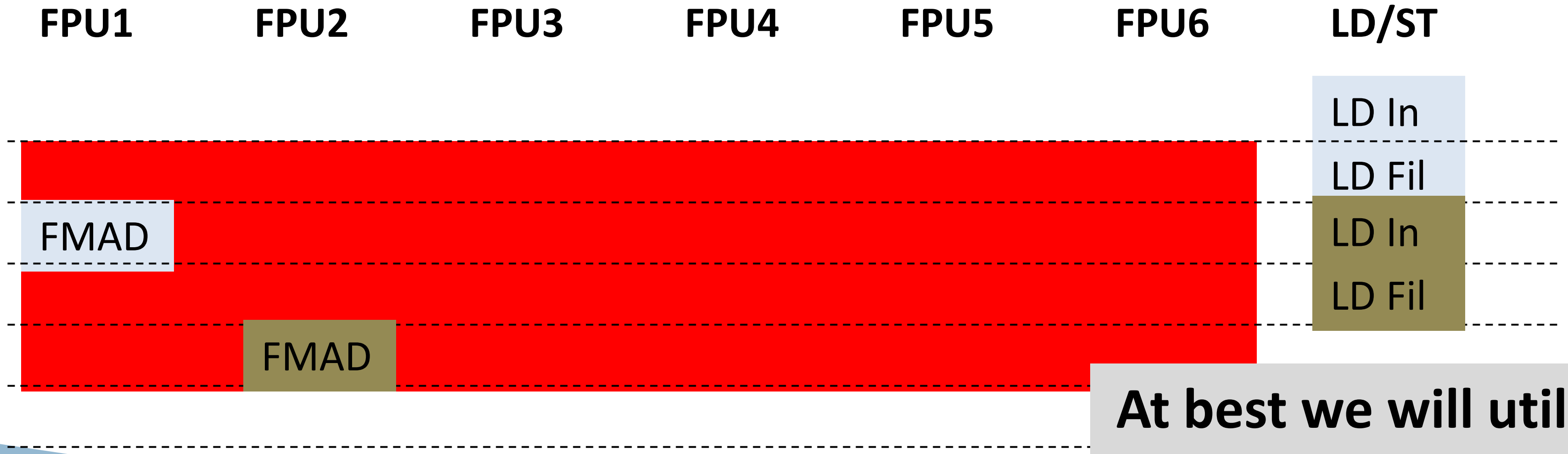
```
For all image pixels  ———→ Map to each thread
    For all filter pixels ——→ For loop for each thread
        output += In(x,y)*Fil(x,y)
```

Everything in global memory

```
For all image pixels            → Map to each thread
    For all filter pixels       → For loop for each thread
        output += In(x,y)*Fil(x,y)
```

GK 110: 6 FMADs per LD/ST

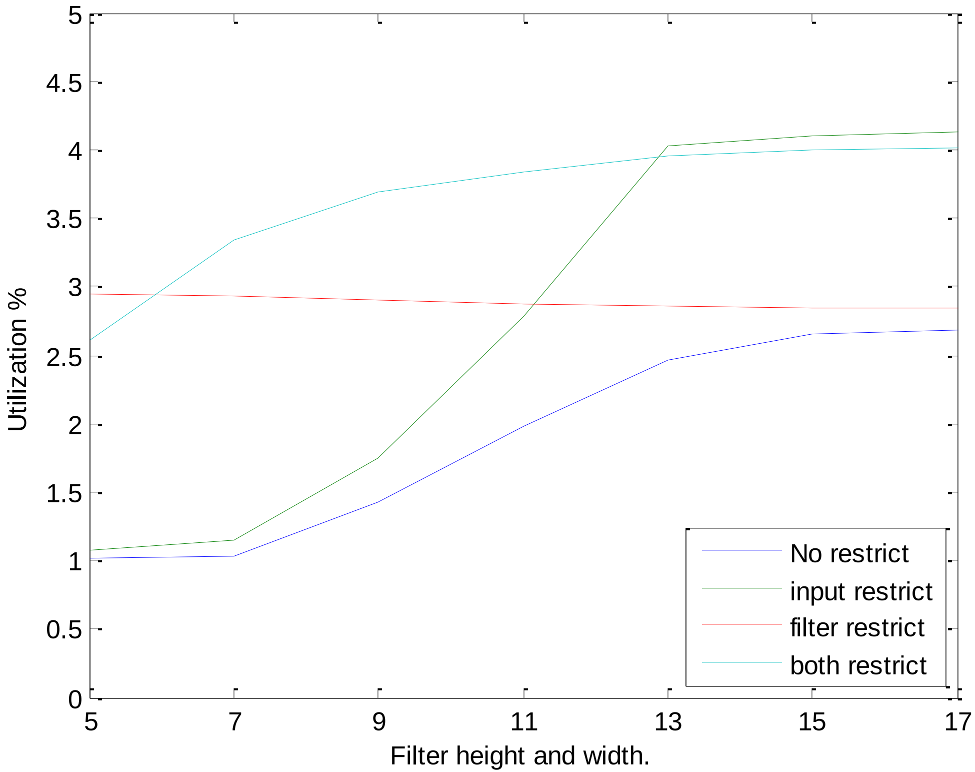
FMAD LD LD → 2 LD per FMAD



At best we will utilize 1/12 of the hardware

Everything in global memory

Roughly 4% utilization



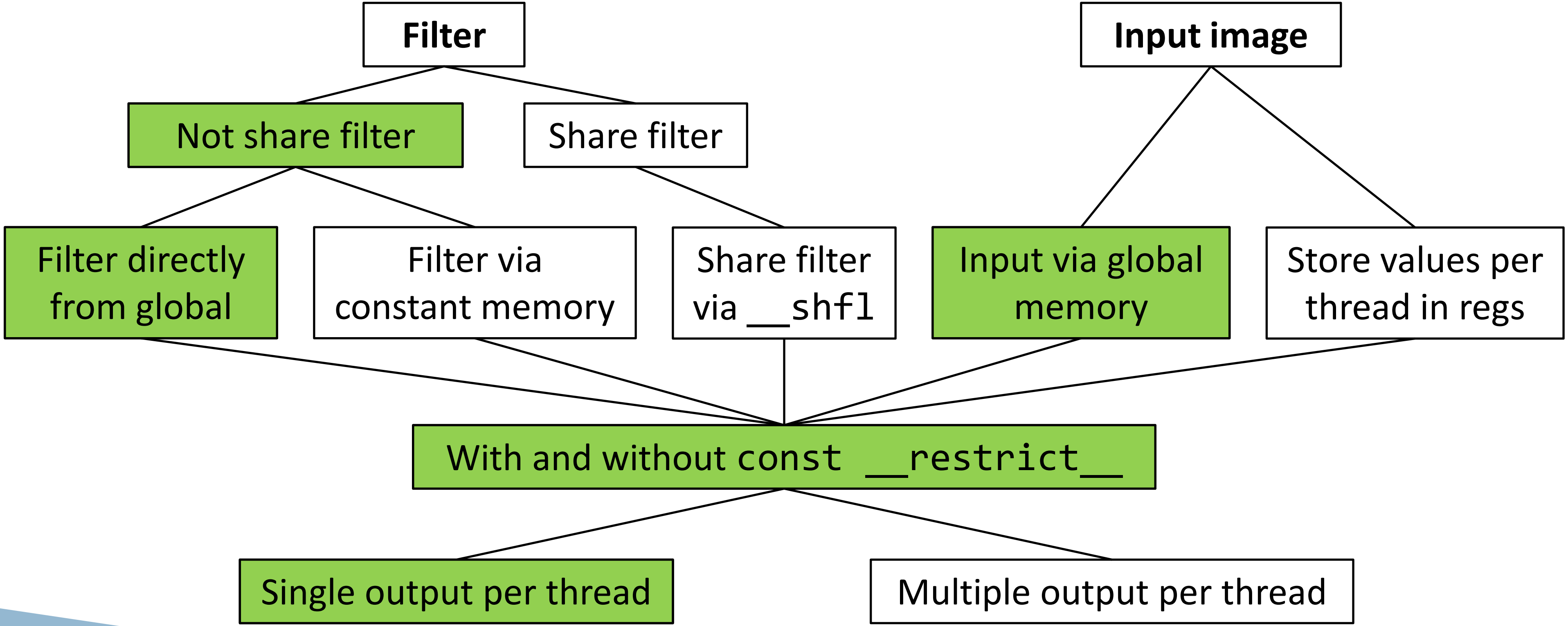
Input restrict

Both restrict

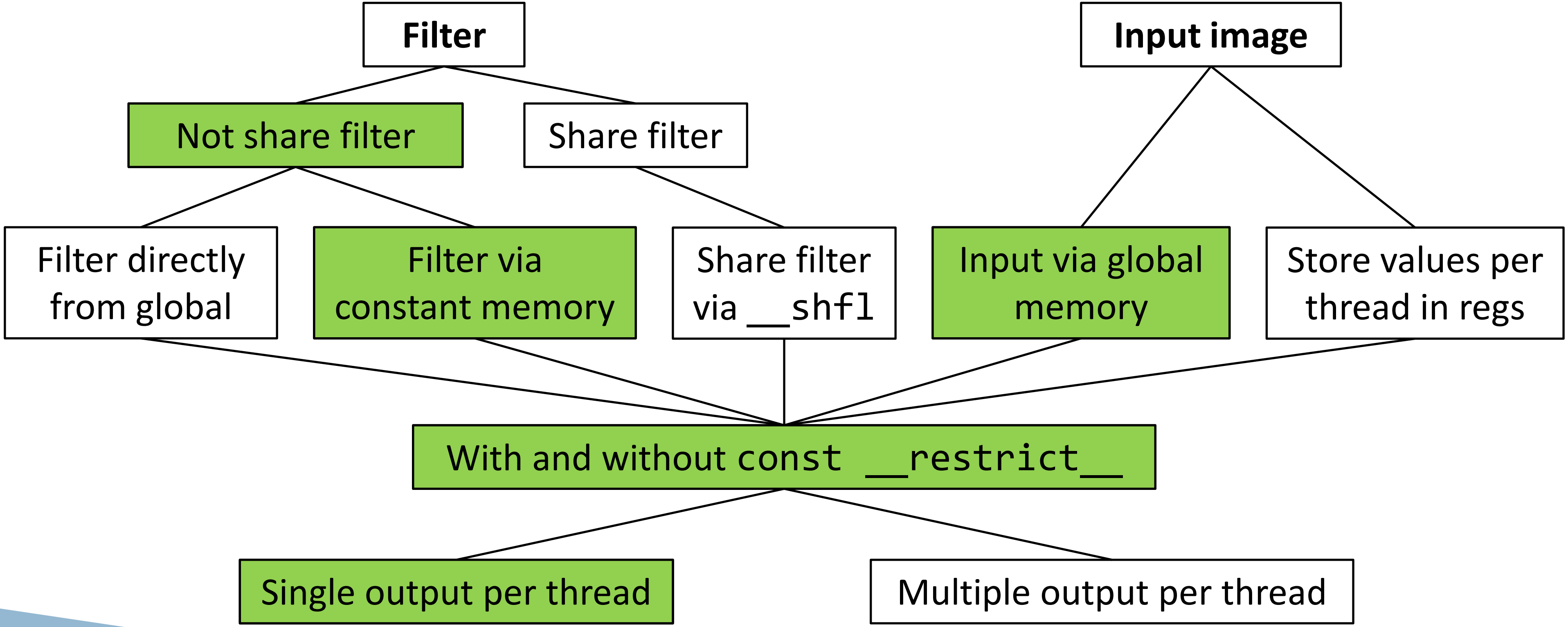
Filter restrict

No restrict

The 2D Convolutions Implemented

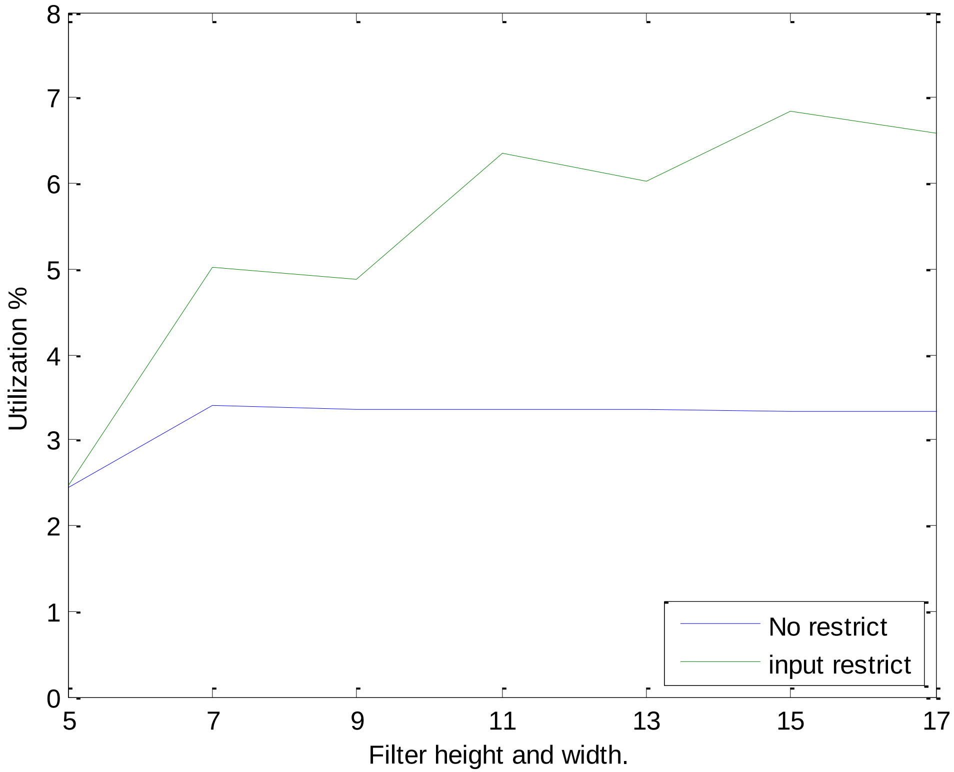


The 2D Convolutions Implemented



Filter in constant, image in global

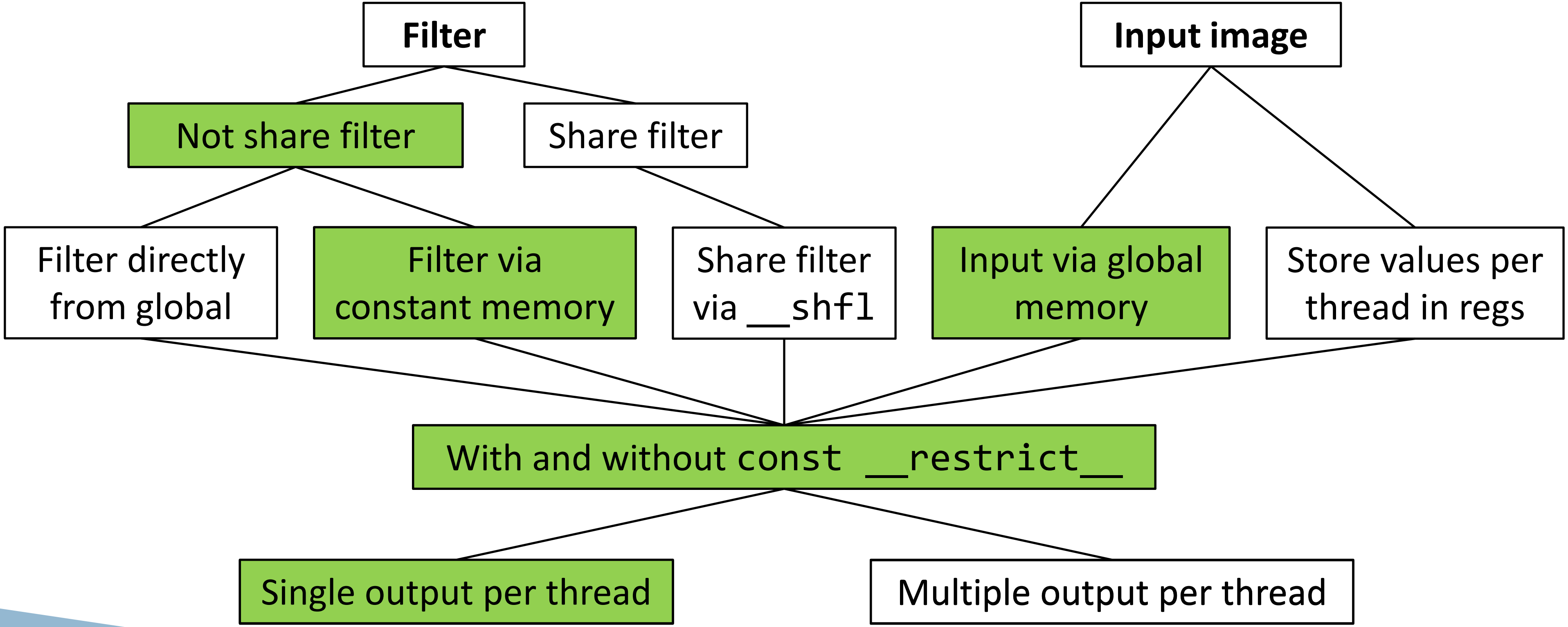
Roughly 7% utilization



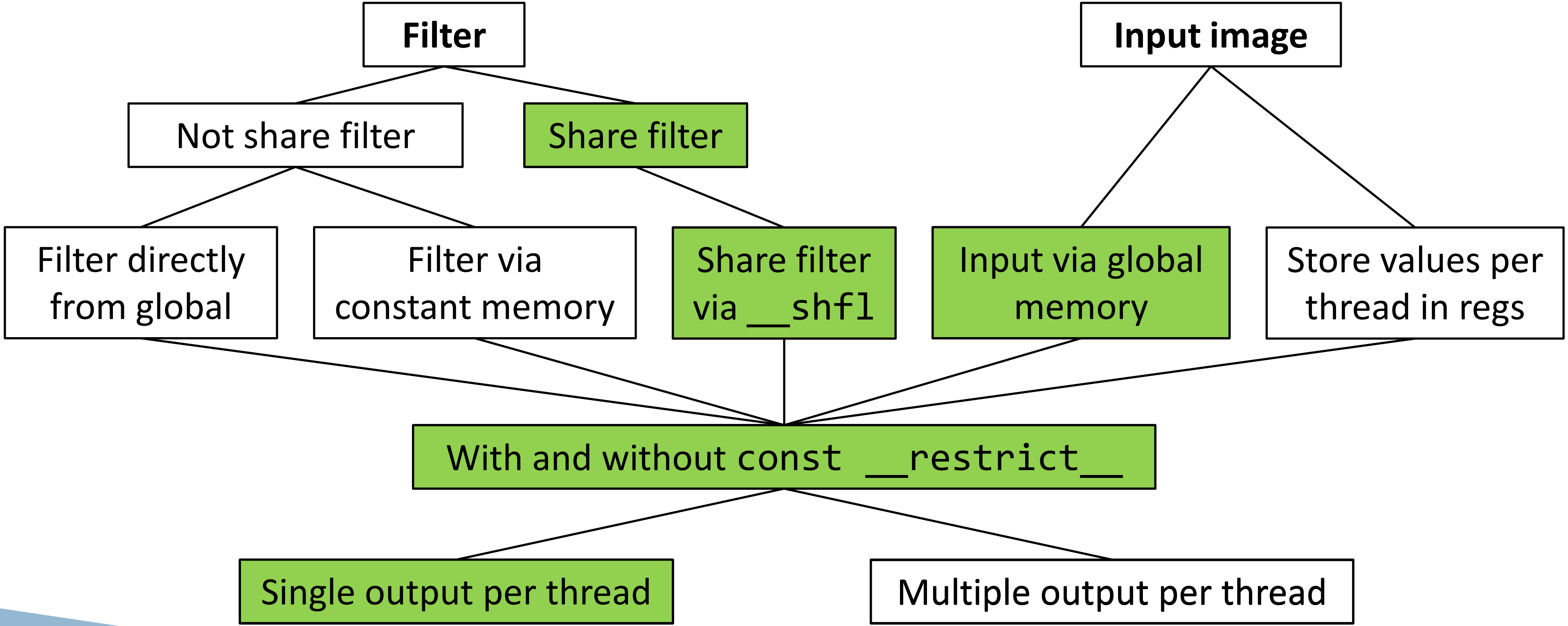
Input restrict

No restrict

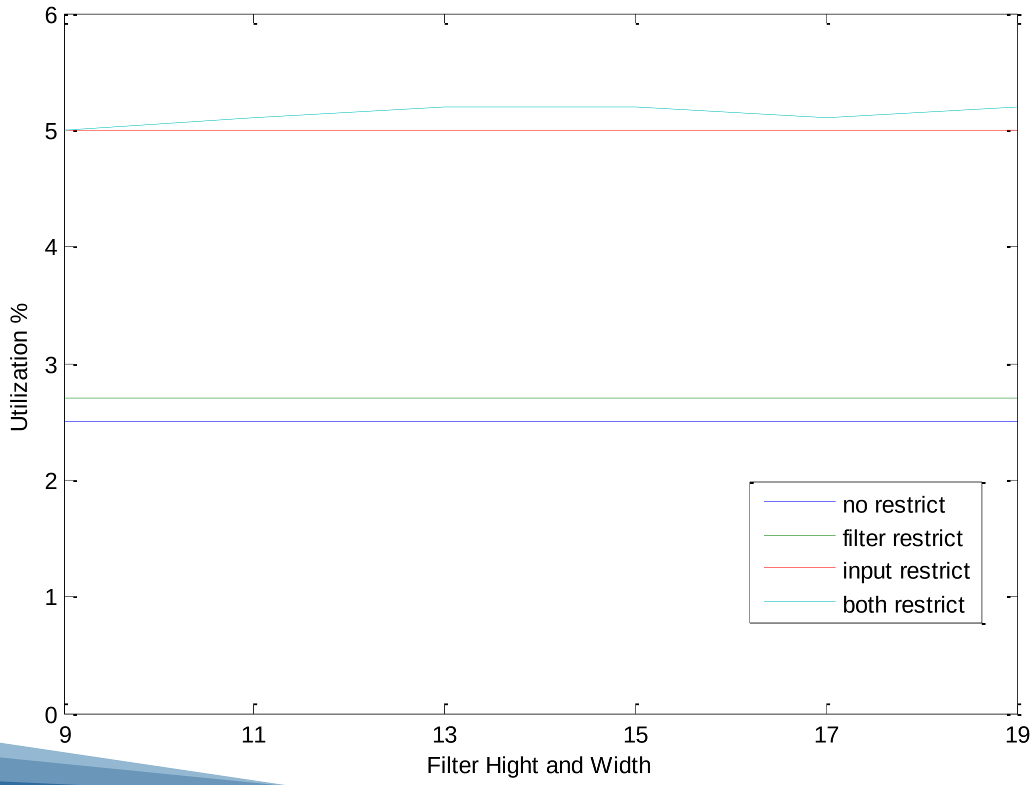
The 2D Convolutions Implemented



The 2D Convolutions Implemented



Share filter via __shfl, image in global



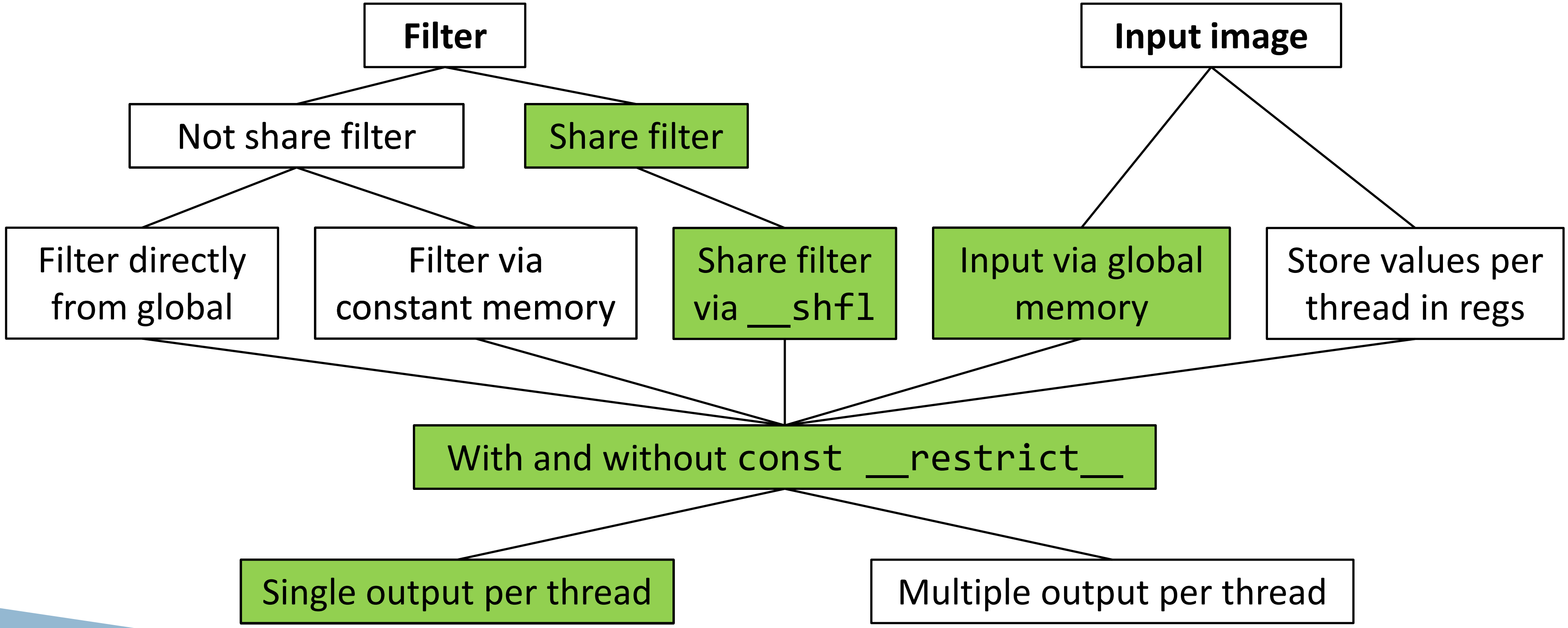
Input restrict

Both restrict

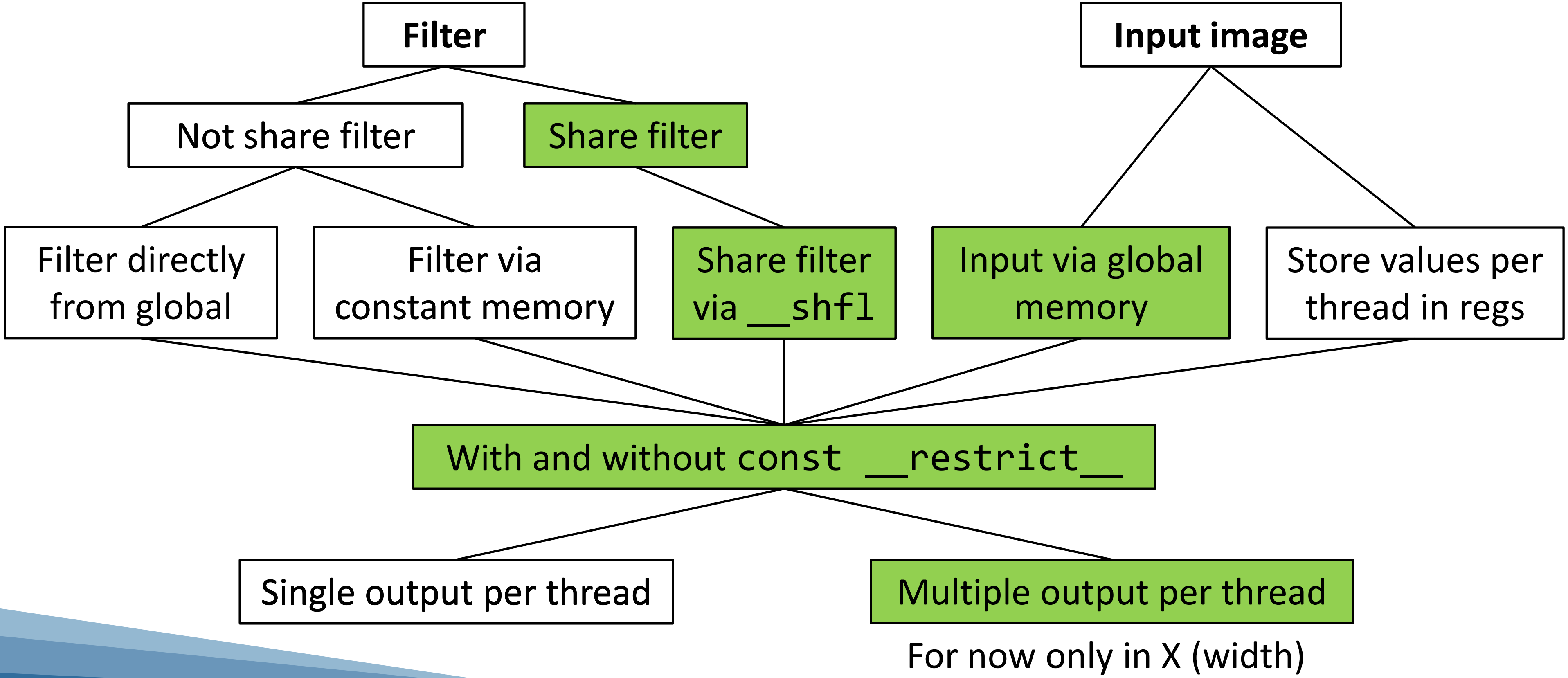
Filter restrict

No restrict

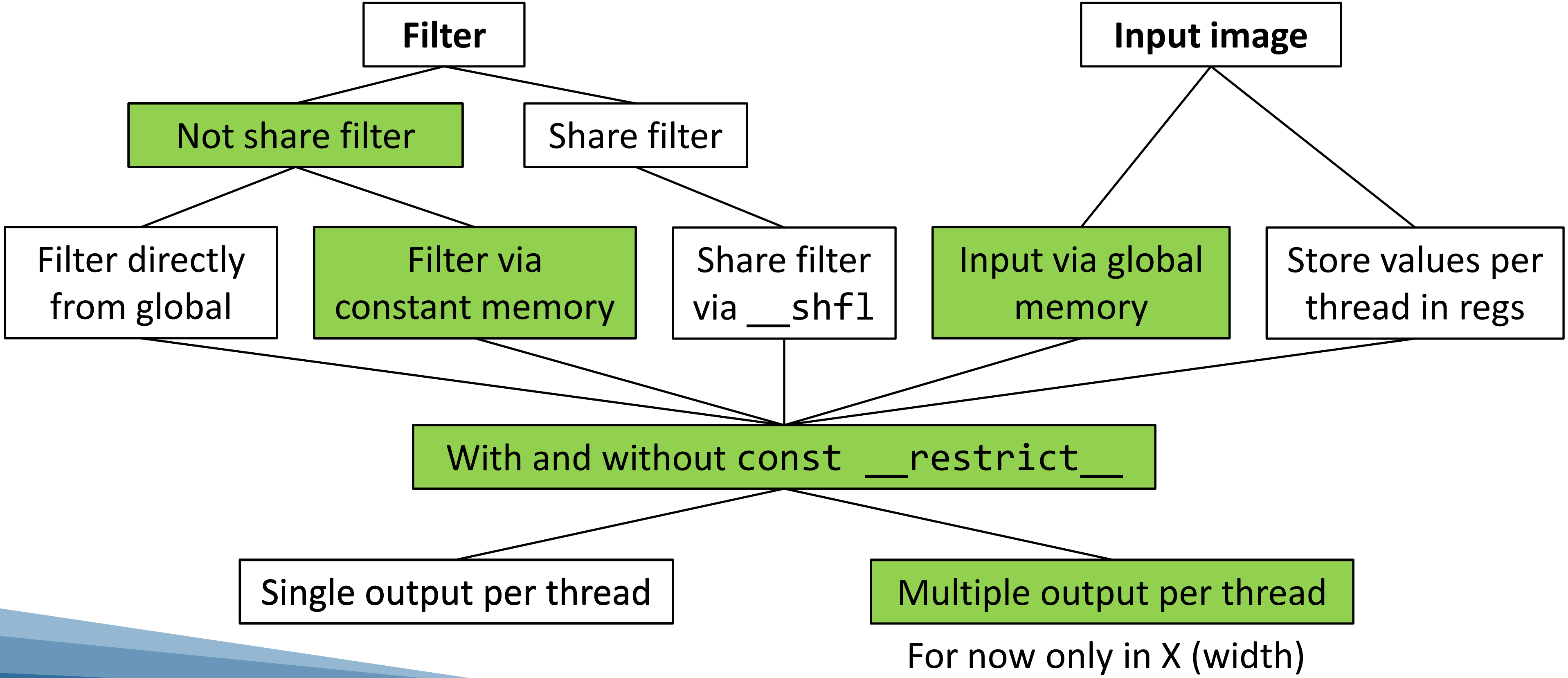
The 2D Convolutions Implemented



The 2D Convolutions Implemented



The 2D Convolutions Implemented



Multiple outputs per thread

```
For all image pixels          ──────→ Map to each thread
    For all filter elements   ──────→ For loop for each thread
        output += In(x.y)*Fil(x,y)
```

```
For all image pixels          ──────────────→ Map one thread to several outputs
    For all filter elements   ──────────────→ For loop for each thread
        fil = Fil(x,y)
        output0 += In(x+0,y)*fil
        output1 += In(x+1,y)*fil
        output2 += In(x+2,y)*fil ...
```

Multiple outputs per thread

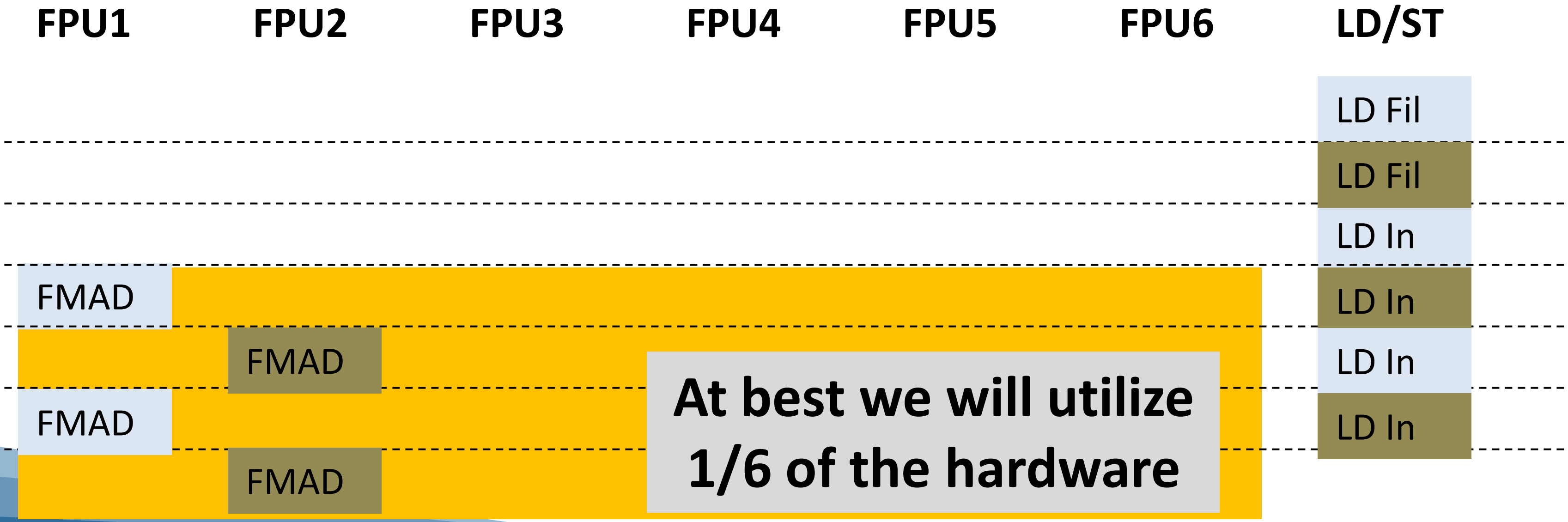
```
For all image pixels
    For all filter elements
        fil = Fil(x,y)
        output0 += In(x+0,y)*fil
        output1 += In(x+1,y)*fil
        output2 += In(x+2,y)*fil ...
```

→

GK 110: 6 FMADs per LD/ST

Place filter value in register, i.e. only 1 LD

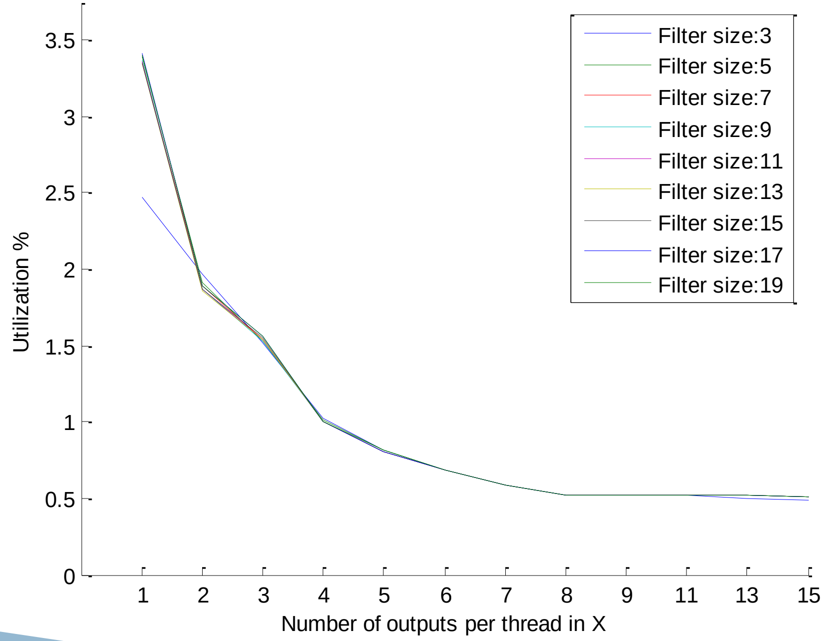
1 LD per FMAD



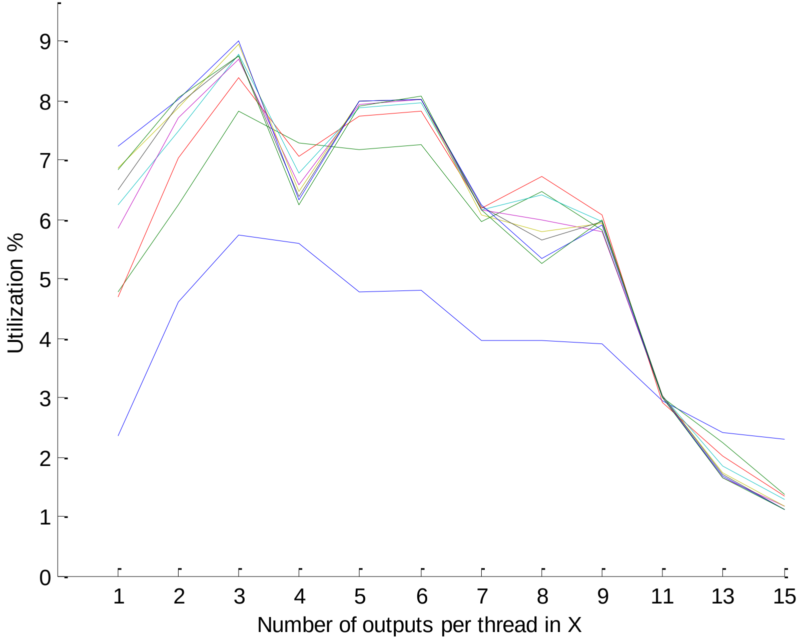
Filter in constant, image in global, multiple outputs per thread

Still no reuse of input data

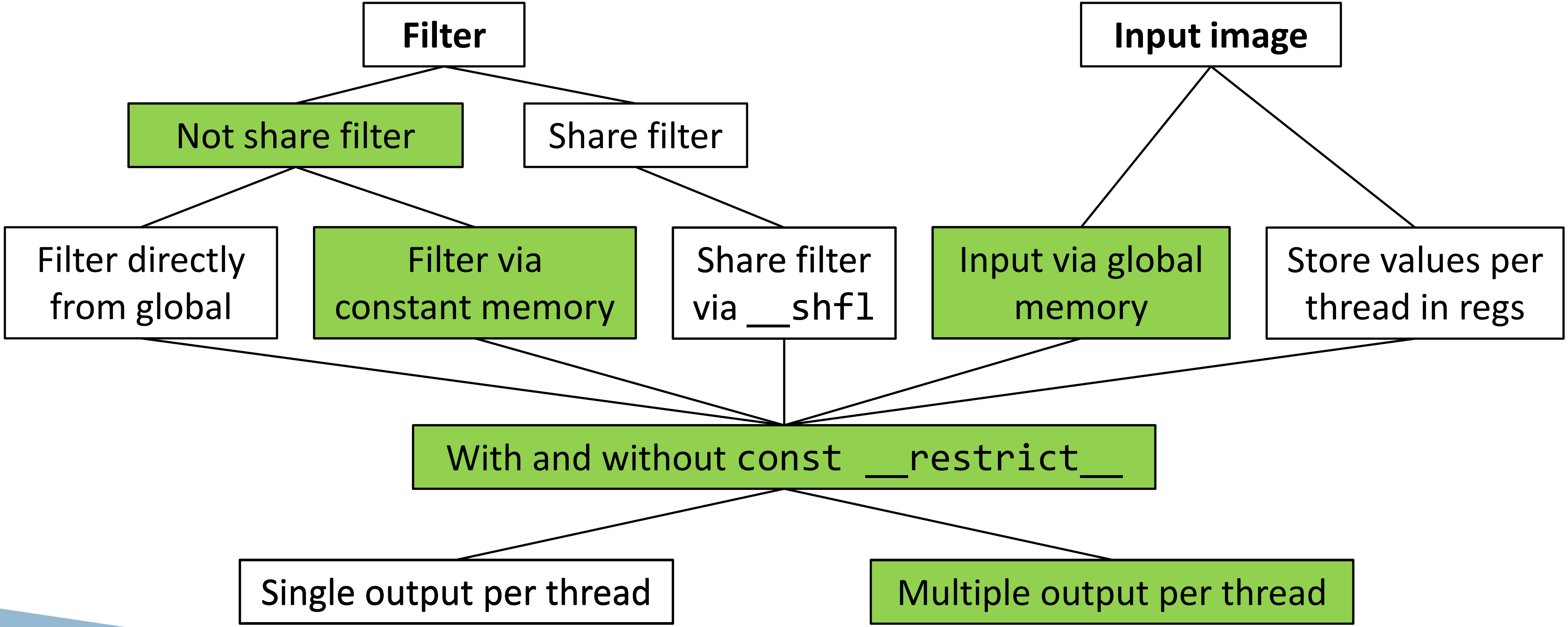
No restrict



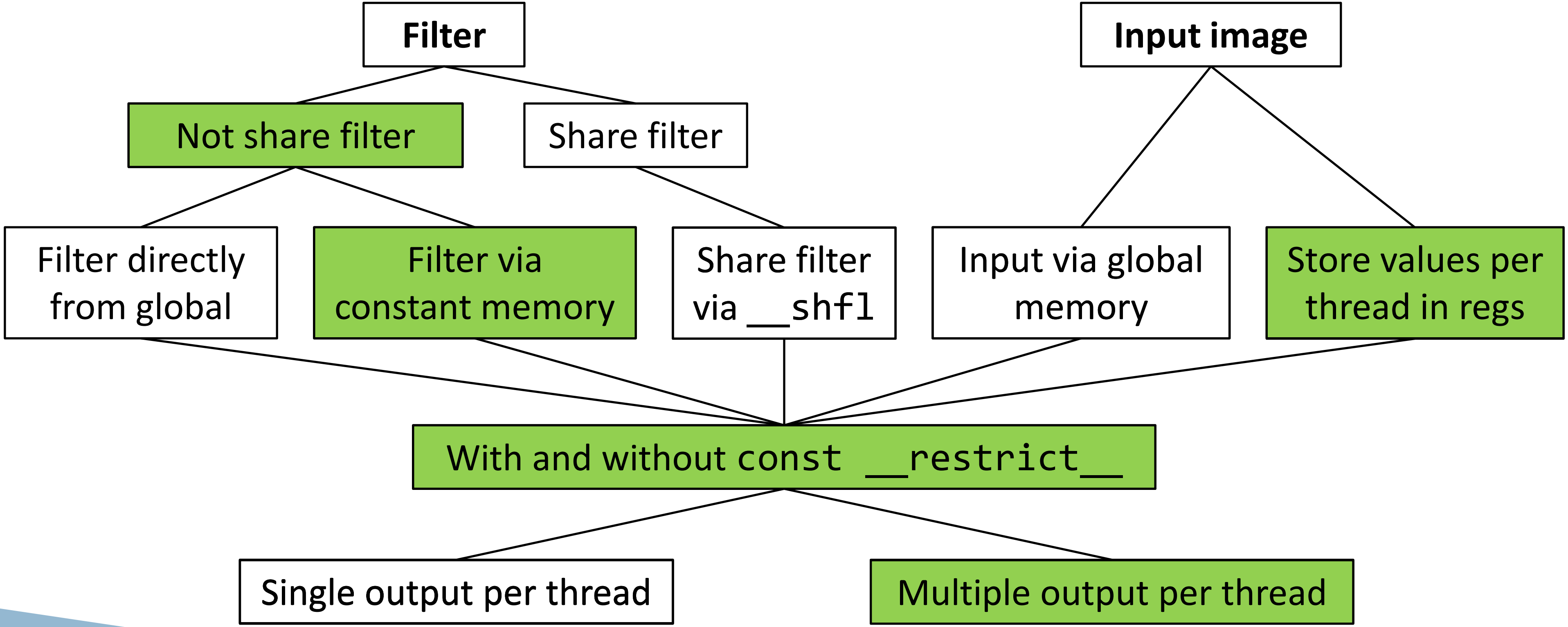
Input restrict



The 2D Convolutions Implemented

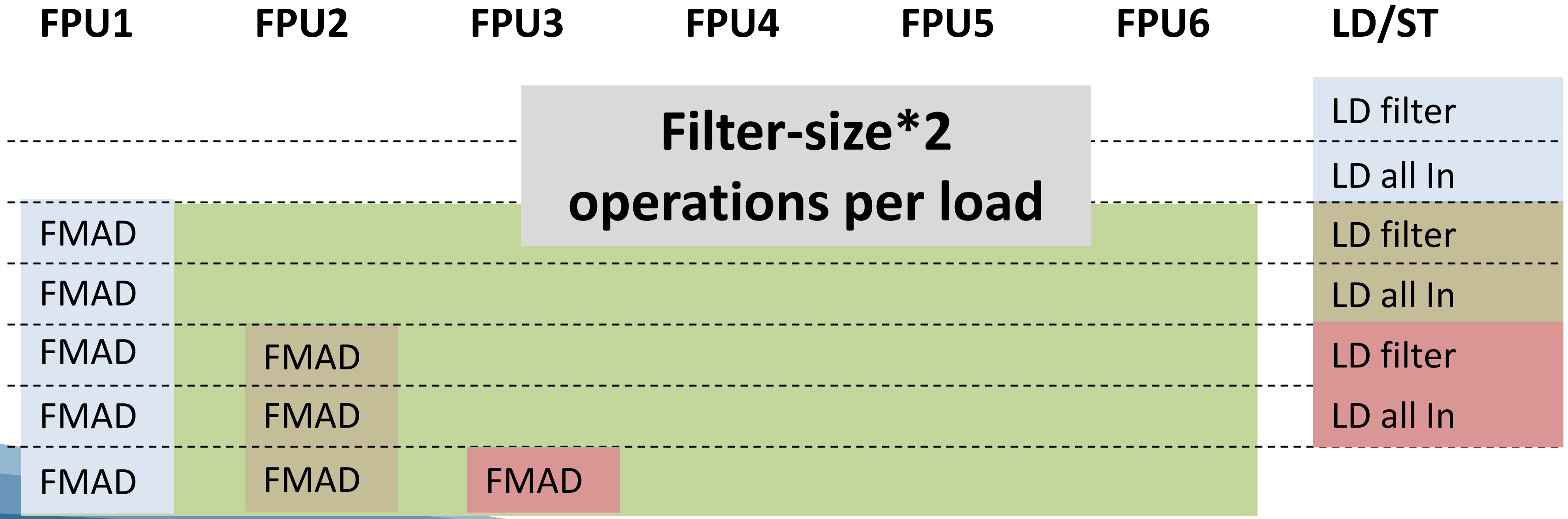


The 2D Convolutions Implemented



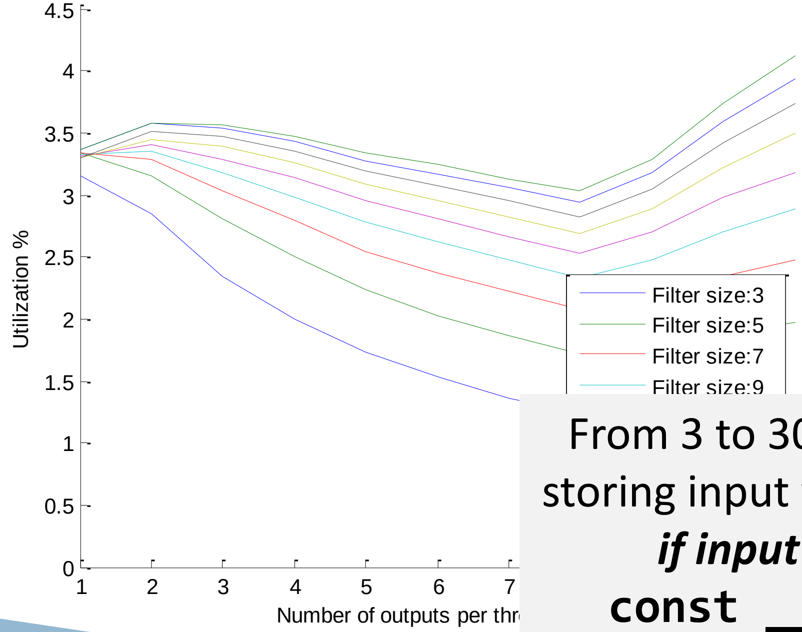
```
For all image pixels
    For all filter elements
        fil0 = Fil(x+0,y);
        in0 = In(x+0,y); in1 = In(x+1,y)...
        output0 += in0*fil0
        output1 += in1*fil0
        output2 += in2*fil0 ...
```

Load filter into register.
Load all input elements into register.
Do all FMAs.

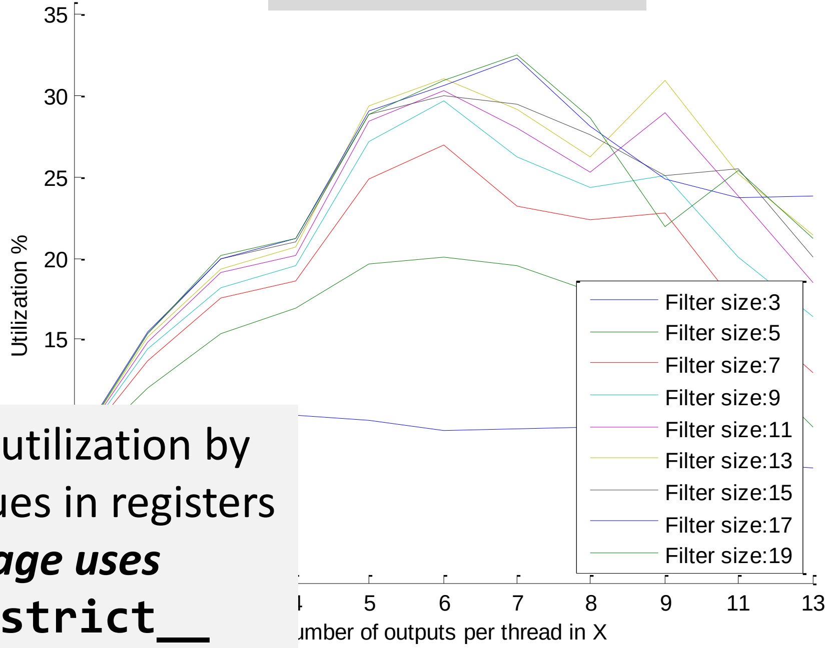


Filter via `const`, reusing thread local input, multiple outputs per thread

No restrict



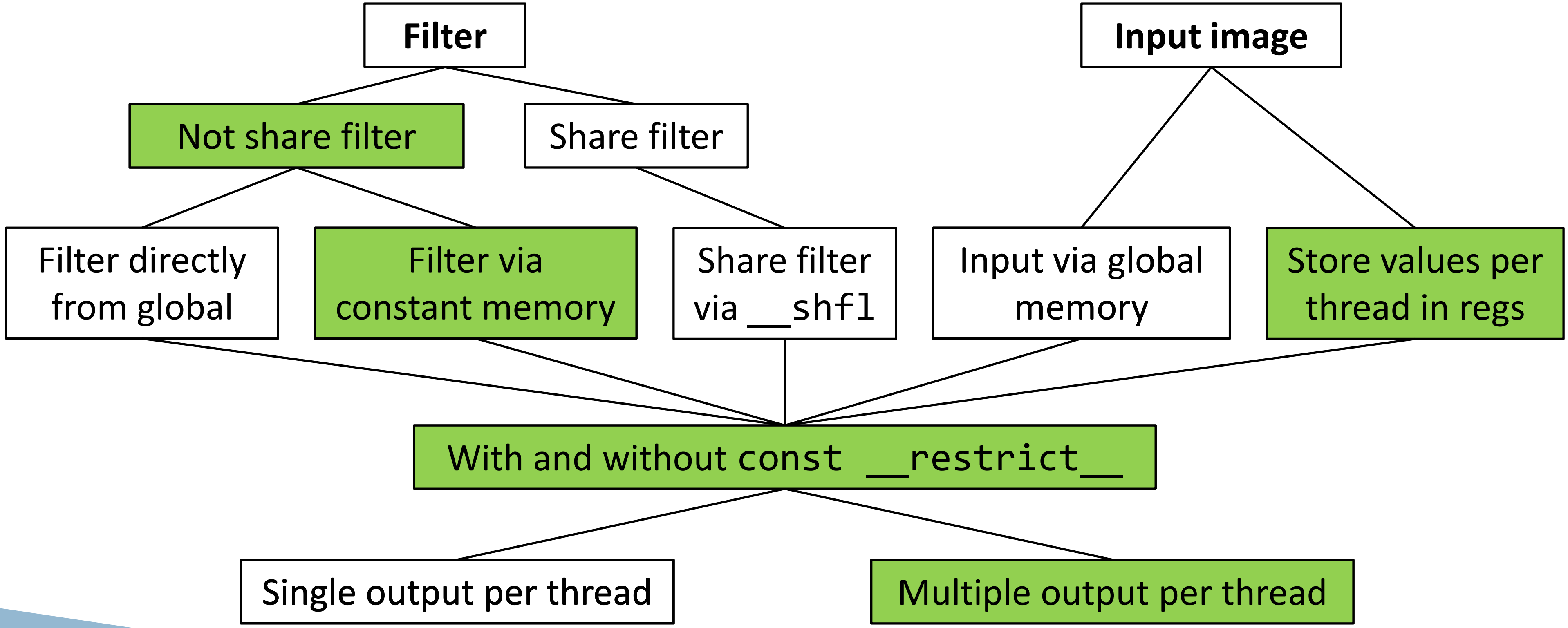
Input restrict



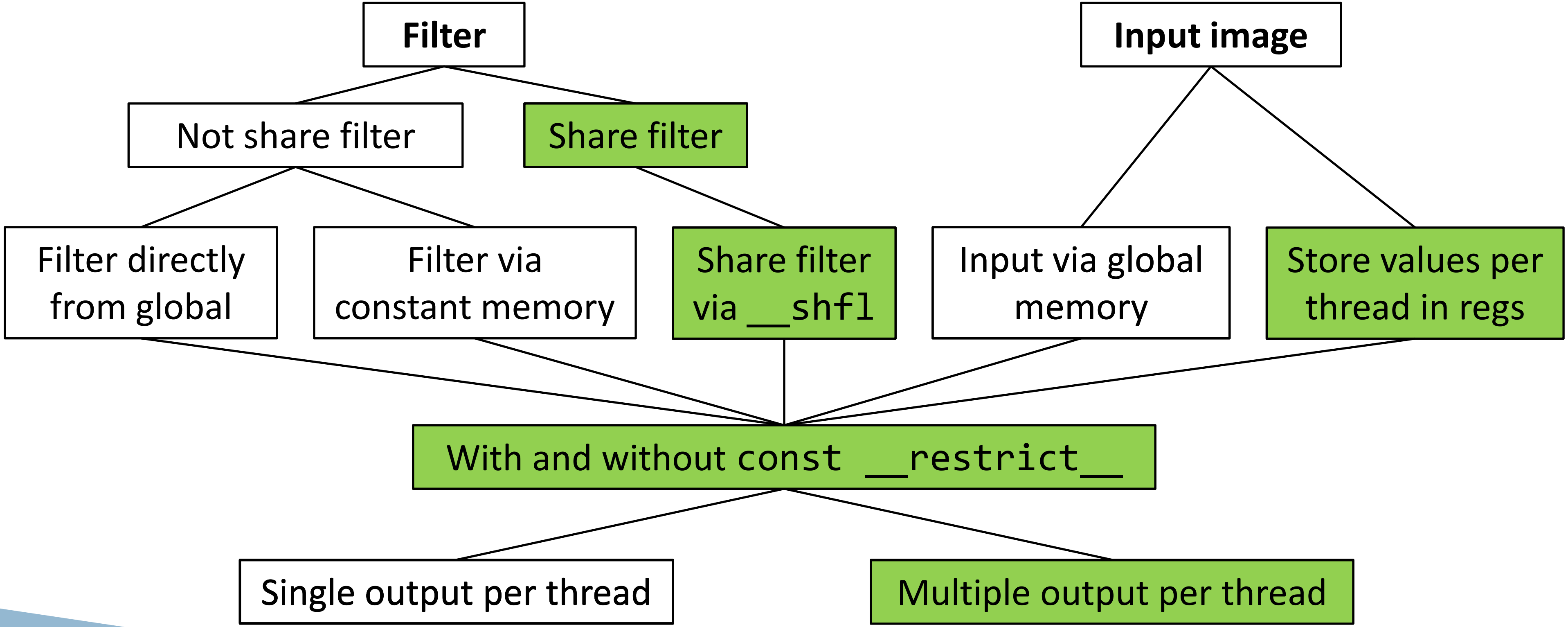
From 3 to 30 % utilization by storing input values in registers ***if input image uses***

`const __restrict__`

The 2D Convolutions Implemented

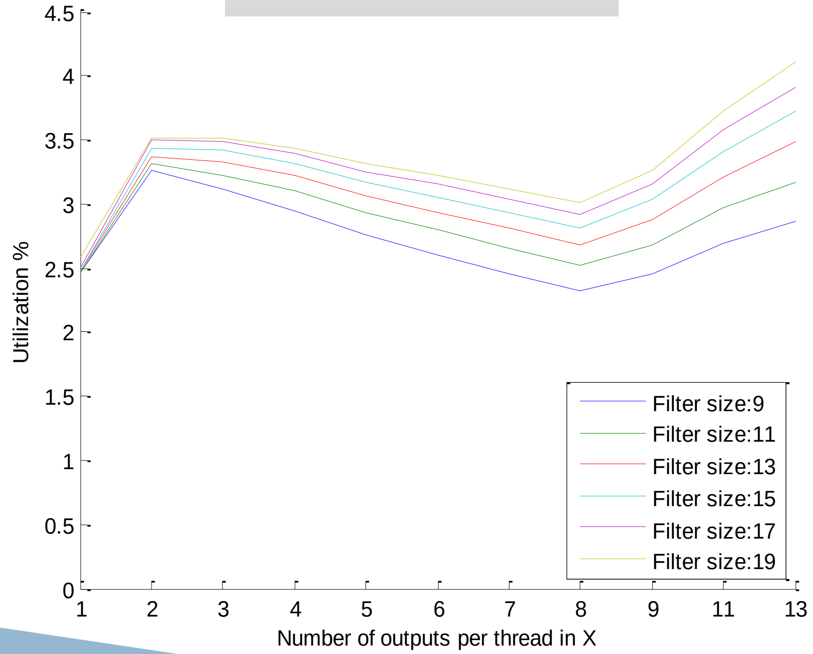


The 2D Convolutions Implemented

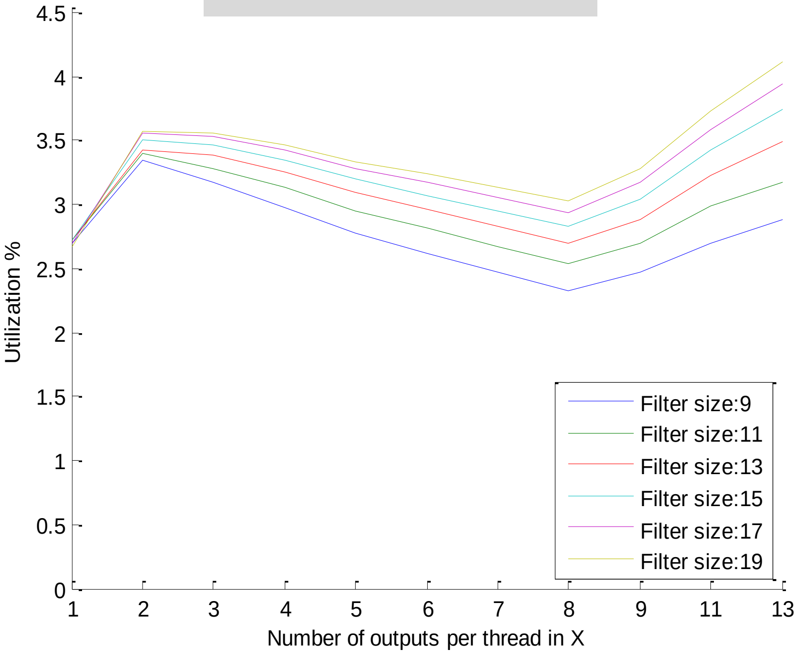


Sharing filter via shfl, reusing thread local input, multiple outputs per thread

No restrict

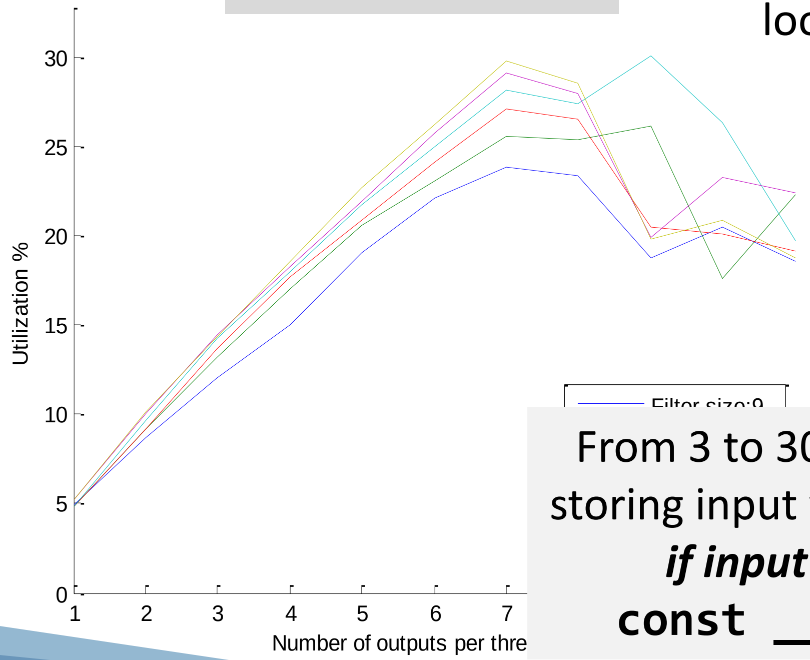


Filter restrict



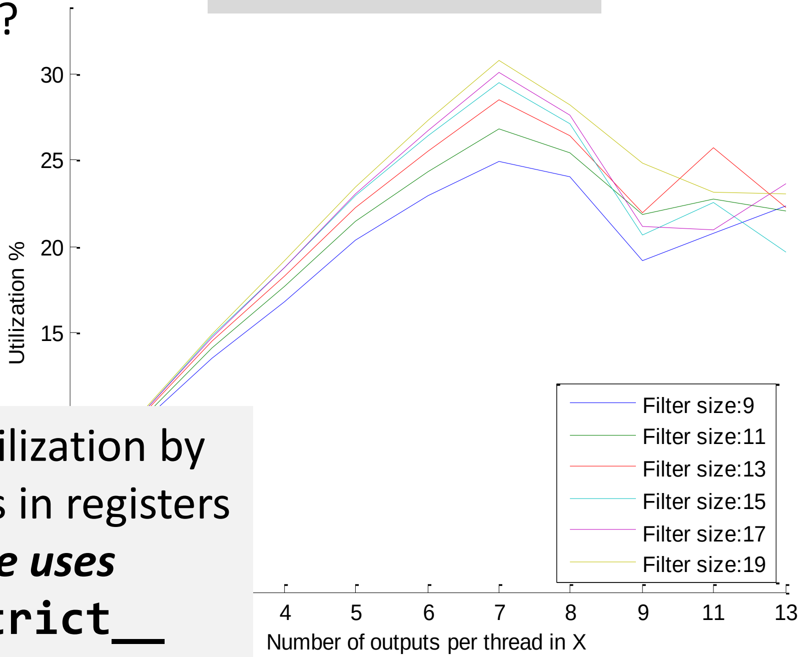
Sharing filter via shfl, reusing thread local input, multiple outputs per thread

Input restrict



What does the code look like?

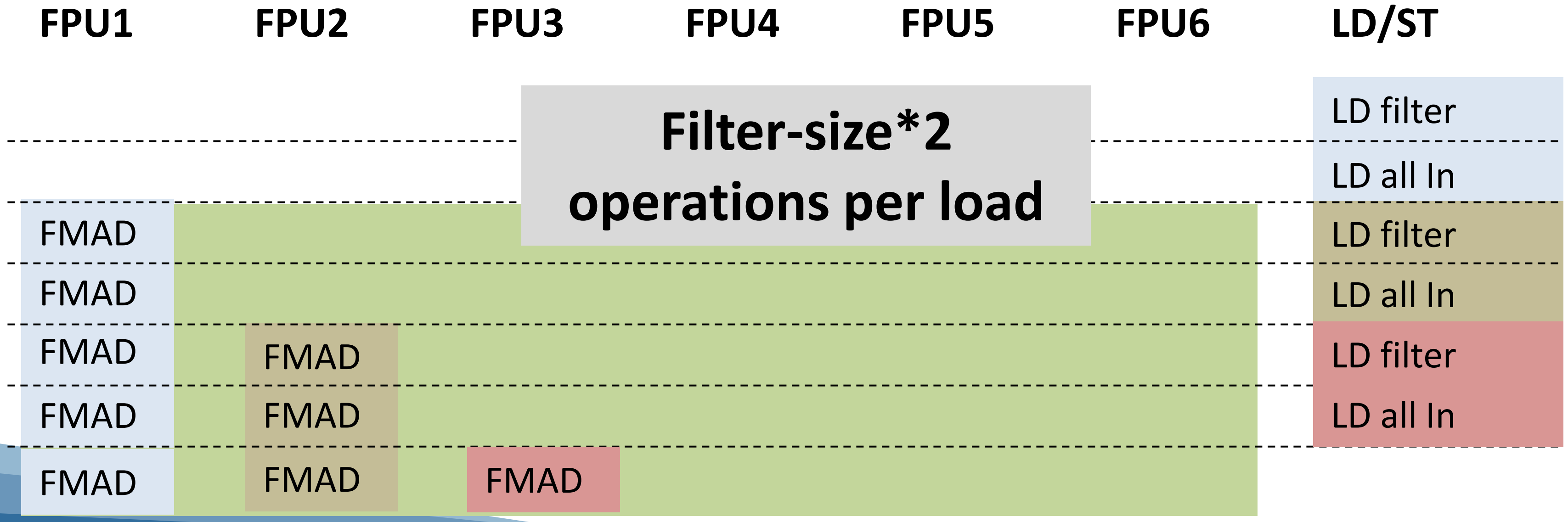
Both restrict



From 3 to 30 % utilization by storing input values in registers ***if input image uses*** **`const __restrict__`**

```
For all image pixels
    For all filter elements
        fil0 = Fil(x+0,y);
        in0 = In(x+0,y); in1 = In(x+1,y)...
        output0 += in0*fil0
        output1 += in1*fil0
        output2 += in2*fil0 ...
```

Load filter into register.
Load all input elements into register.
Do all FMAs.



4 lines of code for reading input into register
9 lines of code for filter loop (shown below)
9 lines of code for output clean-up
= 22 lines of code. → ***Maintainable***

```
for(i=0; i<T_FILTER_WIDTH; i++) {                                                                       1533312
    if(threadIdx.x < T_FILTER_WIDTH) {                                                                  766656
        the_warps_filter_coefs = filter_coefs[threadIdx.x + i*T_FILTER_WIDTH];                          1916640
    }

    const int input_data_float_offset = line_width * y_input_pos + x_block_pos + (i - half_filter_width) * line_width…

    load_data_float<nb_float_input_values>(my_input_values, input_data, input_data_float_offset);

    for(j=0; j<T_FILTER_WIDTH; j++) {
        // Shuffle to get the filter value from thread j;
        present_filter_coef = __shfl(the_warps_filter_coefs, j);

        for(int output_index=0; output_index<T_NB_OUTPUTS_PER_THREAD; output_index++) {
            my_X_output[output_index] += present_filter_coef*my_input_values[output_index + j];         44466048
        }
    }
}
```

No hand-coded assembly
No explicit use of textures
No use of constant memory

Instead:
C++ Templates, `const __restrict__`, `__shfl`

Line	Address	Source	Instruct Execute	Thread Instructio Executed	Threa Execu Efficie	Branche Taken	Branc Efficie	Memory Type	Mem Acces Type	Memor Access Size	L1 Ab Idl Tr	L1 Tr Ov	L2 Tr Ov
179	11D3A0	FFMA.FTZ R34, R35, R12, R34;	383328	12266496	100.0								
180	11D3A8	FFMA.FTZ R35, R33, R11, R20;	383328	12266496	100.0								
181	11D3B0	FFMA.FTZ R19, R33, R15, R30;	383328	12266496	100.0								
182	11D3B8	FFMA.FTZ R39, R41, R12, R35;	383328	12266496	100.0								
183	11D3C8	FFMA.FTZ R18, R33, R7, R18;	383328	12266496	100.0								
184	11D3D0	FFMA.FTZ R37, R41, R10, R19;	383328	12266496	100.0								
185	11D3D8	FFMA.FTZ R22, R33, R10, R22;	383328	12266496	100.0								
186	11D3E0	FFMA.FTZ R21, R33, R8, R21;	383328	12266496	100.0								
187	11D3E8	FFMA.FTZ R32, R33, R12, R32;	383328	12266496	100.0								
188	11D3F0	FFMA.FTZ R21, R41, R11, R21;	383328	12266496	100.0								
189	11D3F8	FFMA.FTZ R20, R33, R14, R34;	383328	12266496	100.0								
190	11D408	FFMA.FTZ R34, R41, R9, R36;	383328	12266496	100.0								
191	11D410	FFMA.FTZ R36, R41, R6, R31;	383328	12266496	100.0								
192	11D418	FFMA.FTZ R31, R41, R15, R18;	383328	12266496	100.0								
193	11D420	FFMA.FTZ R18, R41, R8, R22;	383328	12266496	100.0								
194	11D428	FFMA.FTZ R4, R41, R7, R4;	383328	12266496	100.0								
195	11D430	FFMA.FTZ R22, R41, R14, R32;	383328	12266496	100.0								
196	11D438	TEXDEPBAR 0x3;	383328	12266496	100.0								
197	11D448	FFMA.FTZ R30, R33, R16, R38;	383328	12266496	100.0								
198	11D450	SHFL.IDX PT, R33, R0, 0x7, 0x1f;	383328	12266496	100.0								
199	11D458	FFMA.FTZ R20, R41, R16, R20;	383328	12266496	100.0								
200	11D460	FFMA.FTZ R6, R33, R6, R34;	383328	12266496	100.0								
201	11D468	FFMA.FTZ R34, R33, R7, R36;	383328	12266496	100.0								
202	11D470	FFMA.FTZ R36, R33, R10, R31;	383328	12266496	100.0								
203	11D478	FFMA.FTZ R31, R33, R11, R18;	383328	12266496	100.0								
204	11D488	FFMA.FTZ R32, R33, R8, R37;	383328	12266496	100.0								
205	11D4[illegible]0	FFMA.FTZ R37, R33, R14, R39;	383328	12266496	100.0								
206	11D498	FFMA.FTZ R32, R3, R11, R32;	383328	12266496	100.0								
207	11D4A0	TEXDEPBAR 0x2;	383328	12266496	100.0								
208	11D4A8	FFMA.FTZ R30, R41, R13, R30;	383328	12266496	100.0								
209	11D4B[illegible]	FFMA.FTZ R21, R33, R12, R21;	383328	12266496	100.0								
210	11D4B8	FFMA.FTZ R22, R33, R16, R22;	383328	12266496	100.0								
211	11D4C8	FFMA.FTZ R38, R33, R15, R4;	383328	12266496	100.0								
212	11D4D0	MOV R4, RZ;	383328	12266496	100.0								
213	11D4D8	FFMA.FTZ R40, R33, R13, R20;	383328	12266496	100.0								
214	11D4E0	FFMA.FTZ R20, R3, R12, R31;	383328	12266496	100.0								
215	11D4E8	FFMA.FTZ R18, R3, R15, R34;	383328	12266496	100.0								
216	11D4F0	FFMA.FTZ R19, R3, R16, R37;	383328	12266496	100.0								
217	11D4F8	FFMA.FTZ R31, R3, R14, R21;	383328	12266496	100.0								
218	11D508	TEXDEPBAR 0x1;	383328	12266496	100.0								
219	11D510	FFMA.FTZ R39, R33, R5, R30;	383328	12266496	100.0								
220	11D518	FFMA.FTZ R16, R3, R13, R22;	383328	12266496	100.0								
221	11D520	FFMA.FTZ R30, R3, R7, R6;	383328	12266496	100.0								
222	11D528	FFMA.FTZ R34, R3, R10, R38;	383328	12266496	100.0								
223	11D530	FFMA.FTZ R33, R3, R8, R36;	383328	12266496	100.0								
224	11D538	FFMA.FTZ R22, R3, R5, R40;	383328	12266496	100.0								
225	11D548	TEXDEPBAR 0x0;	383328	12266496	100.0								
226	11D550	FFMA.FTZ R21, R3, R2, R39;	383328	12266496	100.0								

~ 100 FMAD

Use compile time in inner loops

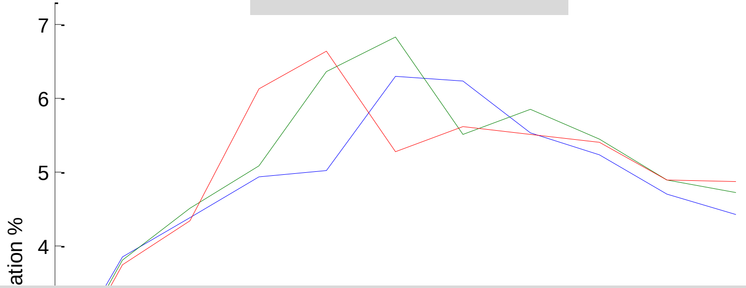
Inner most loop is *run time*

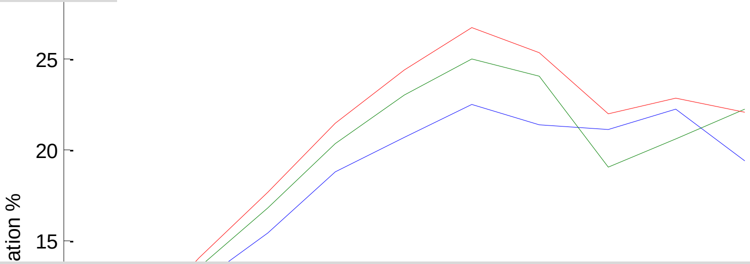
Both restrict

A factor of 3 difference in performance!

Inner most loop is *compile time*

Both restrict






```
for(j=0; j<filtersize_compile_time_constant; j++)
{
    const float present_filter_coef = __shfl(the_warps_filter_coefs, j);

    for(int output_index=0; output_index<T_NB_OUTPUTS_PER_THREAD; output_index++)
    {
        my_X_output[output_index] += present_filter_coef*my_input_values[output_index + j];
    }
}
```

Use compile time in inner loops

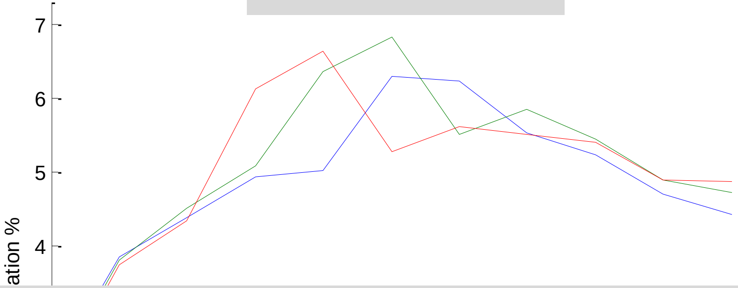
Inner most loop is *run time*

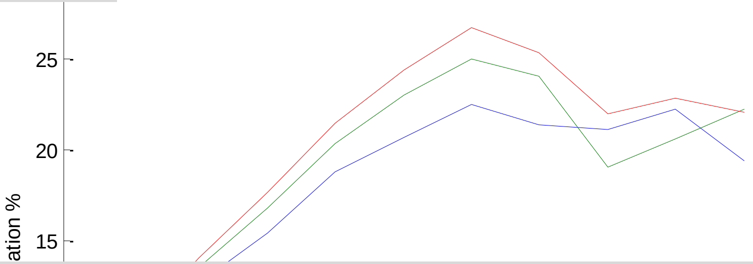
Both restrict

A factor of 3 difference in performance!

Inner most loop is *compile time*

Both restrict






```
for(j=0; j<filtersize_run_time_constant; j++)
{
    const float present_filter_coef = __shfl(the_warps_filter_coefs, j);

    for(int output_index=0; output_index<T_NB_OUTPUTS_PER_THREAD; output_index++)
    {
        my_X_output[output_index] += present_filter_coef*my_input_values[output_index + j];
    }
}
```

2D Convolutions
What are they?

Filter

3	3	1
1	1	2
1	2	3

Image

1 * 3	2 * 3	3 * 1	1
4 * 1	5 * 1	2 * 2	3
4 * 1	1 * 2	1 * 3	5
1	2	5	2

Output

2D Convolutions

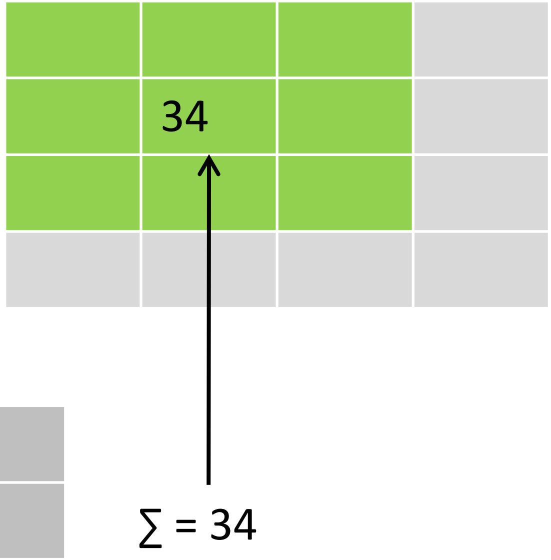
Filter

3	3	1
1	1	2
1	2	3

Image

1 * 3	2 * 3	3 * 1	1
4 * 1	5 * 1	2 * 2	3
4 * 1	1 * 2	1 * 3	5
1	2	5	2

Output



3	6	3
4	5	4
4	2	3

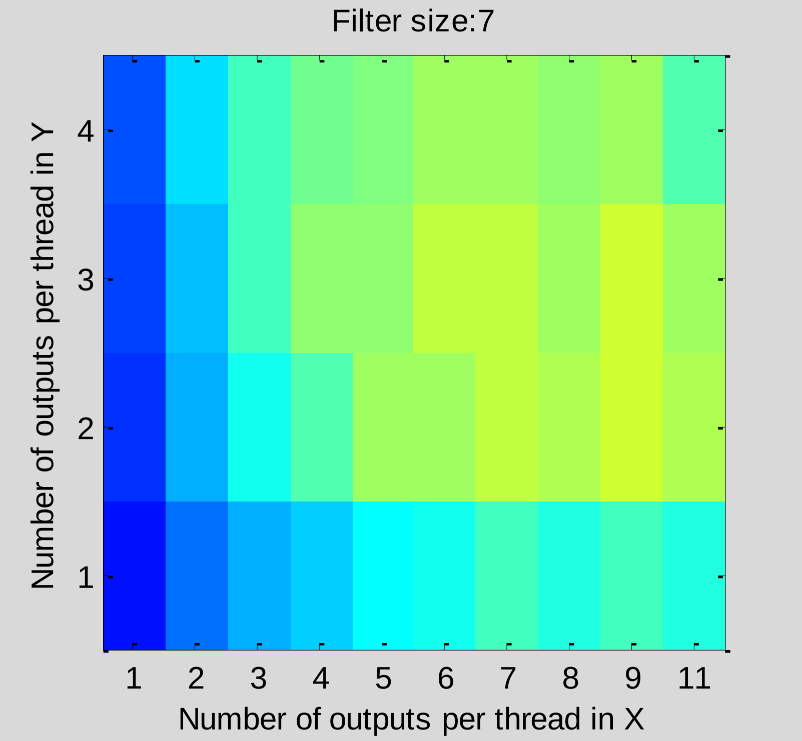
Extend reuse of input data to Y also.

Reuse input in X *and* Y

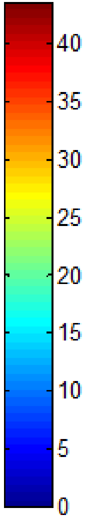
```
For all image pixels         → Map one thread to several outputs
    For all filter pixels    → For loop for each thread
        fil = Fil(x,y)
        output0 += In(x+0,y)*fil
        output1 += In(x+1,y)*fil
        output2 += In(x+2,y)*fil
        output3 += In(x+3,y)*fil
        ...
```

```
For all image pixels         → Map one thread to several outputs in x and y
    For all filter pixels    → For loop for each thread
        fil = Fil(x,y)
        out00 = In(x+0,y+0)*fil
        out01 = In(x+0,y+1)*fil
        out10 = In(x+1,y+0)*fil
        out11 = In(x+1,y+1)*fil
```

Reuse input in X *and* Y

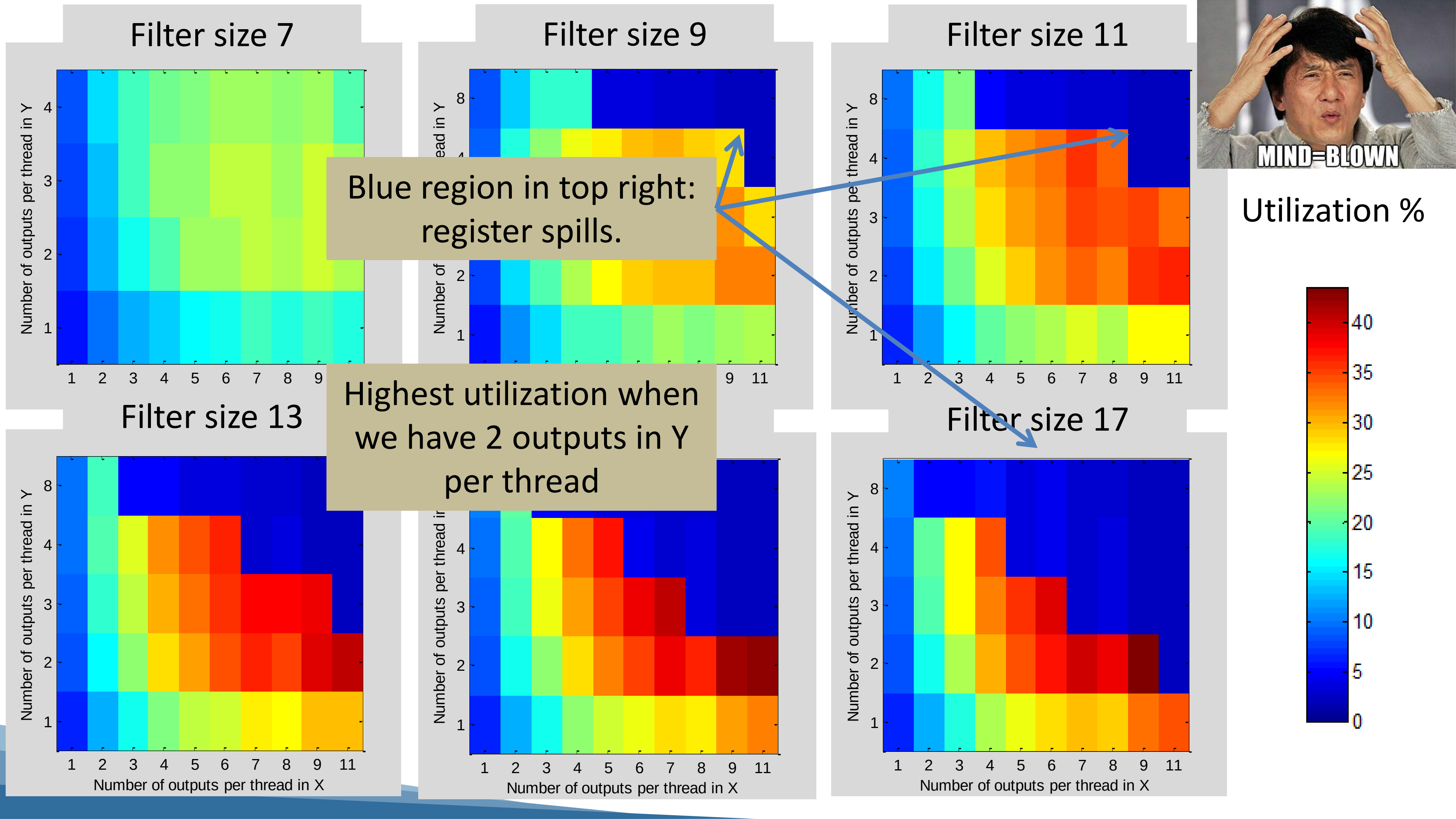


Utilization heat map:
blue is low utilization,
red is high utilization.



Multiple outputs per thread in Y (height)

Multiple outputs per thread in X (width)



Filter size 7
Filter size 9
Filter size 11
Filter size 13
Filter size 17
Number of outputs per thread in Y
Number of outputs per thread in X
1 2 3 4 5 6 7 8 9 11
Blue region in top right: register spills.
Highest utilization when we have 2 outputs in Y per thread
MIND=BLOWN
Utilization %
0 5 10 15 20 25 30 35 40

2D Convolutions Conclusion

DOs:

- Map multiple outputs to each thread.
- Use templates to hardcode loops as non-constant indexed arrays ”[are] likely to [be] placed in local memory”.
- Its helpfull to have a basic understanding and model of the hardware you’re working with.
- Keep looking at you assembly: What lines map to register based compute, and what is LD/ST integer spaghetti code?

DON’Ts:

- Use run-time sizes in inner most loops.
- Use textures or constant memory.

`const __restrict__` gets the job done and its very simple to use!

Any questions?

High Performance Consulting Sweden

ian.wainwright@hpcsweden.se