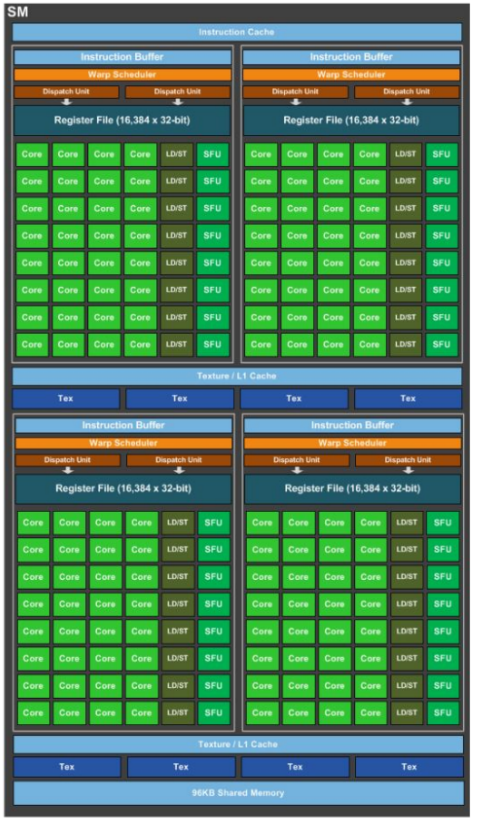
Lab 4 - Cuda

Parallel Computing

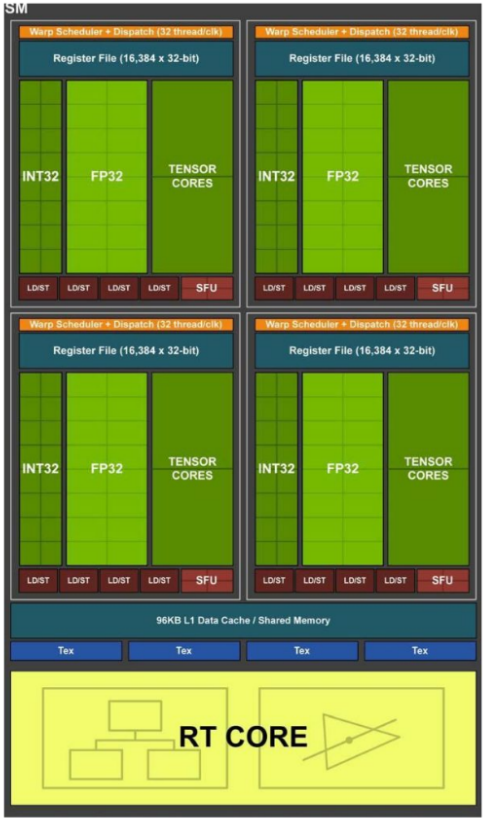
Mateo de Mayo - Benjamín Ocampo

Hardware

- RTX 2080 Ti (target device): 68 Turing SMs
- GTX 1060 MaxQ (test device): 10 Pascal SMs



Pascal SM



Turing SM

Migration: from CPU to GPU

Migration: from CPU to GPU

In order to put our hands over the work and looking up optimizations, we needed to migrate all our implementation to Cuda code.

Remember that the simulation consists of a number of updates or steps

Simulation Step

- *react*
- *step*:
  - *addsource*
  - *diffuse*
  - *project*
  - *advect*

Migration: from CPU to GPU

The migration process consisted in implementing them one by one, and checking if the simulation keeps well after it.

Everything was performed incrementaly, synchronizing with the device and the host.

But it was momentary until the migration was fully-implemented. It might be thought as the device "asking for help to the host", since it can not handle the th the functions.

Migration: from CPU to GPU

Our Goal: Launch the entire program to the GPU at the beggining (**fullburst**), and synchronize at the end of the simulation. Then, measures will be normalized at that point.

Things that were on mind during the process:

- changes of signatures
- makefile rules
- deletion of omp pragma
- new includes
- differences between device and host data.
- synchronizations (by means of kernels)

reactburst

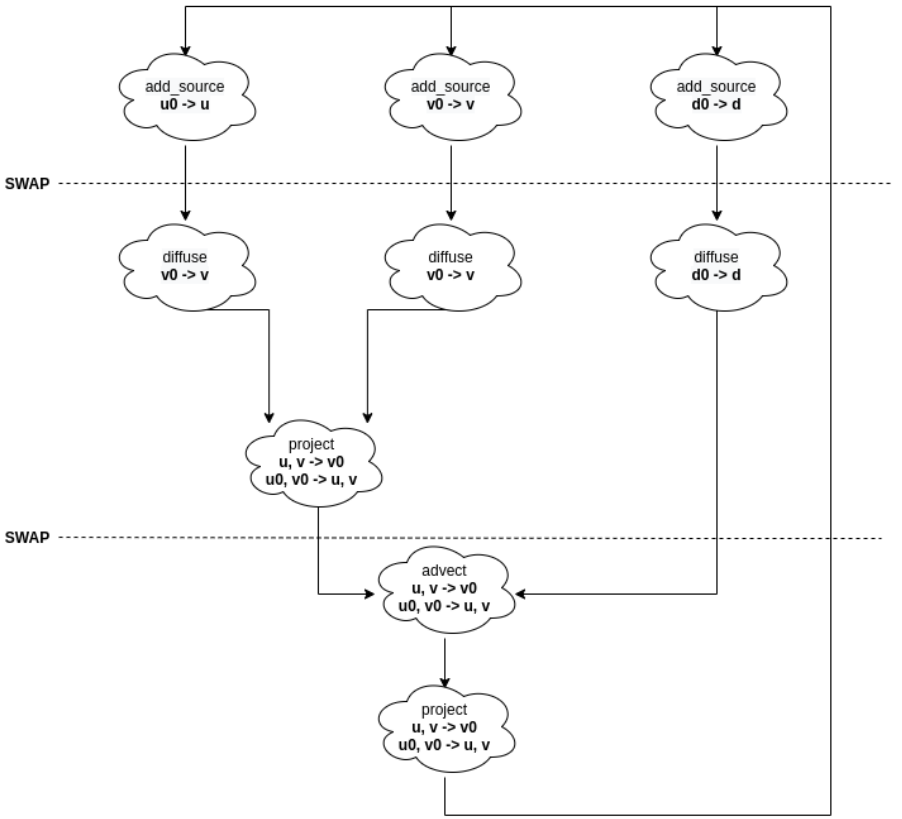
reactburst

The first version of this lab!

It was called *reactburst* since *step* was computed by the GPU, but *react* was still performed by the CPU (without changes from the lab 3), i.e, first *react*, then burst of kernels.

In order to do so, we had to deal with the migration of *step*.

reactburst

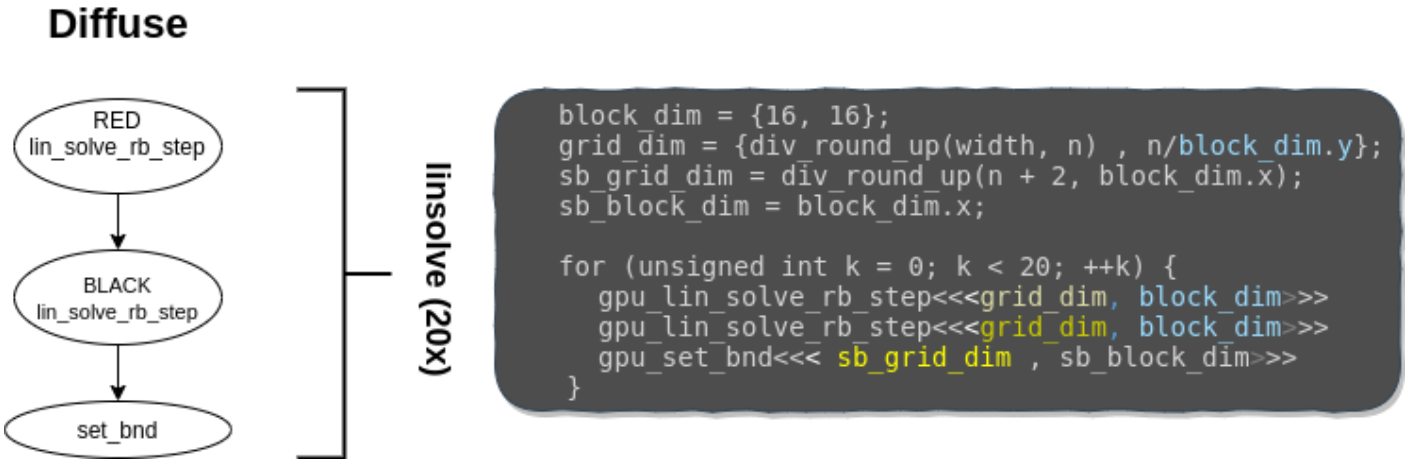


- We had to decide which cloud might be a kernel and which one might be just a host function that launches other kernels.

reactburst: add_source

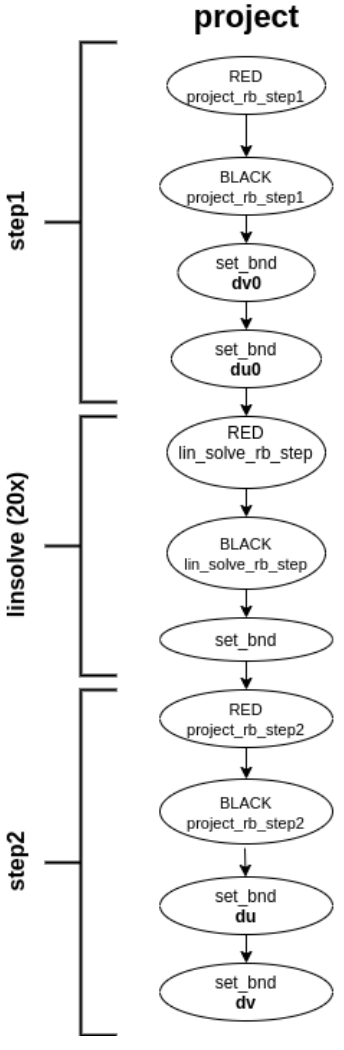
In this case was straighforward, since *add_source* is not a complex function and *diffuse* can not start without the previous execution of *add_source*. It was converted into a kernel.

reactburst: diffuse



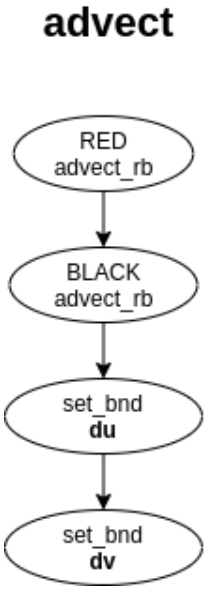
- Three kernel launches for each iteration.
- Implicit synchronizations with kernels.
- block_dim was chosen according to the hardware.
- grid_dim was chosen according to the problem.
- gpu_set_bnd is launched in one dimension.

reactburst: project




```
block_dim = {16, 16};
grid_dim = {div_round_up(width, block_dim.x) , n/block_dim.y};
sb_grid_dim = div_round_up(n + 2, block_dim.x);
sb_block_dim = block_dim.x;

gpu_project_rb_step1<<<grid_dim, block_dim>>>(...)
gpu_project_rb_step1<<<grid_dim, block_dim>>>(...)

gpu_set_bnd<<<sb_grid_dim, sb_block_dim>>>(...)
gpu_set_bnd<<<sb_grid_dim, sb_block_dim>>>(...)

lin_solve(n, NONE, 1, 4, du0, dv0);

gpu_project_rb_step2<<<grid_dim, block_dim>>>(...)
gpu_project_rb_step2<<<grid_dim, block_dim>>>(...)

gpu_set_bnd<<<sb_grid_dim, sb_block_dim>>>(...);
gpu_set_bnd<<<sb_grid_dim, sb_block_dim>>>(...);
```

reactburst: advect




```
block_dim{16, 16};
grid_dim{div_round_up(width, block_dim.x), n / block_dim.y};
sb_grid_dim = div_round_up(n + 2, block_dim.x);
sb_block_dim = block_dim.x;

gpu_advect_rb<<<grid_dim, block_dim>>>
gpu_advect_rb<<<grid_dim, block_dim>>>
gpu_set_bnd<<<sb_grid_dim, sb_block_dim>>>
gpu_set_bnd<<<sb_grid_dim, sb_block_dim>>>
```

reactburst: kernels

```
__global__
void gpu_lin_solve_rb_step(...) {
  ... // initialize grid_width, grid_height, gtidx and gtidx
  for (int y = 1 + gtidy; y <= n; y += grid_height) {
    for (int x = start + gtidx; x < width - (1 - start); x += grid_width) {
      int index = y * width + x;
      same[index] = (same0[index] + a * (
          neigh[index - width] +
          neigh[index - start] +
          neigh[index - start + 1] +
          neigh[index + width]
      )) / c;
    }
  }
}
```

- For each kernel, grid stride loop was implemented (even if unused).
- looping over the grids, working with the global id of each thread related to the current grid that we are working.
- project and advect are similar.

reactburst: kernels

```
__global__
void gpu_set_bnd(...) {
  const int grid_width = gridDim.x * blockDim.x;
  const int gtid = blockIdx.x * blockDim.x + threadIdx.x;
  for (unsigned int i = 1 + gtid; i <= n; i += grid_width)
    ... // fill i cell of {first, last} × {row, column}
  if (gtid == 0)
    ... // fill the four corners
}
```

- Again, grid stride loop over 1 dimension.
- border cases were handled by just one thread (*gtid = 0*).

reactburst: pitfalls

```
react(hd_prev, hu_prev, hv_prev); // Works on host

checkCudaErrors(cudaMemcpy(dd_prev, hd_prev, size_in_m,cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(du_prev, hu_prev, size_in_m, cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(dv_prev, hv_prev, size_in_m, cudaMemcpyHostToDevice));

step( // Works on device
  N, diff, visc, dt, dd, du, dv, dd_prev, du_prev, dv_prev, from, to
);

checkCudaErrors(cudaDeviceSynchronize());
checkCudaErrors(cudaMemcpy(hd_prev, dd_prev, size_in_m, cudaMemcpyDeviceToHost));
checkCudaErrors(cudaMemcpy(hu_prev, du_prev, size_in_m, cudaMemcpyDeviceToHost));
checkCudaErrors(cudaMemcpy(hv_prev, dv_prev, size_in_m, cudaMemcpyDeviceToHost));
```

- `react` is computed by the host. `step` by device.
- We can not avoid a synchronization and copies from host to device at that point for each simulation step.

threactburst

threactburst

Obviously, the bottleneck of *reactburst* is how *react* is computed. So let us get deep into that function to see what can be done to make it faster.

threactburst: before optimizing

```
// Two reductions
for (i = 0; i < size; i++) {
  if (max_velocity2 < uu[i] * uu[i] + vv[i] * vv[i]) {
    max_velocity2 = uu[i] * uu[i] + vv[i] * vv[i];
  }
  if (max_density < d[i]) {
      max_density = d[i];
  }
}
// Memsets
for (i = 0; i < size; i++) {
  uu[i] = vv[i] = d[i] = 0.0f;
}
```

- Two reductions to be handled.
- Use of memset instead of loops.

threactburst: before optimizing

```
// Do something with the reductions.
if (max_velocity2 < 0.0000005f) {
  uu[IX(N / 2, N / 2)] = force * 10.0f;
  vv[IX(N / 2, N / 2)] = force * 10.0f;
  for (int y = 64; y < N; y += 64)
    for (int x = 64; x < N; x += 64) {
      uu[IX(x, y)] = force * 1000.0f * (N / 2 - y) / (N / 2);
      vv[IX(x, y)] = force * 1000.0f * (N / 2 - x) / (N / 2);
    }
}
if (max_density < 1.0f) {
  d[IX(N / 2, N / 2)] = source * 10.0f;
  for (int y = 64; y < N; y += 64)
    for (int x = 64; x < N; x += 64) d[IX(x, y)] = source * 1000.0f;
}
```

- Read and writes that can be done by the GPU.

threactburst: Reductions

```
// Reduction 1
  dfloatp tdd_prev(dd_prev);
  float max_density = *thrust::max_element(tdd_prev, tdd_prev + size);
  // Implicit cudaDeviceSynchronize();
```

threactburst: Reductions

```
// Reduction 2
// Note: Not real c++ syntax, a more pythonic pseudocode

struct compare_dfloatp2 { // Comparisson functor
  __device__
  bool operator()(tfloat2 lhs, tfloat2 rhs) {
    ... // compute which cell is faster
    return is_rhs_faster;
  }
};

auto uvs_begin = zip(du_prev, dv_prev);
auto uvs_end = zip(du_prev + size, dv_prev + size)
auto zmaxvel2 = max_element(uvs_begin, uvs_end, compare_dfloatp2());
dfloatp2 mv2 = zmaxvel2.get_iterator_tuple();
float mvu = *mv2[0]; // Implicit cudaDeviceSynchronize()
float mvv = *mv2[1]; // on device pointer dereference
float max_velocity2 = mvu * mvu + mvv * mvv;
```

threactburst: launches and memsets

```
// Memsets and kernel launches
size_t size_in_mem = size * sizeof(float);
checkCudaErrors(cudaMemsetAsync(du_prev, 0, size_in_mem));
checkCudaErrors(cudaMemsetAsync(dv_prev, 0, size_in_mem));
checkCudaErrors(cudaMemsetAsync(dd_prev, 0, size_in_mem));

// The gridblock mapping is one thread per reactionary point
dim3 block_dim{16, 16};
dim3 grid_dim{N / 64 / block_dim.x, N / 64 / block_dim.y}; // rounded up div

if (max_velocity2 < 0.0000005f)
  gpu_react_velocity<<<grid_dim, block_dim>>>(du_prev, dv_prev, force, N);

if (max_density < 1.0f)
  gpu_react_density<<<grid_dim, block_dim>>>(dd_prev, source, N);
```

- Note: We are not dealing with how react and step communicate, so those copies that were shown before are still here in *threactburst*.

stepburst

stepburst

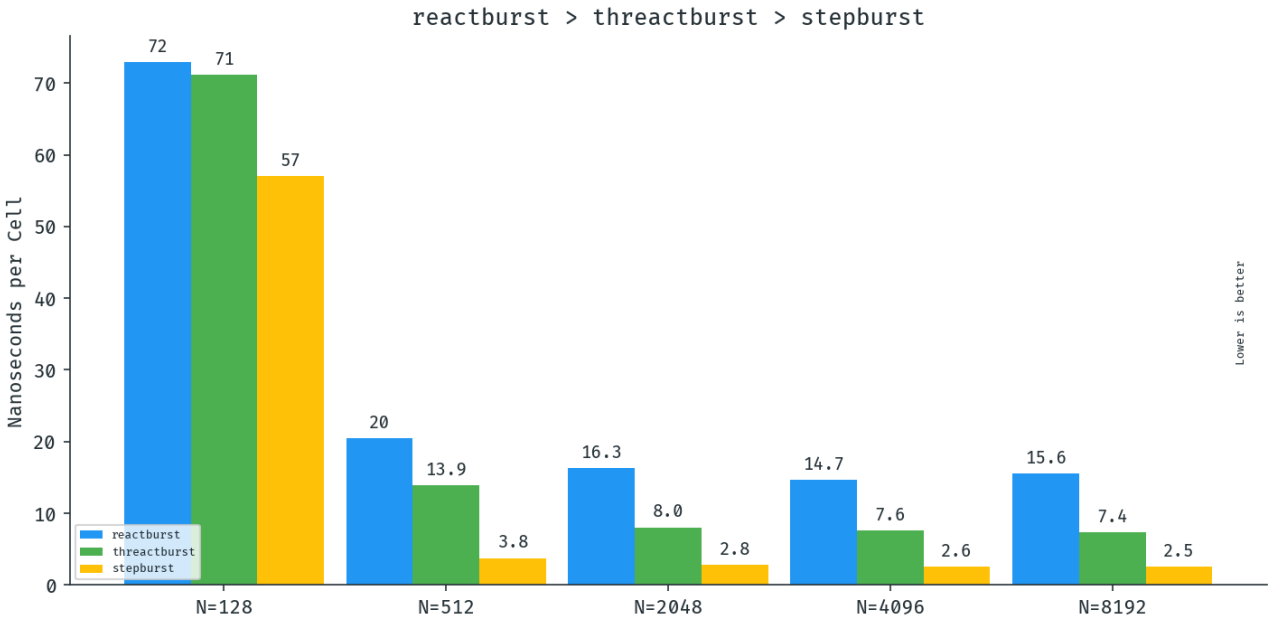
It is time to make copies between react and step dissapear!

```
int main() {
  initialize_grids();
  for (i = 0; i < steps; i++) one_step();
  cudaMemcpy(hd, dd, size_in_mem,cudaMemcpyDeviceToHost);
  cudaMemcpy(hu, du, size_in_mem,cudaMemcpyDeviceToHost);
  cudaMemcpy(hv, dv, size_in_mem,cudaMemcpyDeviceToHost);
  cudaMemcpy(hd_prev, dd_prev, size_in_mem,cudaMemcpyDeviceToHost);
  cudaMemcpy(hu_prev, du_prev, size_in_mem,cudaMemcpyDeviceToHost);
  cudaMemcpy(hv_prev, dv_prev, size_in_mem,cudaMemcpyDeviceToHost);
}
```

- In this version, a bunch of kernels launches are queued up to the thrust synchronization.
- Copies appear now only at the end of the simulation.
- *react* and *step* use memory that is on the device. So, the CPU does not do more than just launch kernels.
- If it were not for the fact that thrust dereferences are blocking, this one would be the *fullburst* version that we were aiming for.

reactburst VS threactburst VS stepburst

Results


reactburst > threactburst > stepburst
Nanoseconds per Cell
Lower is better
reactburst
threactburst
stepburst
72
71
57
20
13.9
3.8
16.3
8.0
2.8
14.7
7.6
2.6
15.6
7.4
2.5
N=128
N=512
N=2048
N=4096
N=8192

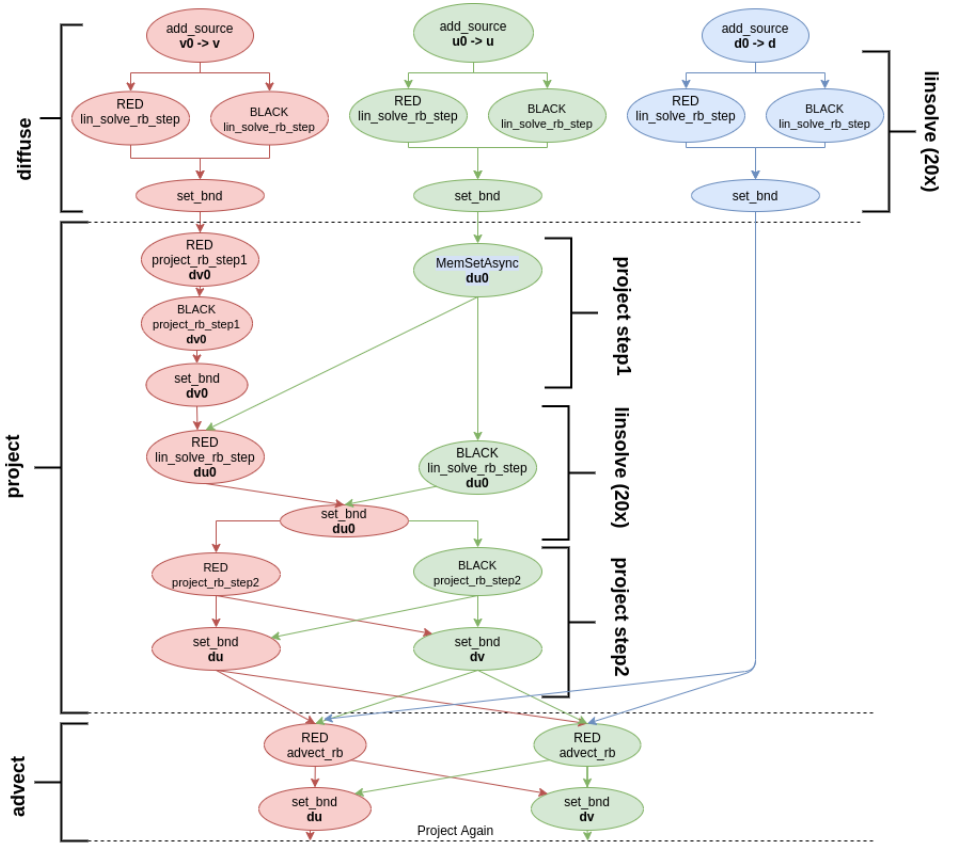
Failed Versions

<rant>

[1 / ?] Failed Versions

- `fullburst` (experimental): thrust blocking, limited kernel queue, cub non-blocking -
- `streamburst` (experimental): new stream per launch, unstable, counterproductive workload (~2/~3 result oscillation)

[2/?] graphburst (implemented)



- Use streams but also cuda Graphs.
- Idea: Reorganize and trace which kernels can be executed overlapped.
- Memset, instead of *project_rb_step1* and *set_bnd* (Green).

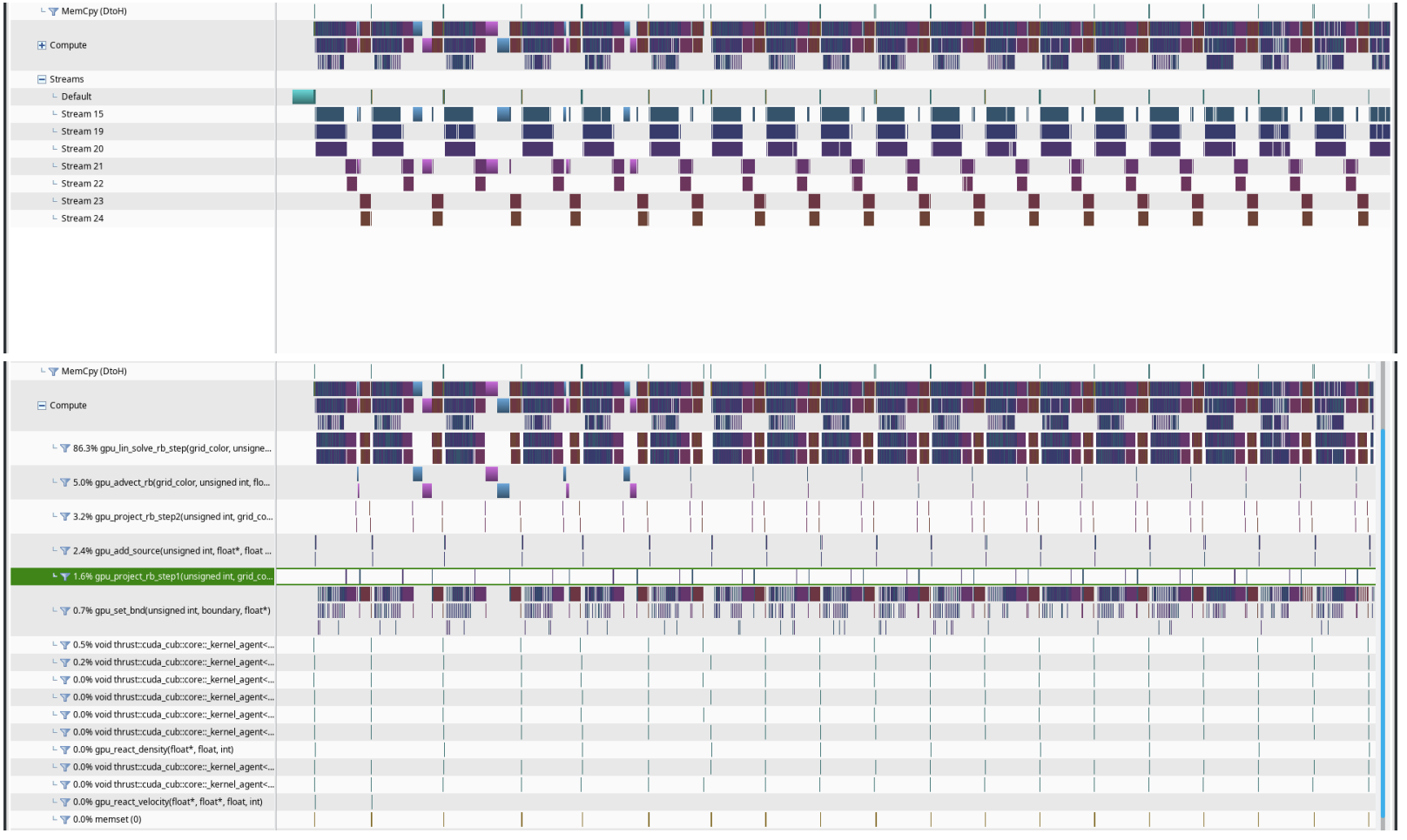
graphburst

```
cudaGraph_t graph;
create_stream_events();
checkCudaErrors(cudaStreamBeginCapture(stream0,cudaStreamCaptureModeGlobal));
step(
  N, diff, visc, dt,
  dd, du, dv, dd_prev, du_prev, dv_prev,
  stream0, stream1, stream2,
  spread, join_stream0, join_stream1, join_stream2
);
checkCudaErrors(cudaStreamEndCapture(stream0, &graph));
checkCudaErrors(cudaGraphInstantiate(&step_graph, graph, NULL,NULL, 0));
cudaGraphDestroy(graph);
```

- Capture a graph by means of *cudaStreamBeginCapture* and *cudaStreamEndCapture*.
- Step here is not launched. It is just used to define the graph.
- It needs a main stream which commands over the others (RED bubbles).
- Thrust runs over the null stream.

```
one_step(...){
  react(...);
  checkCudaErrors(cudaGraphLaunch(step_graph, stream0));
  ...
}
```

graphburst



- 86% of the entire program is linsolve. More than 3 streams!

[3/?] onekernel (implemented)

- Reuse kernel state (L1 and possibly shared memory)
- Cooperative Groups

```
__device__ gpu_lin_solve_rb_step(...);
__global__ gpu_lin_solve(...) {
  for (int k = 0; k < 20; k++) {
    gpu_lin_solve_rb_step(RED, ...);
    gpu_lin_solve_rb_step(BLACK, ...);
    grid.sync();
  }
}
```

- `grid.sync()` not as optimized in hardware as kernel sync.
- From ~2.5 to ~3.5 with N=4096, in some settings to ~500.
- Unsynced possibility? (~2.5 to ~1.6)
- Occupancy API
- `cudaDeviceSetCacheConfig`

[4 / ?] More Failed Version

- `stepburst-shmem` (implemented): try to use shared memory, smart L1
- `onekernel-shmem` (experimented): sub grid, does not scale good, conditional to where to read, grid.sync() still slower
- `stepburst-occupancy`: occupancy api, 68 SM, lanes unused

[5 / ?] stepburst-roae (implemented)

Root of all Evil It applies and discuss lots of minimal and mostly superficial optimizations

- `cudaDeviceSetCacheConfig`: Hint for increased cache usage as we couldn't use shared memory
- Vector loads: `float4`, `int4`, and similar types are supposed to increase a bit. Tradeoff: Need to align reads and our memory layout did not allow that.
- `#pragma loop unroll`: For unrolling loops.
- `nvcc` flags: basically `-use_fast_math`.
- Block/grid dims: There were many settings which gave good results, we decided to go with 8x8 as 64 is the amount of cuda cores in one SM for the 2080 Ti.
- PTX intrinsics: There are many intrinsics for math (which were already enabled by the nvcc flags) and for memory load/store.
- Similar approach as to the *stream* technique we used in lab2 with *__stwt* but it didn't give any gains.
- *__ldg* intrinsic which hints that a load should be cached in a read-only faster cache, it improved about 2% the performance in our pascal GTX 1060 MaxQ but nothing on the target RTX 2080 Ti.

[5 / ?] stepburst-shidden (experimental)

- In practice the driver does not clean shared memory between calls and just sets it unused.
- Idea: Reuse "undefined" shared memory between kernel launches.

```
void lin_solve(...) {
  gpu_lin_solve_rb_step_shtore(RED);
  gpu_lin_solve_rb_step_shtore(BLACK);
  for (int k = 0; k < 19; k++) {
    gpu_lin_solve_rb_step_shload(RED);
    gpu_lin_solve_rb_step_shload(BLACK);
  }
}
```

stepburst-shidden

```
__global__
void gpu_lin_solve_rb_step_shtore(...) {
  __shared__ float csame0[BLOCK_HEIGHT][BLOCK_WIDTH];
  __shared__ int bx, by, id1, id2;
  // Save block information for later discovery
  if (first thread of block) {
    bx = blockIdx.x; by = blockIdx.y;
    id1 = bx * by + bx; id2 = bx * by + by;
  }
  // Save appropiaty value of same0 for next kernels
  const float previous = same0[index];
  csame0[threadIdx.y][threadIdx.x] = previous;
  ... // regular lin_solve_rb_step using previous instead of same0
}
```

```
__global__
void gpu_lin_solve_rb_step_shload(...) {
  __shared__ float csame0[BLOCK_HEIGHT][BLOCK_WIDTH];
  __shared__ int bx, by, id1, id2;
  float previous = id1 == bx * by + bx && id2 == bx * by + by // Undefined vars
    ? csame0[threadIdx.y][threadIdx.x] // Found'em!
    : same0[index]; // Regular read to global memory
  ... // Regular lin_solve_rb_step but using bx and by as block indexes instead
}
```

stepburst-shidden

- What is the hit ratio in which this check effectively "finds" what is looking for?

GPU	N <= 64	N = 512	N = 2048
GTX 1060 MaxQ	100%	50%	50%
RTX 2080 Ti	100%	25%	2.21%

- Pascal vs Turing
- The real implementation has some catches

Failed Version: stepburst-tex

stepburst-tex

- The cuda programming guide and lots of posts praise the use of texture memory. Especially for stencil patterns.
- Idea: Use of texture cache instead of reading from global memory.

```
texture<float, cudaTextureType2D, cudaReadModeElementType> tex_dred;

// Bind the texture @tex_dred to the array @dred.
cudaChannelFormatDesc channel_desc = cudaCreateChannelDesc<float>();
checkCudaErrors(cudaBindTexture2D(NULL, &tex_dred, dred, &channel_desc, width, n,

// Gauss-Seidel
for (unsigned int k = 0; k < 20; ++k) {
  // Red case is launched both since we can not pass textures by parameter.
  gpu_lin_solve_rb_step_tex<<<grid_dim, block_dim>>>(RED, ...)
  gpu_lin_solve_rb_step_tex<<<grid_dim, block_dim>>>(RED, ...);
  gpu_set_bnd<<<div_round_up(n + 2, block_dim.x), block_dim.x>>>(n, b, dd);
}
cudaUnbindTexture(tex_dred);
```

- Textures are declared in file scope (In order to be known at compile time).
- They can not be passed by parameter.
- We might use texture objects but it needs the use of cuArrays.

stepburst-tex

```
__global__
void gpu_lin_solve_rb_step_tex(...){
  ...
  for (int y = 1 + gtidy; y <= n; y += grid_height) {
    for (int x = start + gtidx; x < width - (1 - start); x += grid_width) {
      int index = y * width + x;
      same[index] = (same0[index] + a * (
          tex2D(tex_dred, x, y - 1) +
          tex2D(tex_dred, x - start, y) +
          tex2D(tex_dred, x - start + 1, y) +
          tex2D(tex_dred, x, y + 1)
      )) / c;
    }
  }
}
```

- Results were not changed. Turing mershes the cache L1 and the texture cache so kernel launches used both caches.
- Use of textures in advect to take the most of *interpolations*.
- Issue: We need indexes to know if a cell is RED or BLACK.

Honorable Mentions

Honorable Mentions

- During the development with CUDA, lots of appealing options and names appeared and so we thought it could be a good idea to mention some of them.

Thrust vs CUB vs ModernGPU

- Thrust: focuses on ease of use, and high level abstractions that run not only on CUDA but also on OpenMP, OpenACC and others. Thrust uses CUB at its core.
- CUB: on the other hand is a specific, performance oriented library for CUDA, it is more cumbersome to use but it is more flexible as well. We particularly notice the ability of CUB to do reductions on device while thrust forces you to bring the result to host.
- ModernGPU: Is more of a learning oriented library that is made to be read and not so much used, the codebase prioritize ease of understanding and novel ideas to get inspired from.

CPU-GPU Implementation

- We did not get to implement an heterogenous version which uses both OpenMP and CUDA, but we thought about it a lot.
- The rough idea of implementation would've been to reuse our OpenMP strip-divided version delimiting the grid with a particular row index.
- In each step the CPU and GPU need to interchange the topmost (or bottommost) row to the other and continue its work.

Turing Specialized Hardware

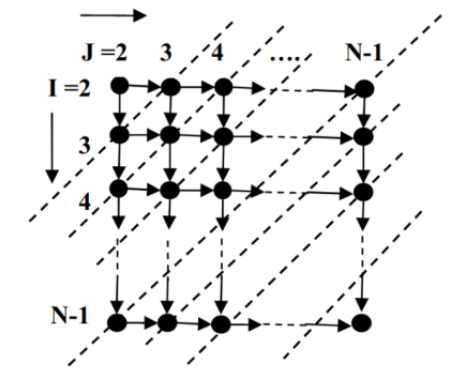
- Hardware that we considered to use but did not really found a good application for them.
- Tensor Cores: Are special functional units in each SM (8 of them in each SM) dedicated to 4x4 matrix fused multiply and add.
- RT Cores: Are a piece of dedicated hardware found in each SM that solves two problems related to ray tracing: ray casting (intersection test) and boundary volume hierarchy traversal (BVH).

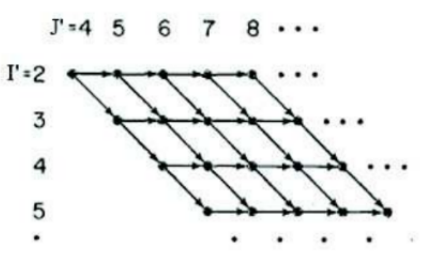
cuSOLVER

- A library that implements many linear solvers in CUDA.
- There was no explicit "Gauss-Seidel Relaxation" solver and our numerical analysis knowledge did not help much helping us to recognize which solver would've been of use in our case.

Loop Skewing

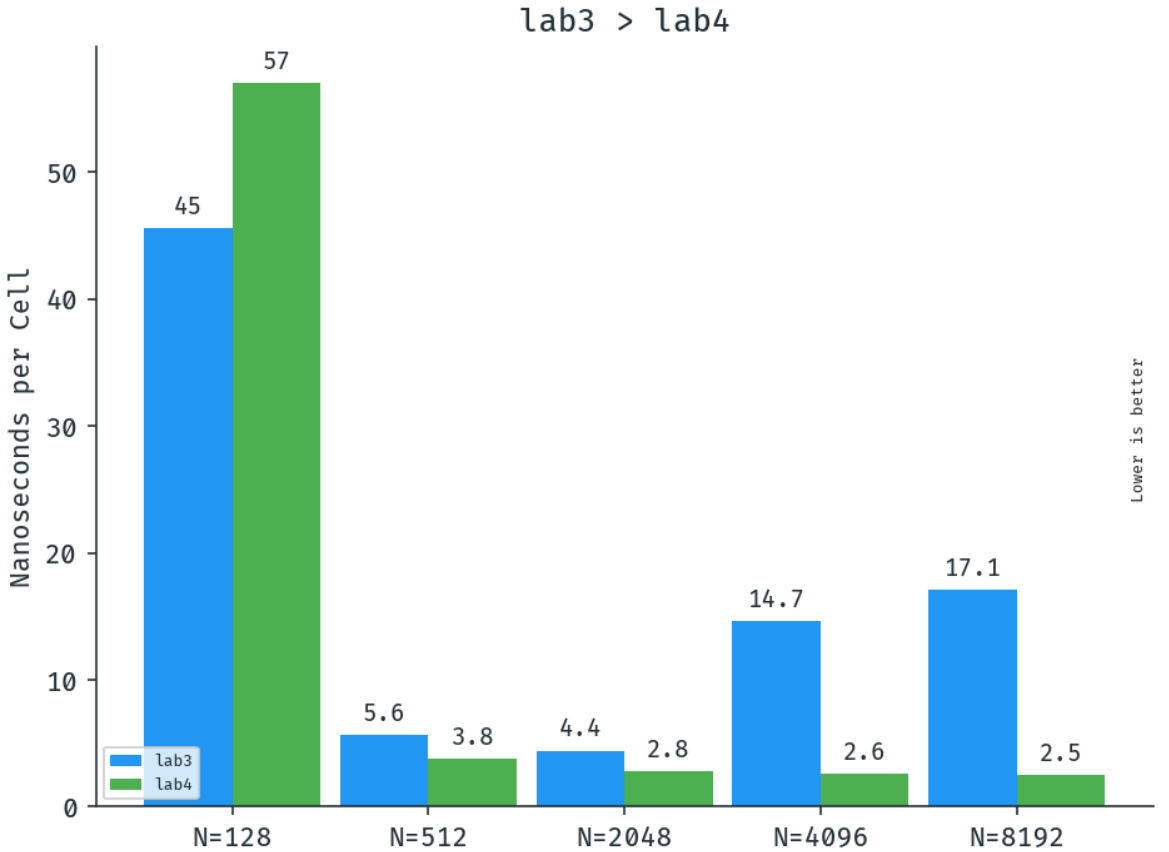
- Loop or Time Skewing algorithms for solving stencil.
- It was called the *Wavefront Pattern* and it was a concept we bumped into many times on our searches.
- In our gauss-seidel traversal, if you consider the dependencies of each cell, you will notice that diagonal cells do not have any dependency between them.
- Diagonal cells can be computed in parallel, having each iteration of a parallel loop traverse *(like wavefronts)* from the upper left cell in diagonals to the bottom right cell in the following way is a correct parallel implementation.



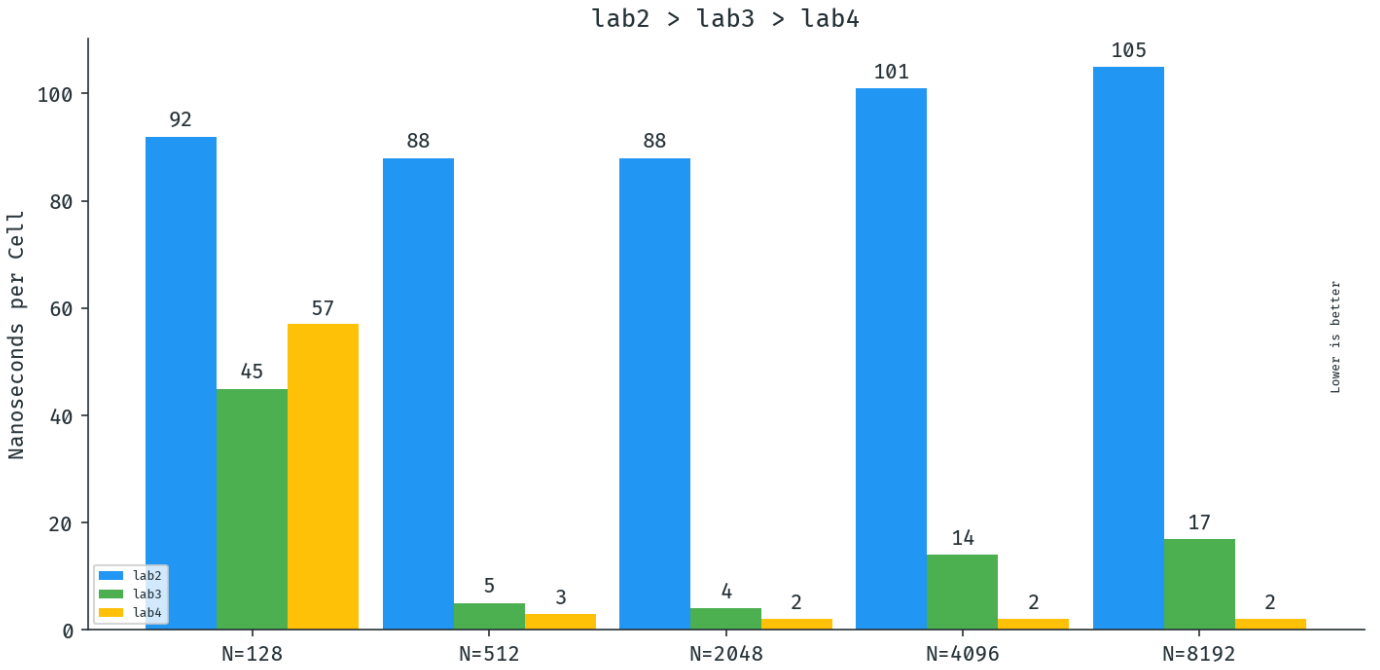


Conclusions and Final Results

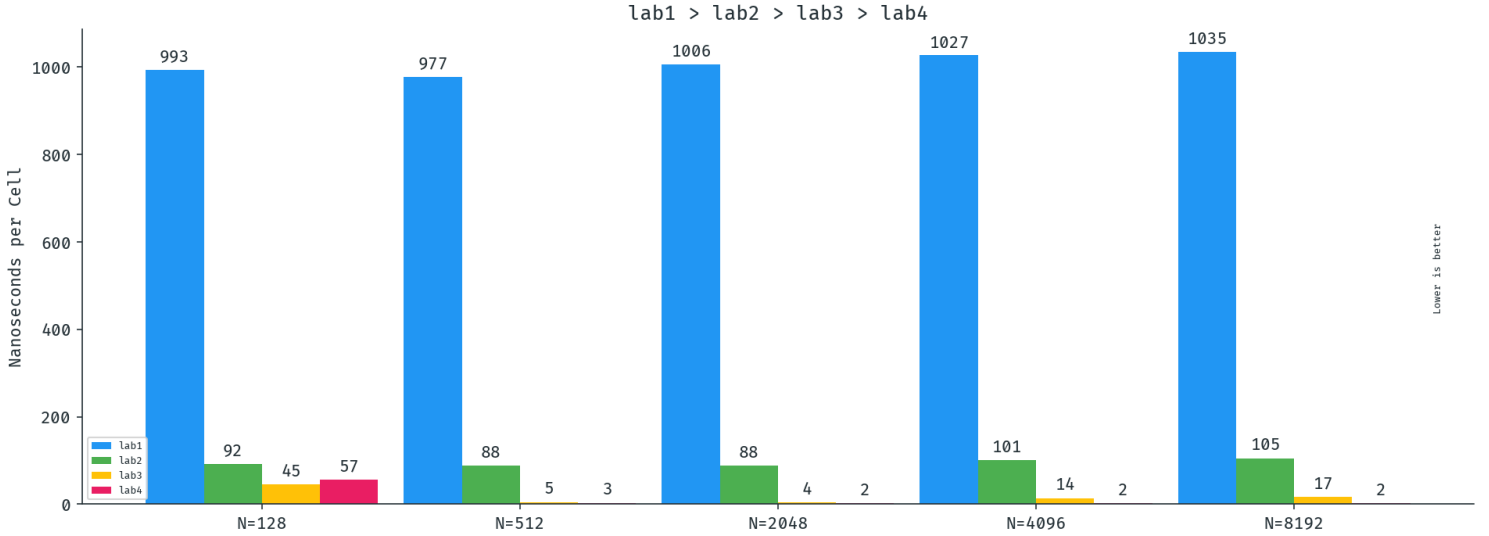
lab3 vs lab4


lab3 > lab4
Nanoseconds per Cell
Lower is better
lab3
lab4
0
10
20
30
40
50
45
57
5.6
3.8
4.4
2.8
14.7
2.6
17.1
2.5
N=128
N=512
N=2048
N=4096
N=8192

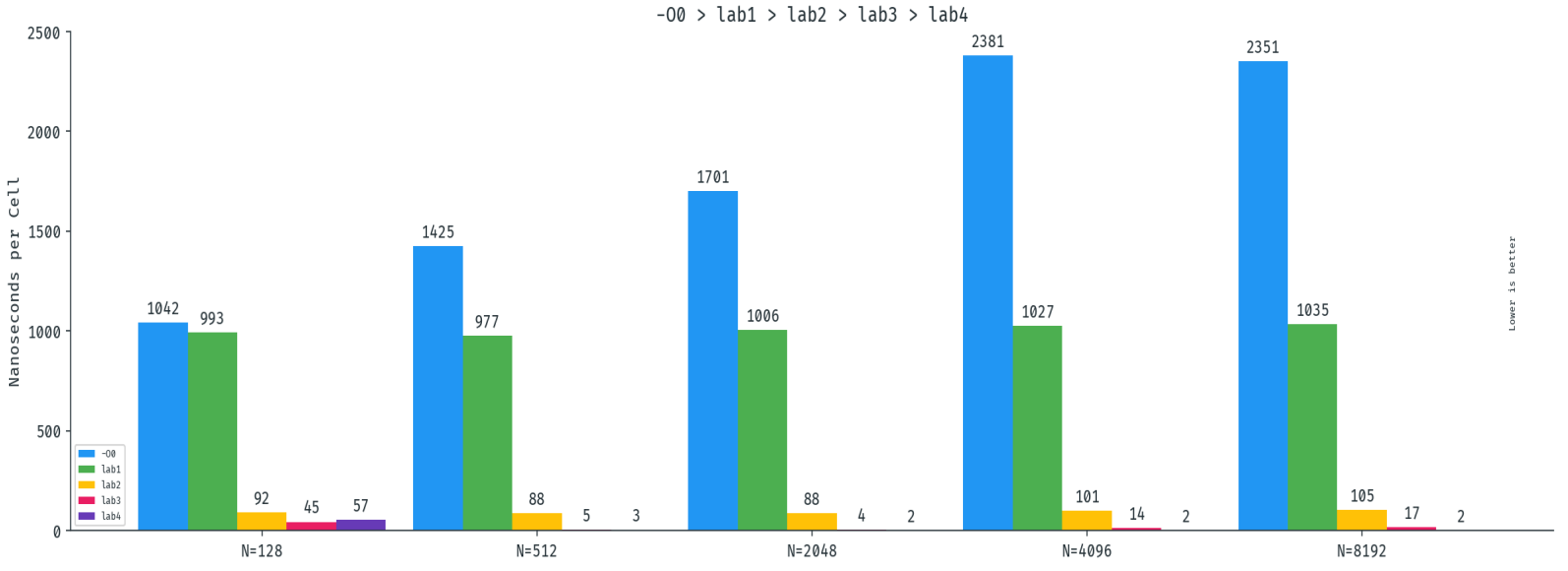
lab2 vs lab3 vs lab4



lab1 vs lab2 vs lab3 vs lab4



Without CP vs lab1 vs lab2 vs lab3 vs lab4



Speedup Table

- Speedup with respect the previous versions (starting with -O0).

N	lab1	lab2	lab3	lab4	total
128	1.05x	10.8x	2.04x	0.79x	18x
512	1.46x	11.1x	15.7x	1.47x	375x
2048	1.69x	11.4x	20.0x	1.57x	608x
4096	2.32x	10.2x	6.87x	5.65x	916x
8192	2.27x	9.85x	6.14x	6.84x	940x

Thank You