Ray tracing

ING

Ray tracing pseudo-algorithm

```
for pixel in pixels {
    pixel.setBlack()
    minDistance = Infinity
    indexMin = -1
    for sphere in spheres {
        if (sphere.isBelow(pixel) && sphere.distance(pixel) < minDistance) {
            minDistance = sphere.distance()
            indexMin = sphereIndex
        }
    }
    if (indexMin != -1) {
        pixel.setHue(spheres[indexMin].getHue())
        pixel.setBrightness(spheres[indexMin],getBrightness())
    }
}
```

Time measures definitions

- CPU
  - Init, memory allocation (spheres, image)
  - Compute, fillImageGL
- GPU
  - Init, host memory allocation + host->device copy
  - Compute, kernel fillImageGL
  - GetResult, copy image device->host

Time measures definitions

- CPU
  - Init, memory allocation (spheres, image)
  - Compute, fillImageGL
- GPU
  - Init, host memory allocation + host->device copy
  - Compute, kernel fillImageGL
  - GetResult, copy image device->host

Time complexity

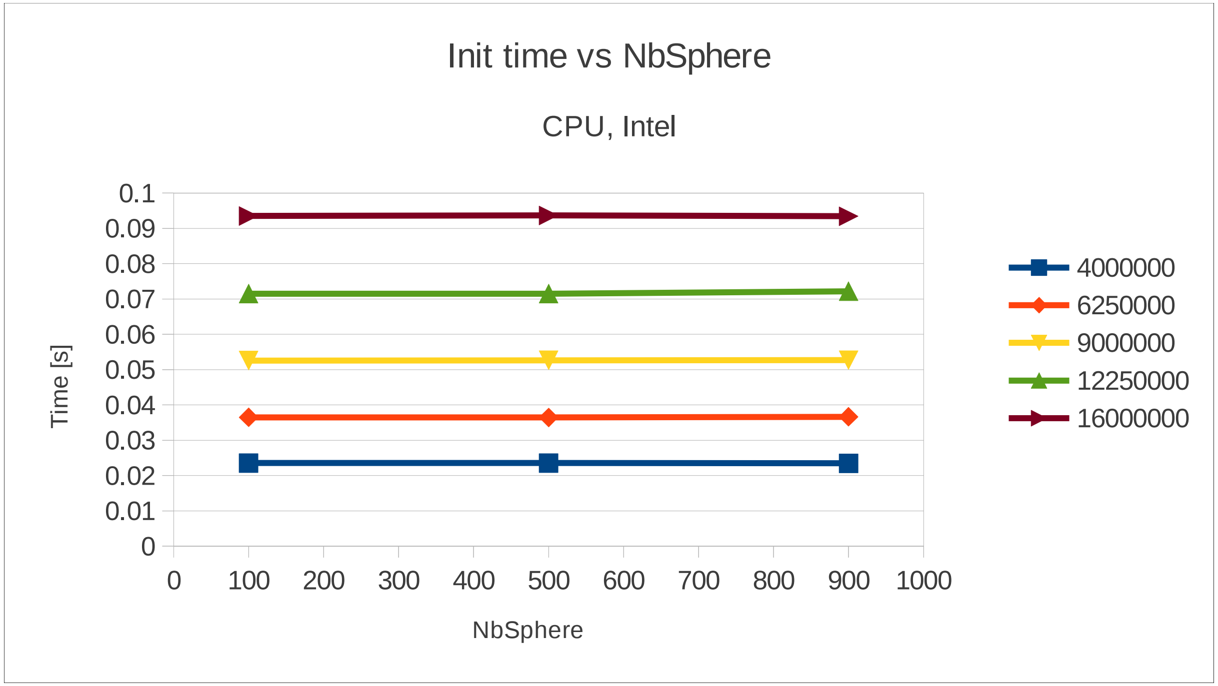
- Program version used
  - CPU
  - Intel compiler
  - OMP 1 thread => 1 CPU core

- Init complexity
- Compute complexity

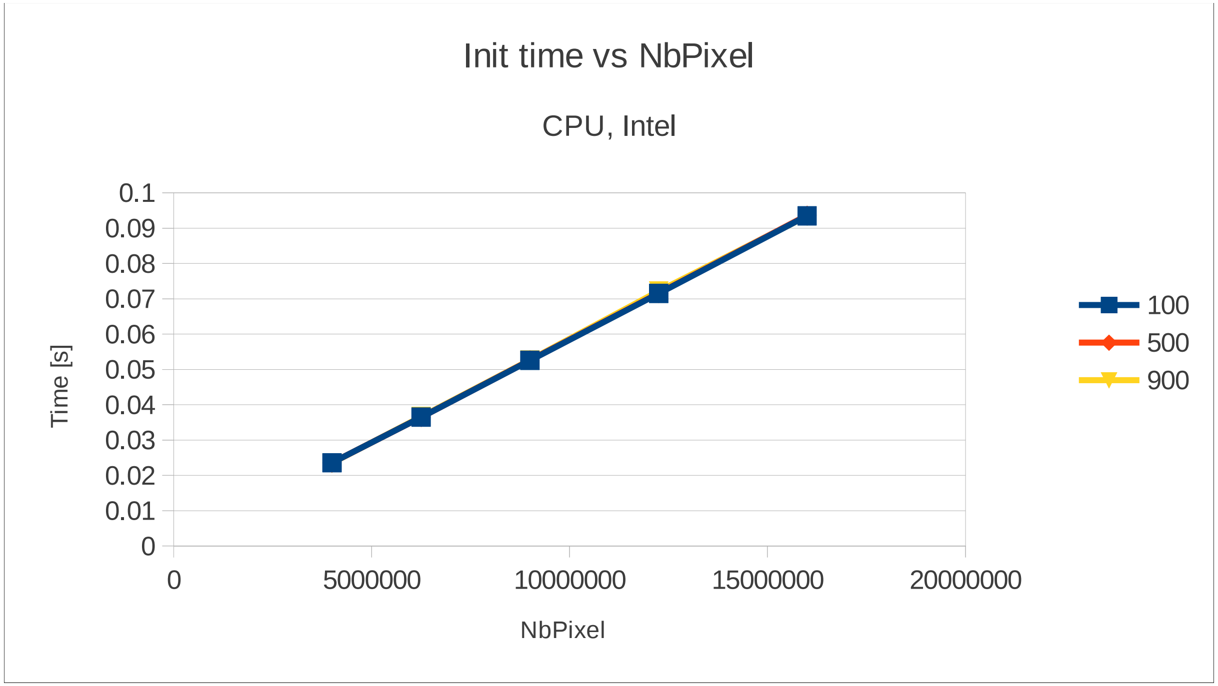
Ray tracing reminder

- Two size parameters
  - Number of spheres
  - Number of pixels

- How the two parameters influence time complexity ?



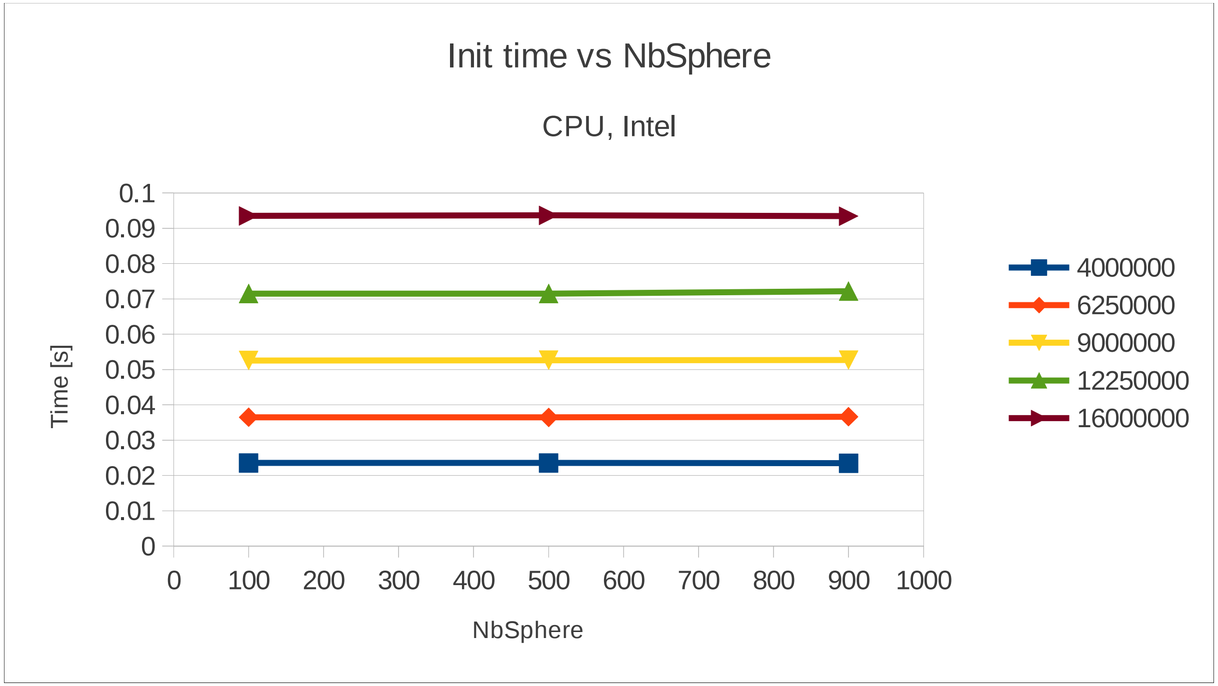
Init time vs NbSphere
CPU, Intel
Time [s]
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
100
200
300
400
500
600
700
800
900
1000
NbSphere
4000000
6250000
9000000
12250000
16000000



Init time vs NbPixel
CPU, Intel
Time [s]
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
5000000
10000000
15000000
20000000
NbPixel
100
500
900

Init time complexity discussion

- Why init time complexity seems to be constant, *O*(1), versus number of sphere ?

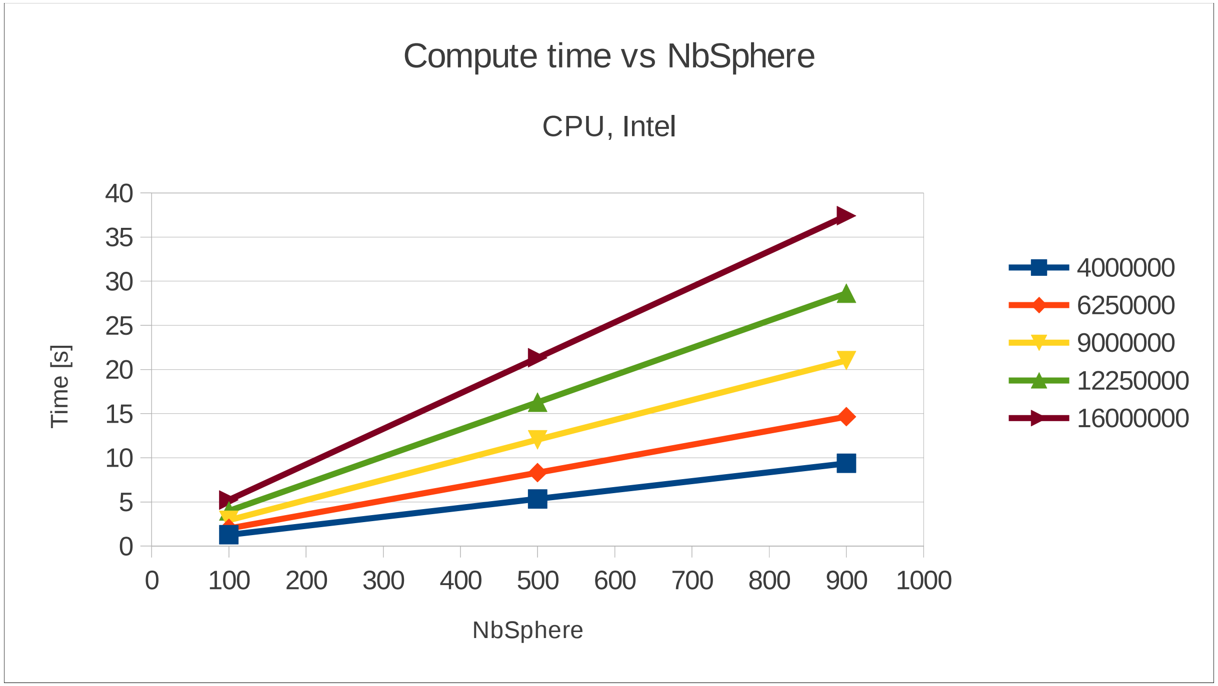


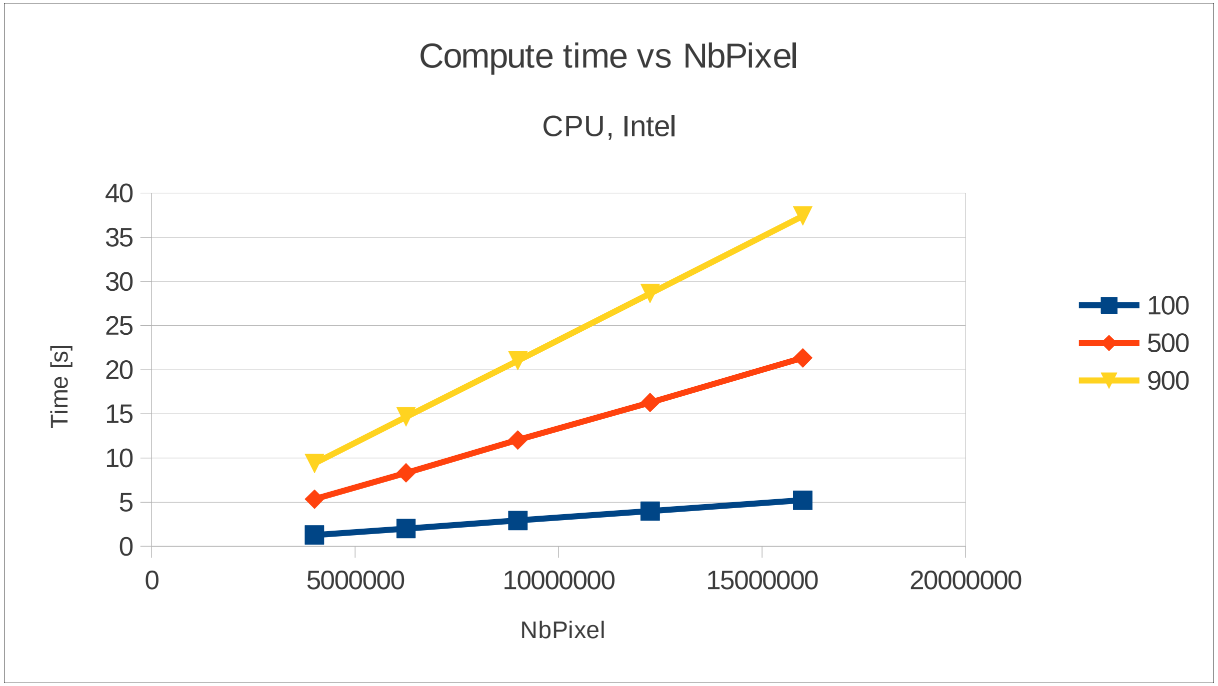
Init time complexity discussion (2)

- Why init time complexity seems to be constant, $O(1)$, versus number of sphere ?
  - Same measure for two different allocations:
    - Image (pixels)
    - Spheres
  - The memory allocation of spheres is not constant, but less significant than the pixel memory allocation, because there is a lot more of pixels than spheres.

Init time complexity conclusions

- One measure for two different things could lead to false interpretation.
- Init time versus number of pixel follows a linear complexity O(n).
- Init time versus number of spheres should follow a linear complexity.
  - Measures with greater numbers of spheres are required to determine this hypothesis.


Compute time vs NbSphere
CPU, Intel
Time [s]
NbSphere
4000000
6250000
9000000
12250000
16000000
0
5
10
15
20
25
30
35
40
0
100
200
300
400
500
600
700
800
900
1000


Compute time vs NbPixel
CPU, Intel
Time [s]
NbPixel
100
500
900
0
5
10
15
20
25
30
35
40
0
5000000
10000000
15000000
20000000

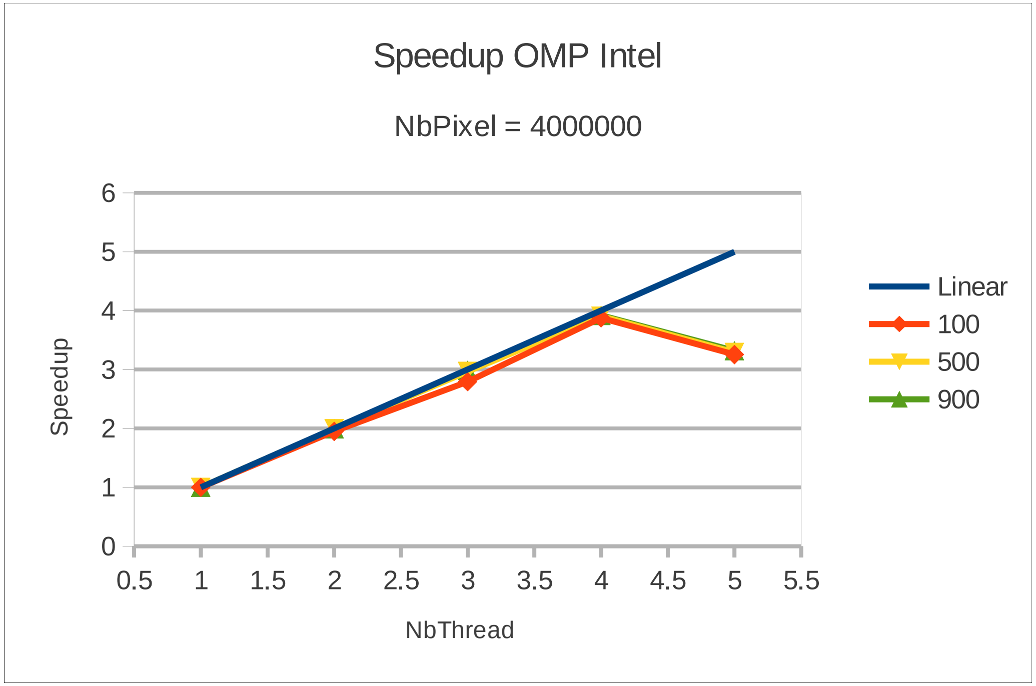
Compute time complexity conclusions

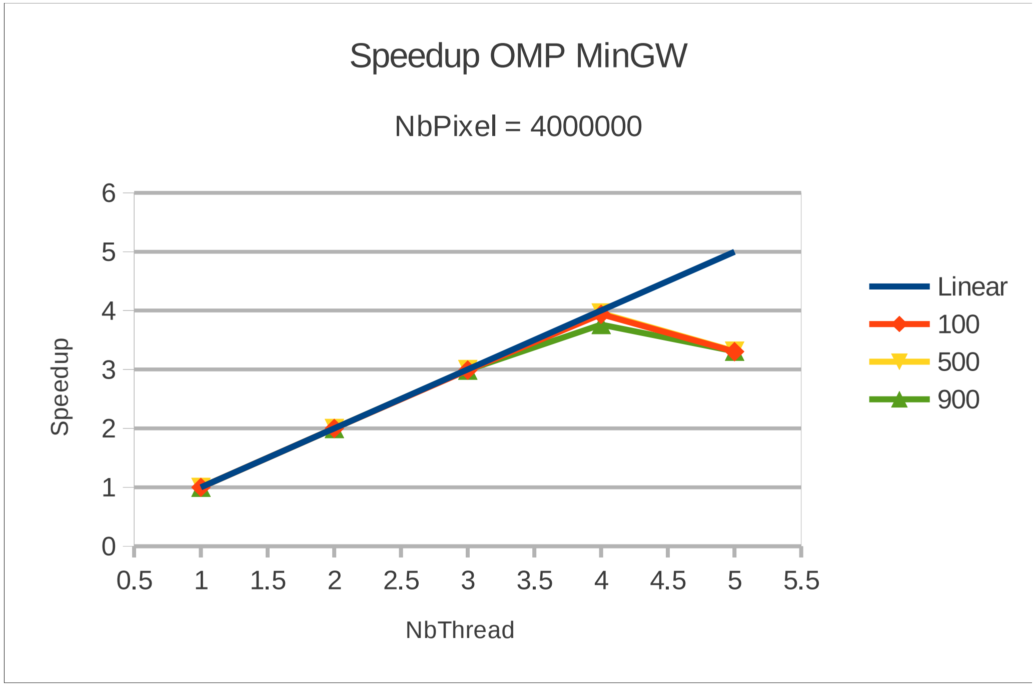
- Both parameters induce a linear complexity.
- Slope of linear law grows as second “size” parameter grows.
- Could be interesting to plot the evolution of slope versus second parameter.

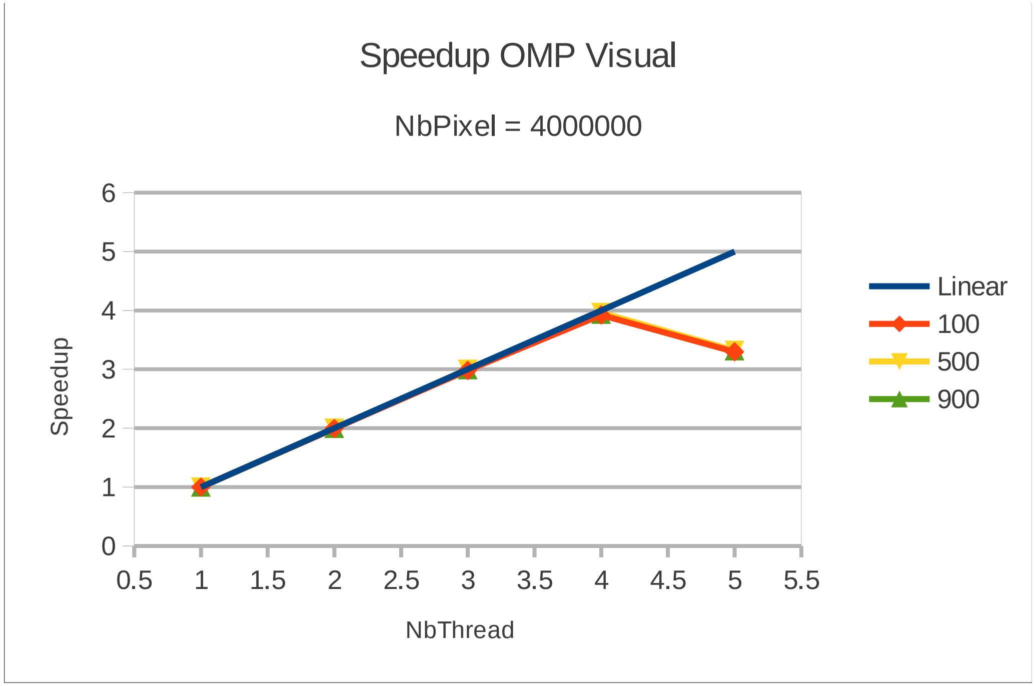
CPU – Compiler comparison

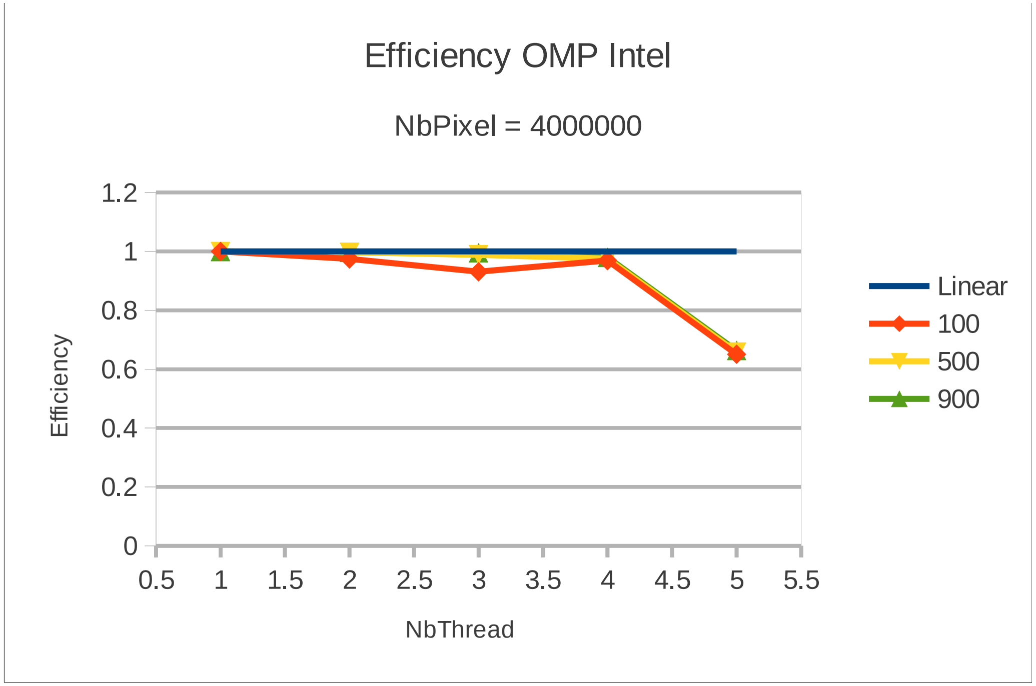
- Compilers
  - Intel
  - MinGW
  - Visual

- Benchmark parameters
  - NbPixel = 2000 x 2000 = 4000000
  - NbSphere = {100, 500, 900}
  - NbThread = {1, 2, 3, 4, 5}

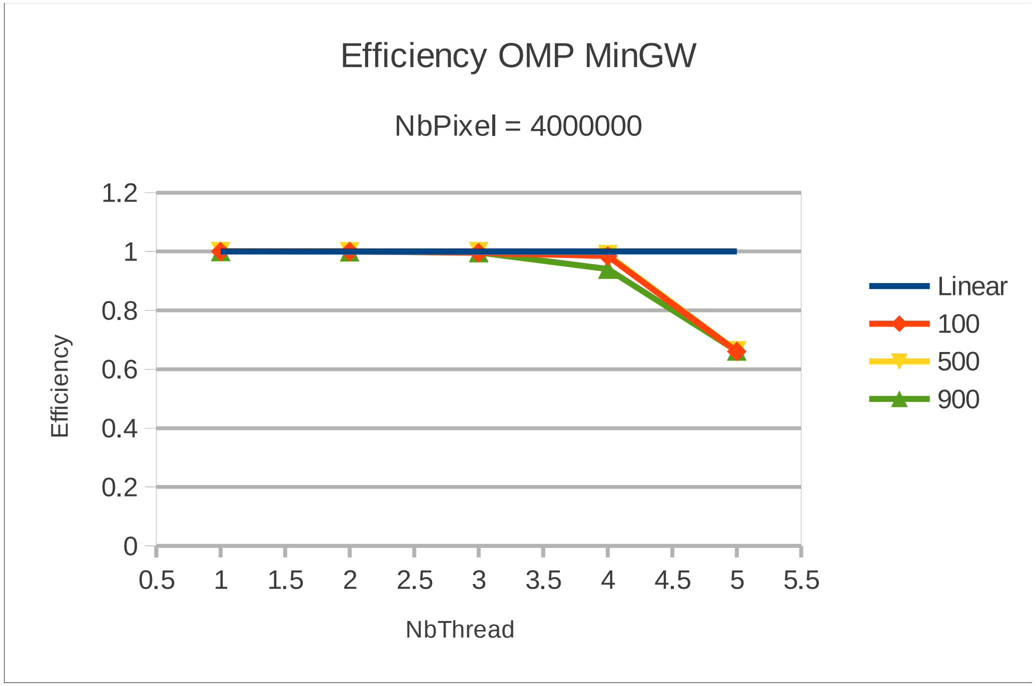

Speedup OMP Intel
NbPixel = 4000000
6
5
4
3
2
1
0
Speedup
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
NbThread
Linear
100
500
900


Speedup OMP MinGW
NbPixel = 4000000
6
5
4
3
2
1
0
Speedup
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
NbThread
Linear
100
500
900

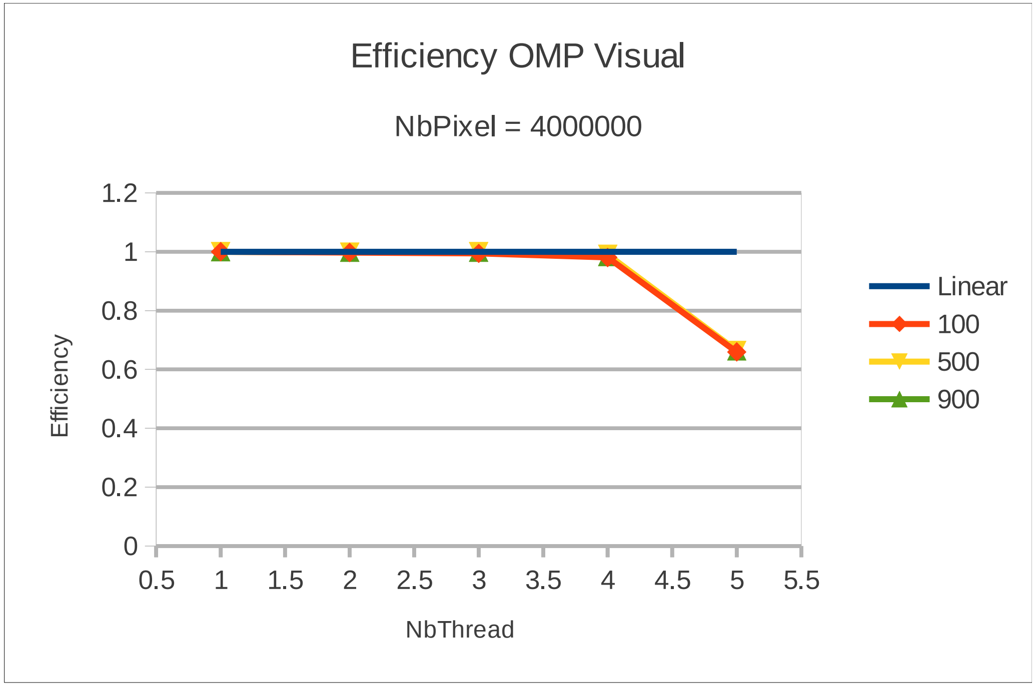

Speedup OMP Visual
NbPixel = 4000000
6
5
4
3
2
1
0
Speedup
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
NbThread
Linear
100
500
900



Efficiency OMP Intel
NbPixel = 4000000
1.2
1
0.8
0.6
0.4
0.2
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
Efficiency
NbThread
Linear
100
500
900



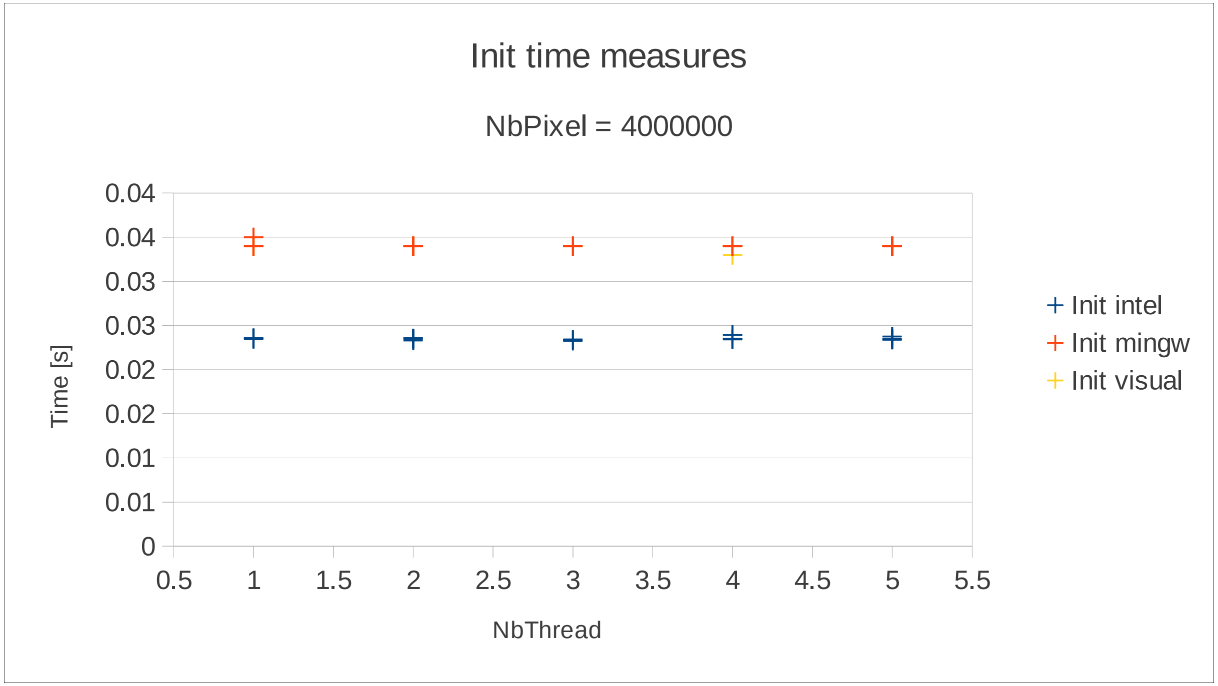
Efficiency OMP MinGW
NbPixel = 4000000
1.2
1
0.8
0.6
0.4
0.2
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
Efficiency
NbThread
Linear
100
500
900

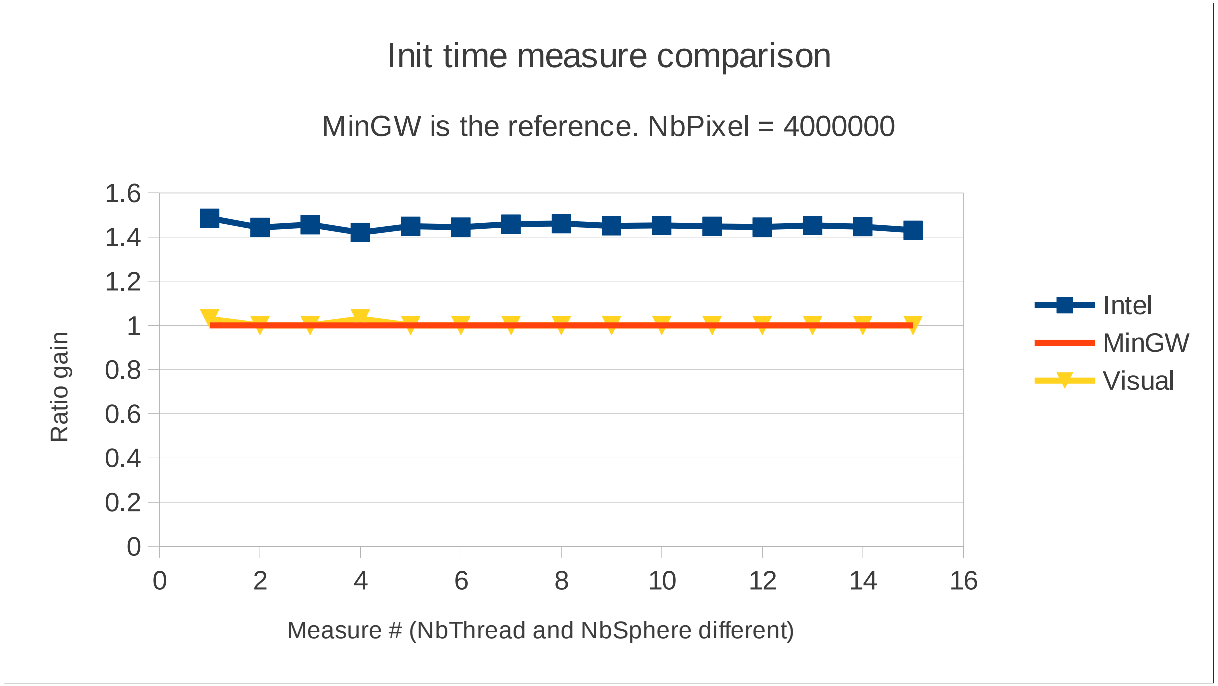


Efficiency OMP Visual
NbPixel = 4000000
1.2
1
0.8
0.6
0.4
0.2
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
Efficiency
NbThread
Linear
100
500
900

Init time comparison

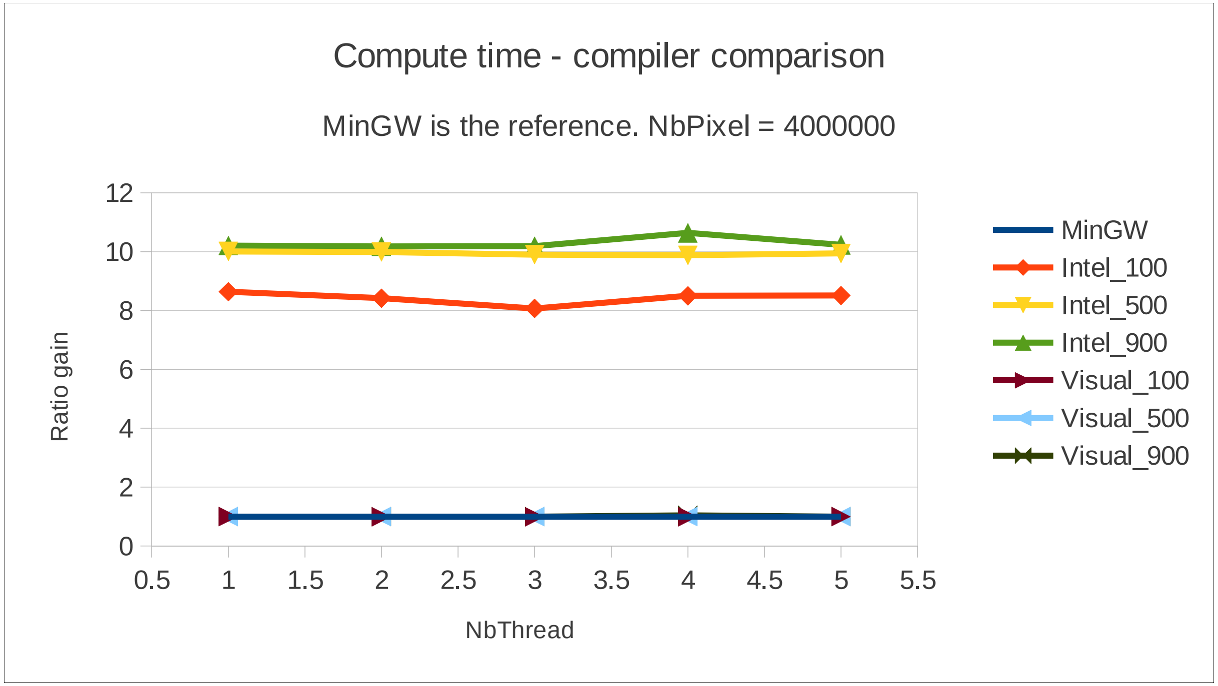
- 3 (NbSphere) * 5 (NbThread) = 15 measures
- 15 measures for each compiler for a given number of pixel

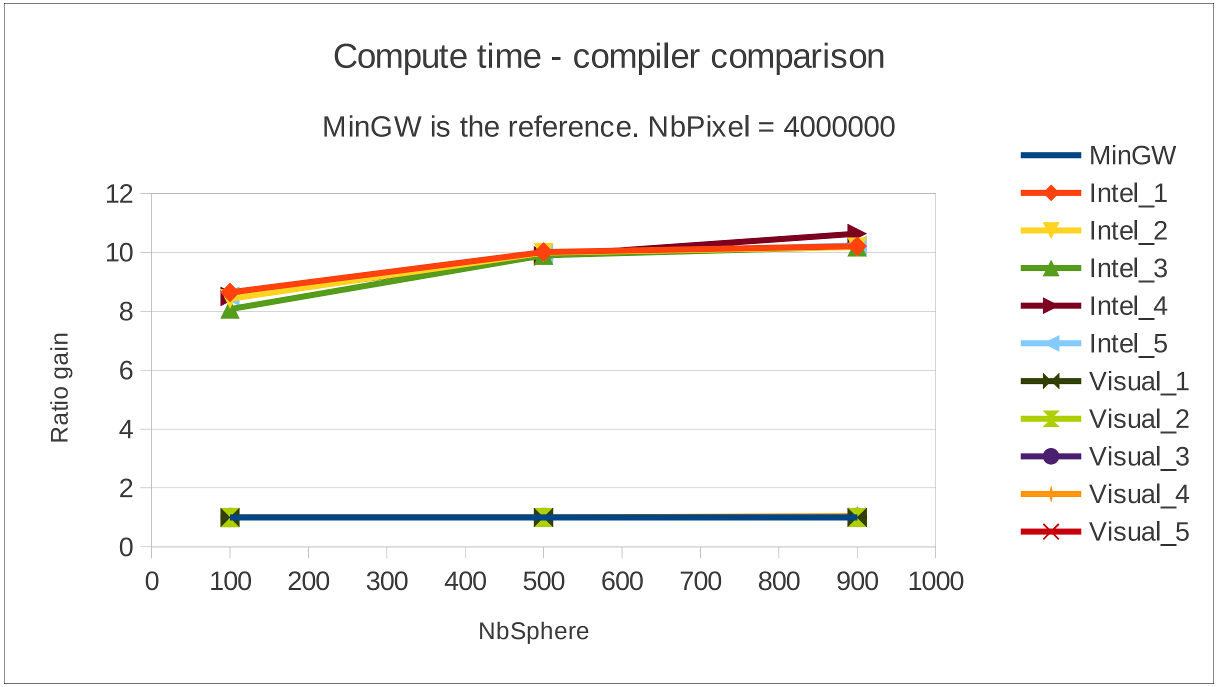



Init time measure comparison
MinGW is the reference. NbPixel = 4000000
Ratio gain
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
12
14
16
Measure # (NbThread and NbSphere different)
Intel
MinGW
Visual

Conclusion of init time comparison

- Intel compiled code is at least 1.4 time faster for memory allocation (pixel mostly) than MinGW or Visual.
- Visual and MinGW compilers have equivalent performance (or are as bad) for memory allocation.
- Init time is not improved by number of thread because it is a sequential code.


Compute time - compiler comparison
MinGW is the reference. NbPixel = 4000000
Ratio gain
12
10
8
6
4
2
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
NbThread
MinGW
Intel_100
Intel_500
Intel_900
Visual_100
Visual_500
Visual_900


Compute time - compiler comparison
MinGW is the reference. NbPixel = 4000000
Ratio gain
12
10
8
6
4
2
0
0
100
200
300
400
500
600
700
800
900
1000
NbSphere
MinGW
Intel_1
Intel_2
Intel_3
Intel_4
Intel_5
Visual_1
Visual_2
Visual_3
Visual_4
Visual_5

Conclusion of compiler comparison for compute time

- Performance gain provided by Intel compiler is independent from number of thread.
- Intel performance gain versus number of sphere follows a logarithmic law.
- Visual and MinGW compilers are as bad as one another for computation.
- Intel compiler is up to ten times faster than Visual or MinGW for computation part.

GPU

- Benchmarks parameters
  - NbPixel = [1'000'000, 100'000'000]
  - NbSphere = 500
  - MemType = {Global, Shared, Constant}
  - dg (dimension of grid) and db (dimension of block) defined by range later.

GPU, wrap a present

- Could a wrap be made of threads from different blocks ?
  - If yes, we could assume that in term of performance 16x32 and 512x1 (dg x db) are closed (for none-shared versions).
  - If no, performance of 512x1 should be horrible compared to 16x32.

GPU, push the limits

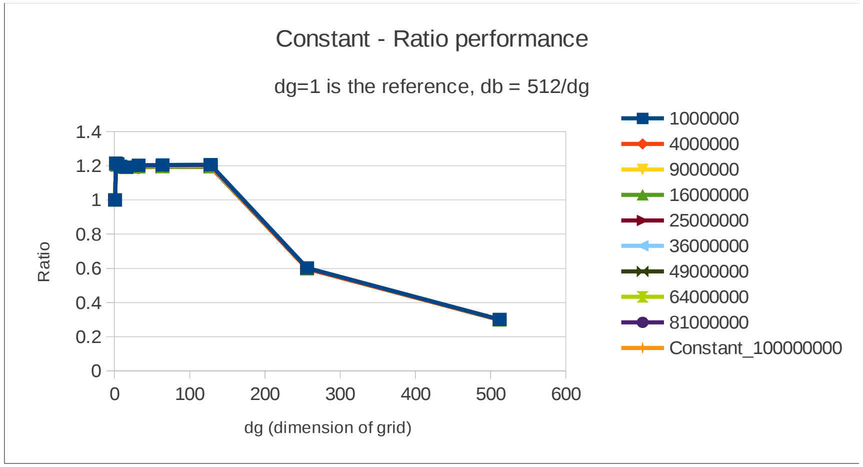
- Could we measure the speedup of streaming multiprocessors ?
  - By design, a block is “assigned” to a SM => dg = [1, 16].
  - dg becomes the correspondence of nbThread in OMP.
  - Total number of thread is kept constant
    - Multiple wraps per SM when dg < 16.
    - Could we obtain “good” performance by using dg < 16 and db > 32 ?
    - What loss to expect when dg > 16 and db < 32 ?
  - Total number of thread is variant
    - Dimension of block is a constant, db = 32.
    - Total number of thread depends of dg.
    - Would better correspond to the speedup we know with CPU cores.

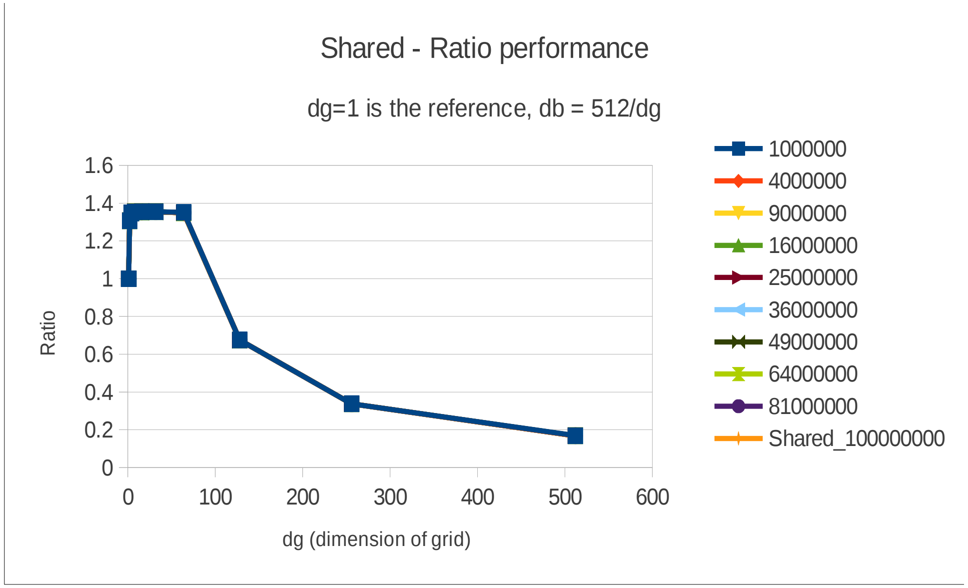
Parameters dg,db

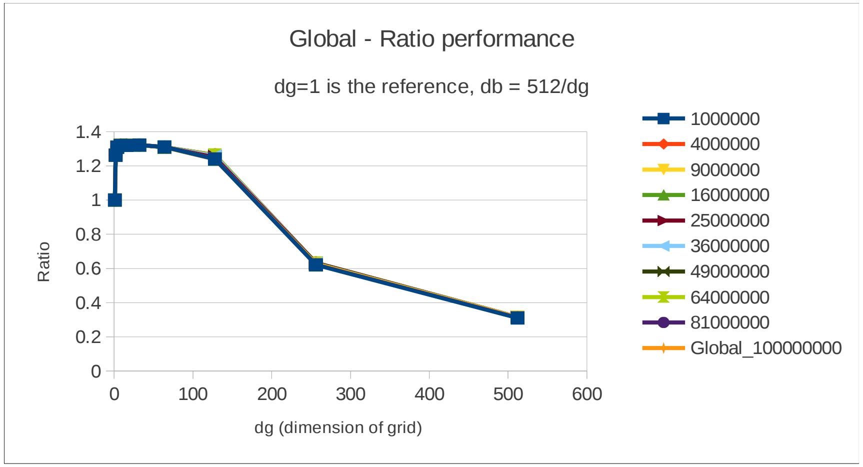
- dg = [1, 512]
- db = [1, 512]
- dg * db = 512
- dg,db = {1,512; 2,256; 4,128; 8,64; 16,32; 32,16; 64,8; 128,4; 256,2; 512,1}

Basic observations

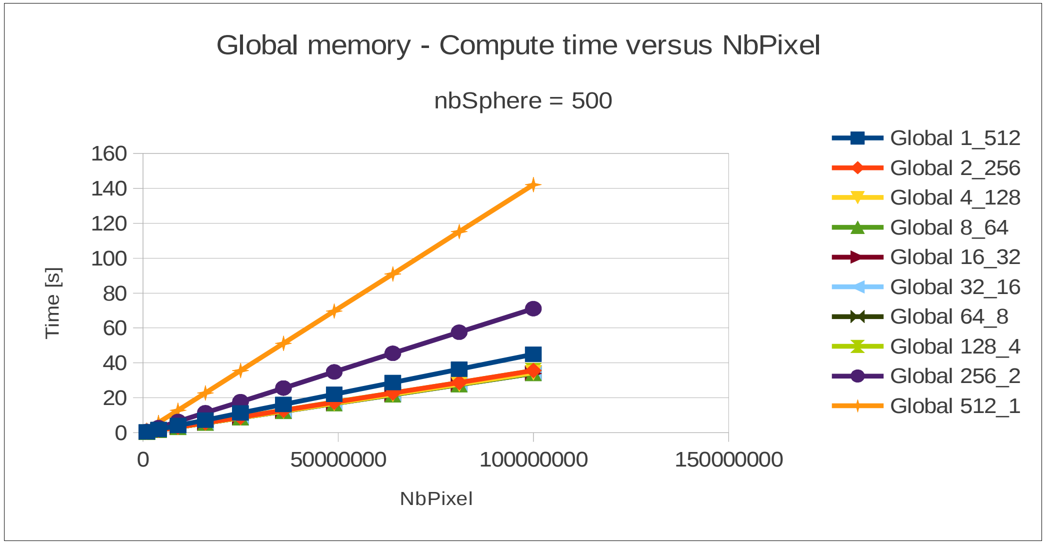
- Performance gain/loss by changing dg,db is independent of number of pixel.
- Performance gain/loss is dependent of memory type.

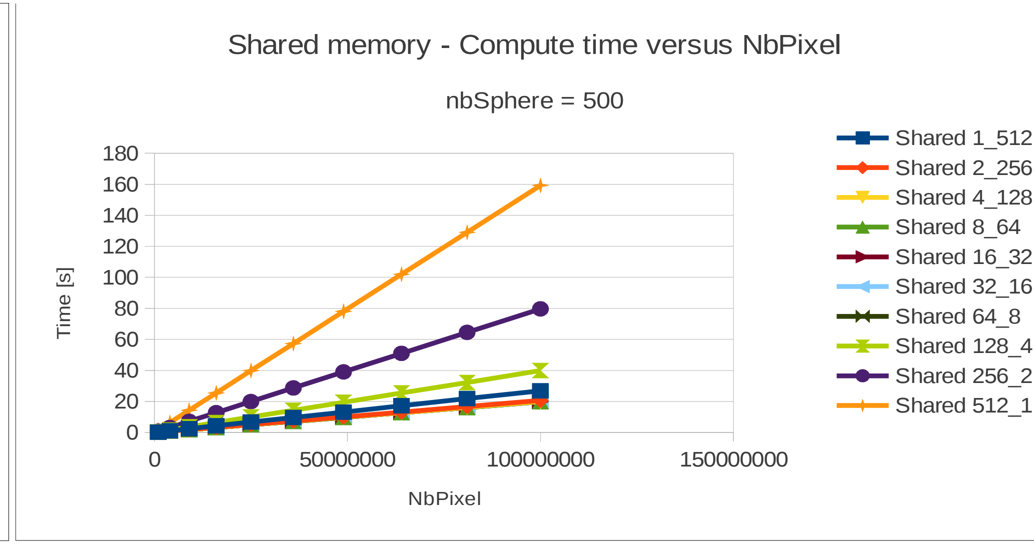


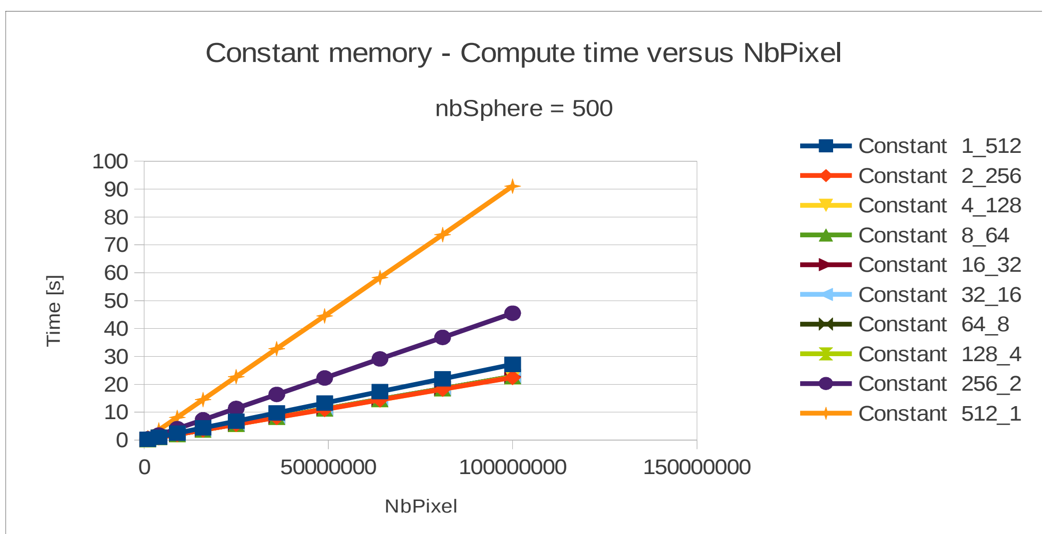


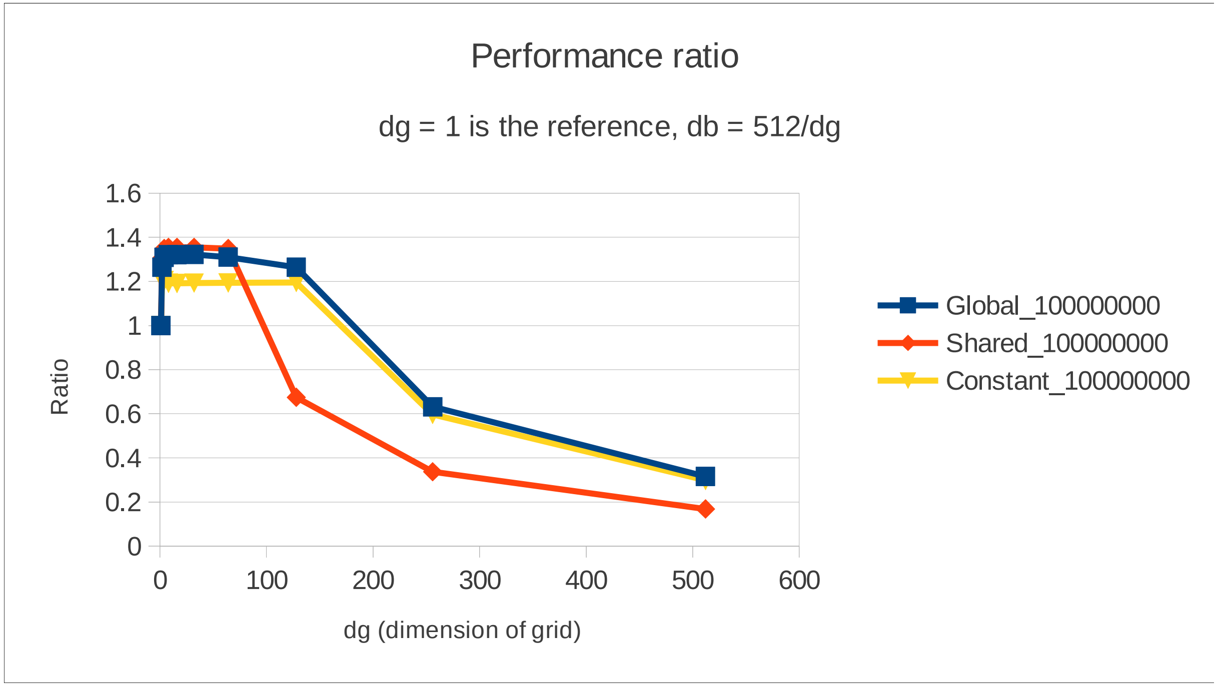


Basic observations

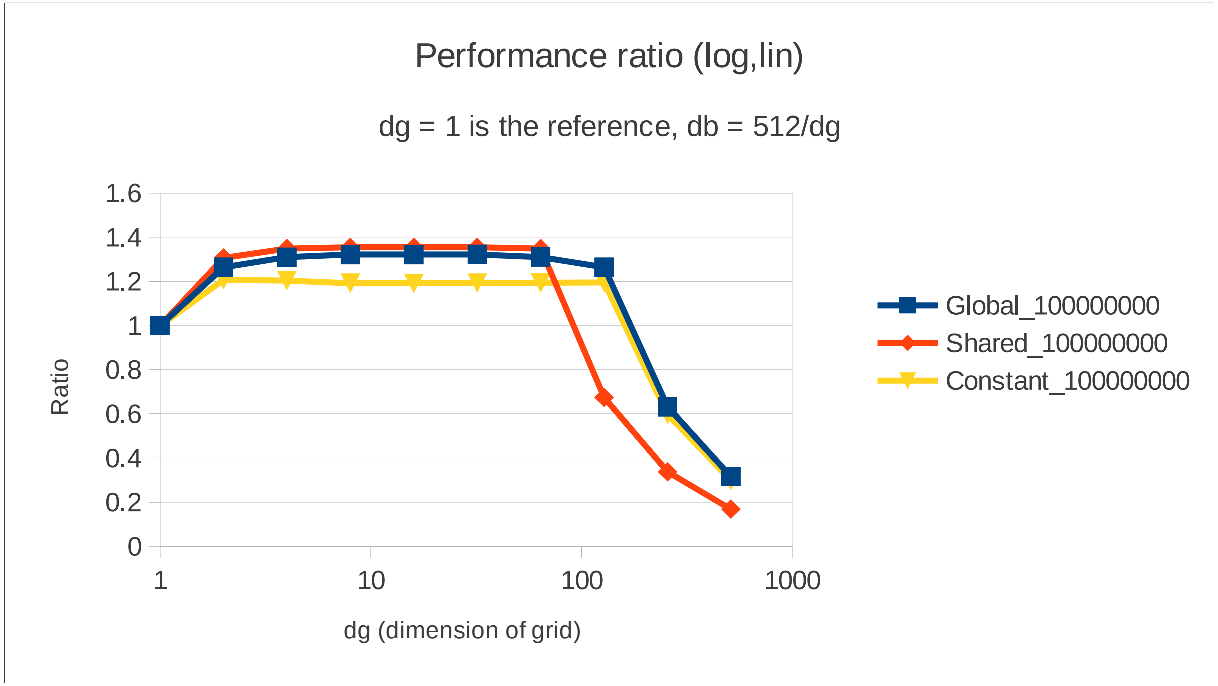








Performance ratio
dg = 1 is the reference, db = 512/dg
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Ratio
0
100
200
300
400
500
600
dg (dimension of grid)
Global_100000000
Shared_100000000
Constant_100000000



Performance ratio (log,lin)
dg = 1 is the reference, db = 512/dg
Ratio
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
10
100
1000
dg (dimension of grid)
Global_100000000
Shared_100000000
Constant_100000000

Conclusion GPU dg,db

- First of all let’s define “good” performance as a performance gain similar of 16x32 (+- 0.1).

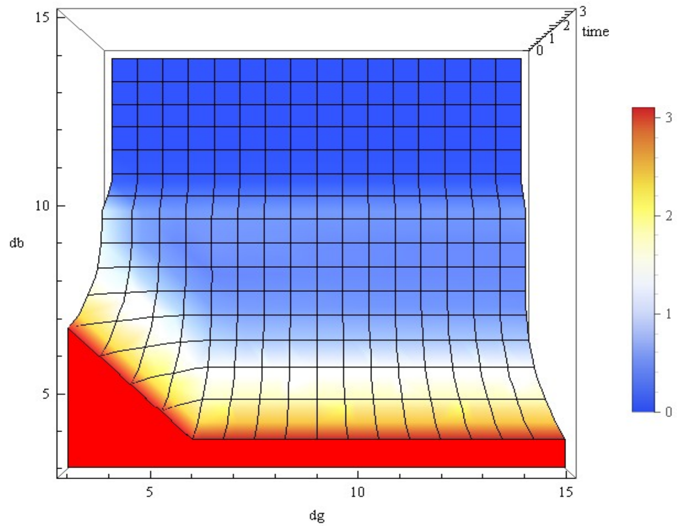
- 2 <= dg <= 128 (4 <= db <= 256)
  - -> performance are “good” for constant and global memory.
- 2 <= dg <= 64 (8 <= db <= 256)
  - -> performance are “good” for shared memory.

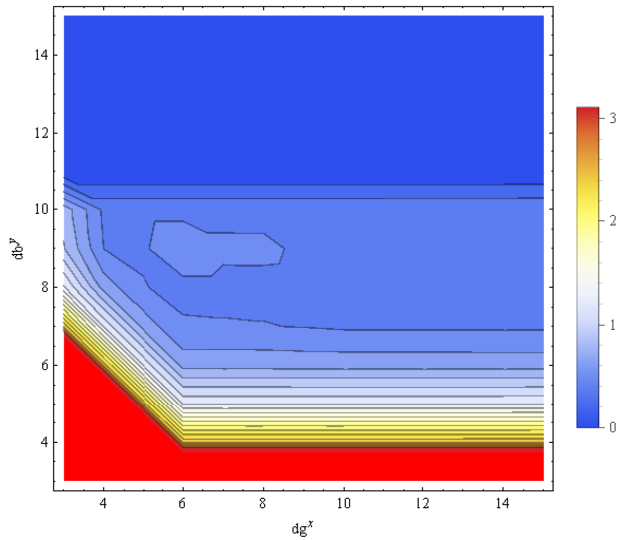
- With dg < 16 (and thus db > 32), “good” performance are achievable.
  - In other words: We can achieve good performance without using all SM.
- Loss when 16 < dg < 64 (and thus db < 32!) is mitigate.
  - In other words: We can achieve good performance with wrap < 32.
- Value of dg > 64 (db < 8) induces great loss.

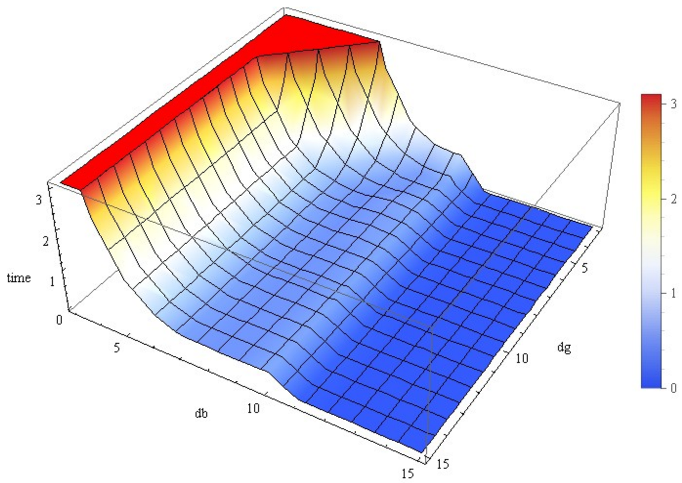
Rule them all

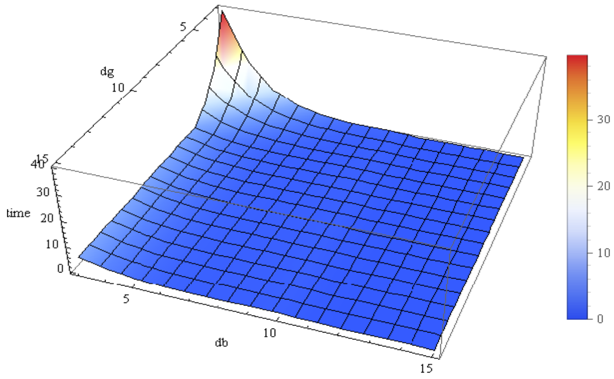
- db = [2^3, 2^15]
- dg = [2^3, 2^15]
- NbSphere = 500
- NbPixel = 25'000'000

Rule them all
Shared

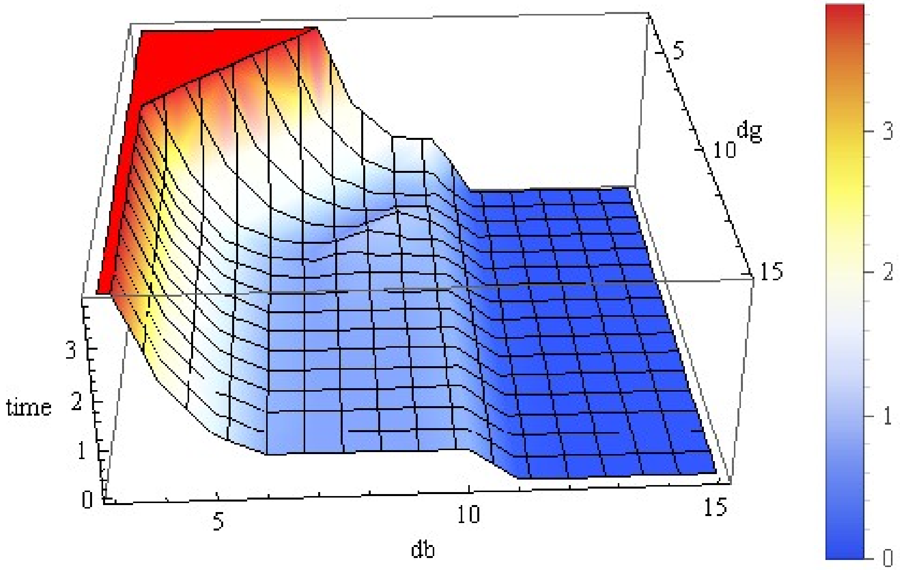


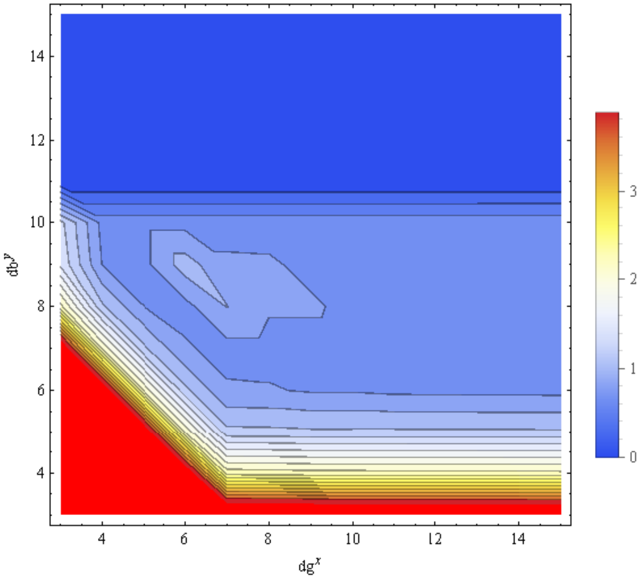




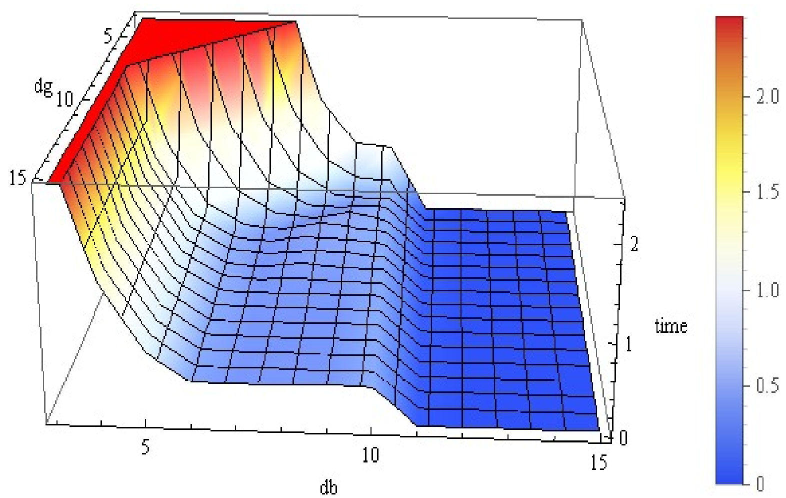


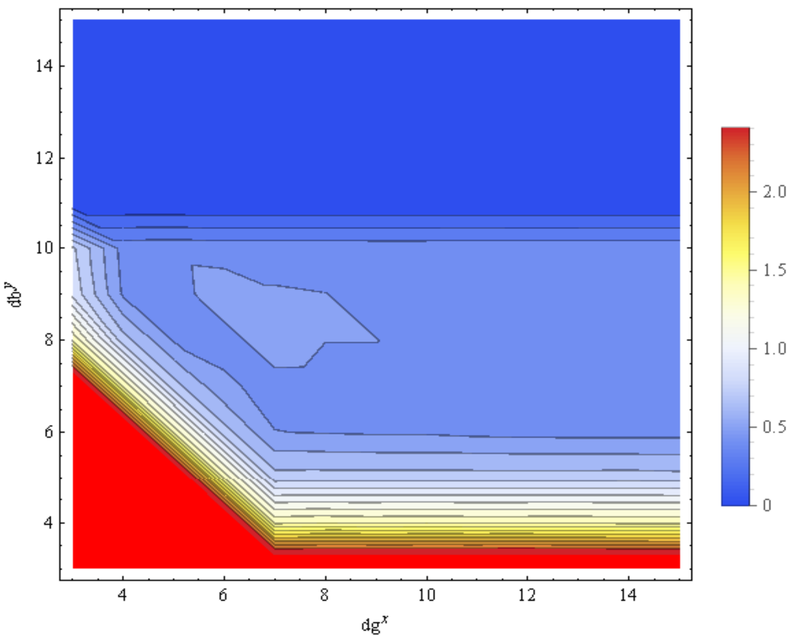
Rule them all
Global





Rule them all
Constant





dg vs db

- Specific for each algorithm
  - And memory implementation
- Apply rules to “stuff it”
  - dg should be a multiple of SM
  - db should be a multiple of wrap size
- Compute dg,db by benchmarking or theory
- Greater dg is for sure useless without a sufficient db

dg vs db

- Specific for each algorithm
  - And memory implementation
- Apply rules to “stuff it”
  - dg should be a multiple of SM
  - db should be a multiple of wrap size
- Compute dg,db by benchmarking or theory
- Greater dg is for sure useless without a sufficient db

Hey listener are you attentive ?

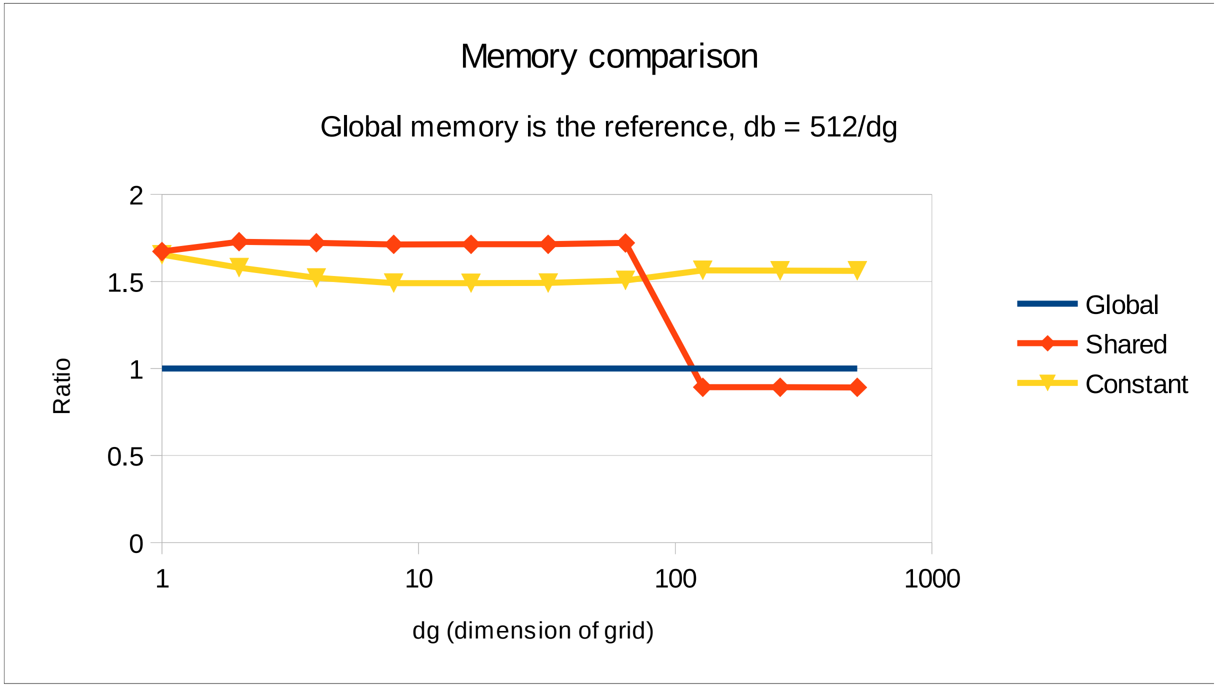
- Why all this deep blue in preceding plots ?
- Is there upper limits for db,dg ?

Hey listener are you attentive ?

- Why all this deep blue in preceding plots ?
- Is there upper limits for db,dg ?
  - Yes they are upper limits:
    - Maximum x- or y-dimension of a block 1024
    - Maximum number of threads per block 1024
- Kernel call with bad dimension without check for errors → data in plot

Memory comparison

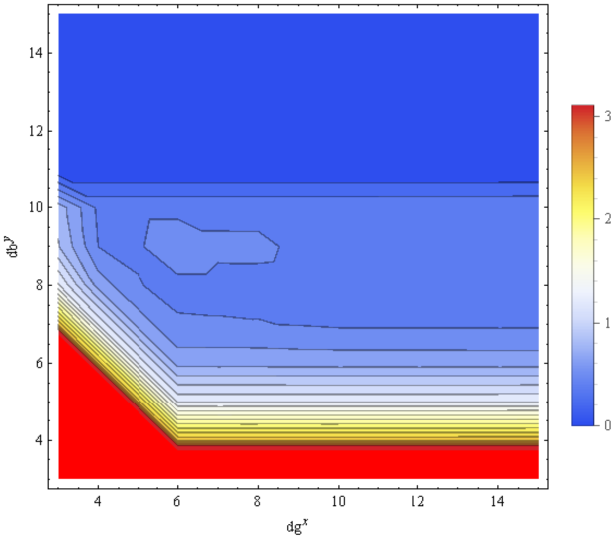
- Performance gain of constant over global is less affected by dg,db values than shared over global.
- Shared is better for normal values of dg,db
  - Normal = multiplier of SM/wrap, 16x32 in this case



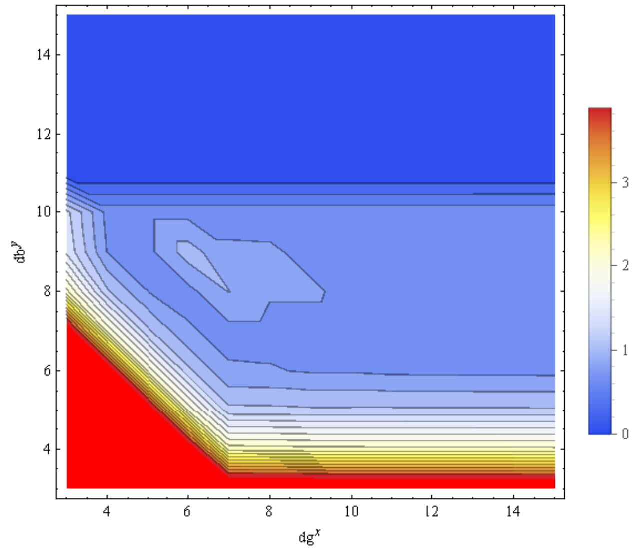
Global, shared or constant

- Each implementation is affected differently by the dg,db parameters
- Thus, each implementation has a different dg,db optimized value.

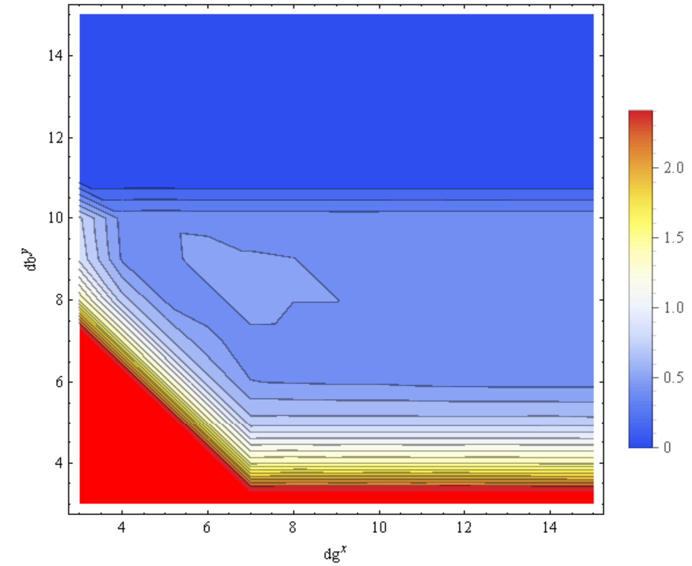
Shared

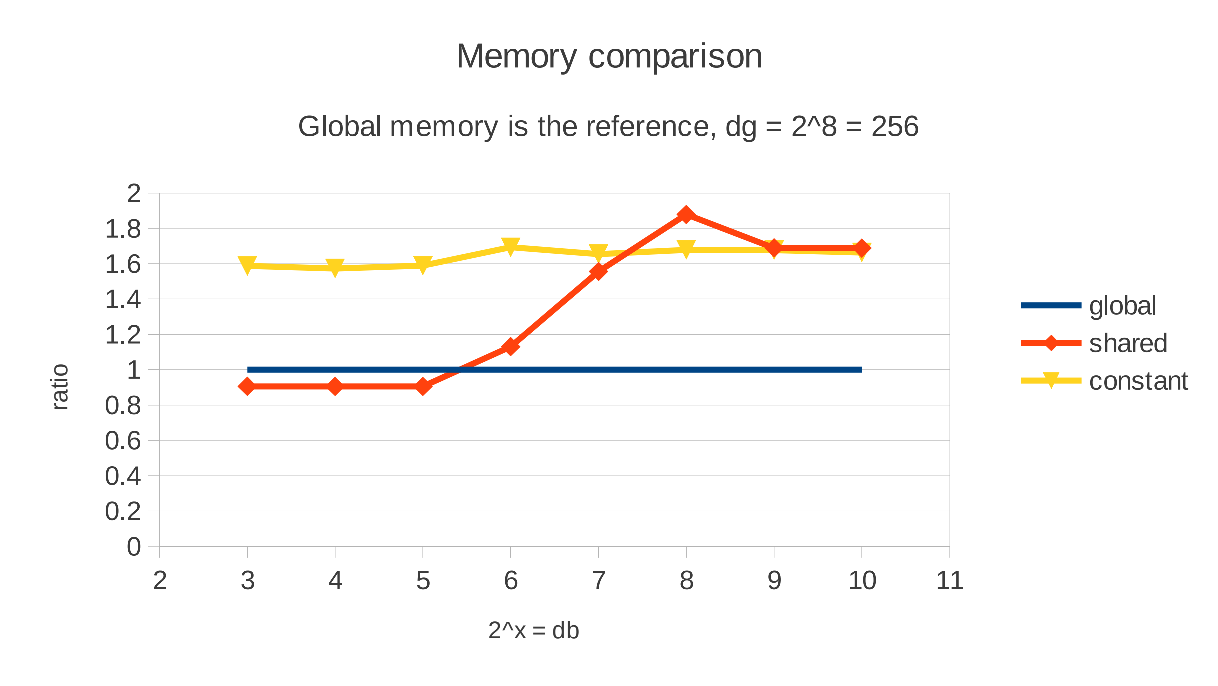


Global



Constant





Final conclusion

- Is the “stuff it” a best practice ?
- Concurrent kernel with Fermi architecture.
- The search of perfect dg,db.
- The others dimensions of dg,db.
- Kernel side measures (memory copies)
- The curious spot in dg,db plots.