Playing Chicken with compressible flows:

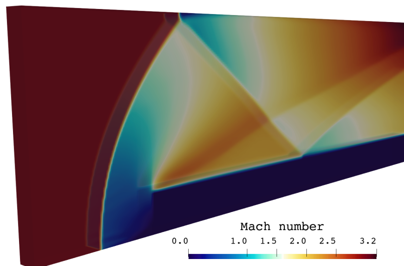
Re-implementing the core of Eilmer's transient-flow solver in CUDA

Peter Jacobs

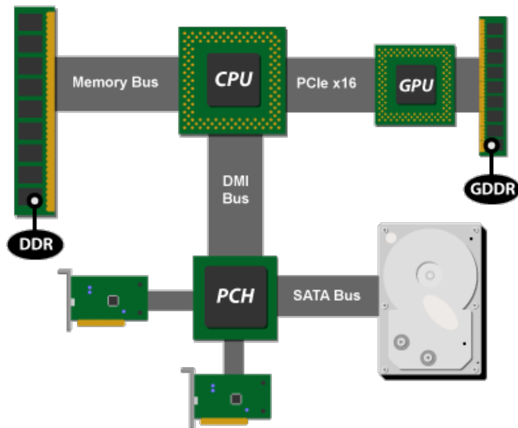
The University of (Southern) Queensland

08 June 2023

Multi-processing
The Chicken Code
Applications

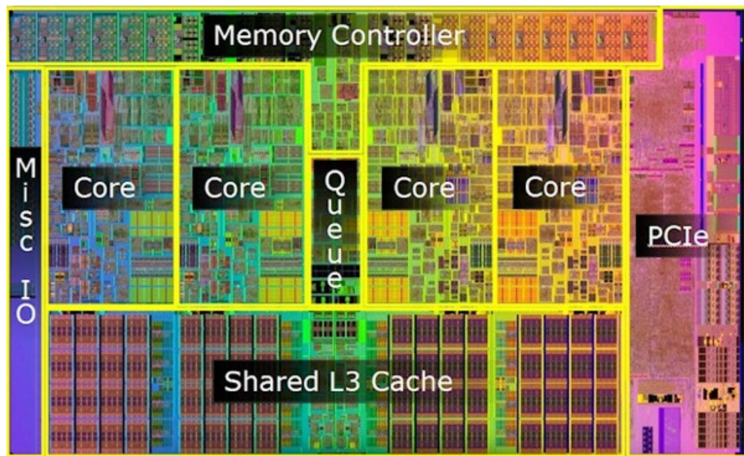


The (not so) modern workstation



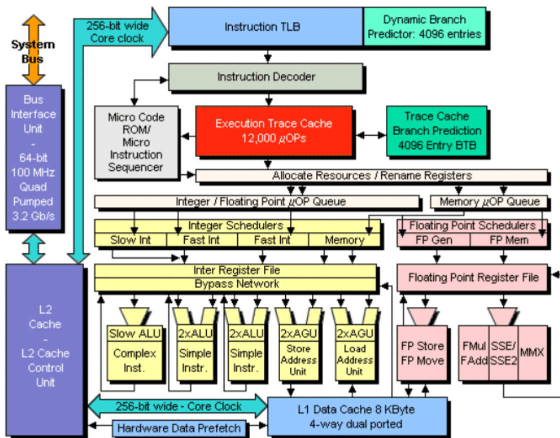
- My Dell workstation is a 2012 model.
- PCH=Platform Controller Hub
- DDR=Double Data Rate memory

Layout of the CPU on silicon



https://arstechnica.com/gadgets/2009/09/intel-launches-all-new-pc-architecture-with-core-i5i7-cpus/

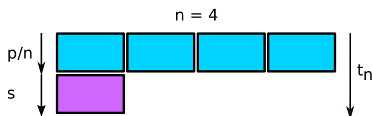
x86 Core processor



https://mechanical876.blogspot.com/2019/11/x86-processor-architecture.html

- It all feeds the floating-point functional units... fast.
- There are 4 processors in my workstation, so lets run the calculations in parallel.

Scaling of run times when using multiple processors

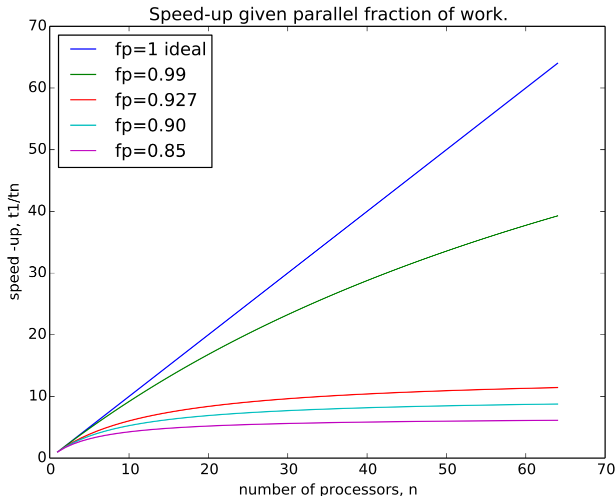



```
/* parallel fraction calculation, pj, 2016-05-24 */
eq0: p + s = t1;
eq1: p/na + s = ta;
eq2: p/nb + s = tb;
solve([eq0, eq1, eq2], [t1, p, s]);
```

```python
# fparallel.py
# Compute fraction of work done in parallel.
na = 3; ta = 3214.0-518.0
nb = 7; tb = 1904.0-453.0
p = -(na*nb*ta - na*nb*tb)/(na - nb)
s = (na*ta - nb*tb)/(na - nb)
t1 = p + s
fp = p/t1
print("p=", p, "s=", s, "t1=", t1, "fp=", fp)
```

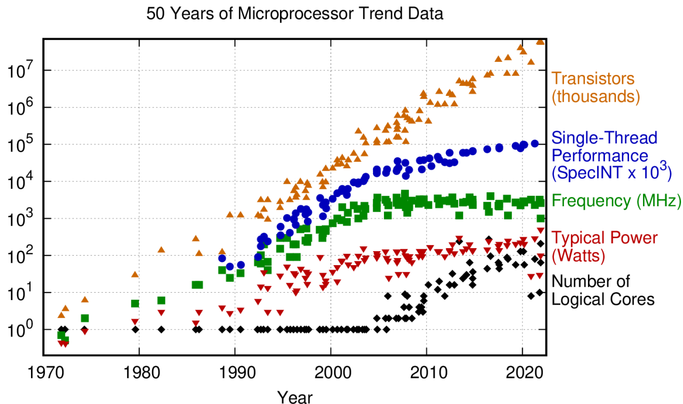
- Amdahl's model for serial and parallel work components with n processors.
- p calculations/work can be done independently by n processors
- s calculations/work needs to be done by a single processor

Amdahl's scaling for parallel calculations



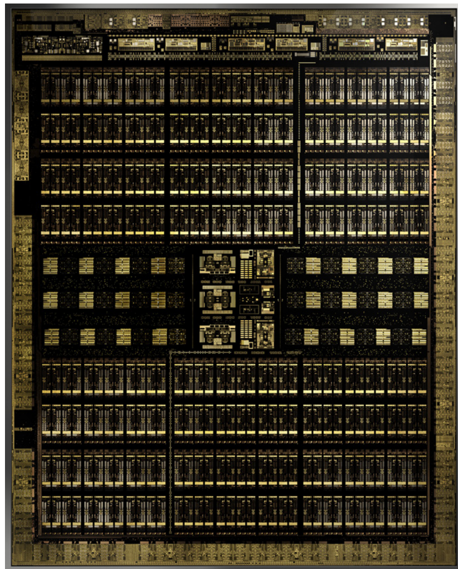
- Estimated fp=0.99987 for Eilmer4 NK variant.
- With 64 cores, expect speed-up t1/tn=63.5.

Microprocessor trend data



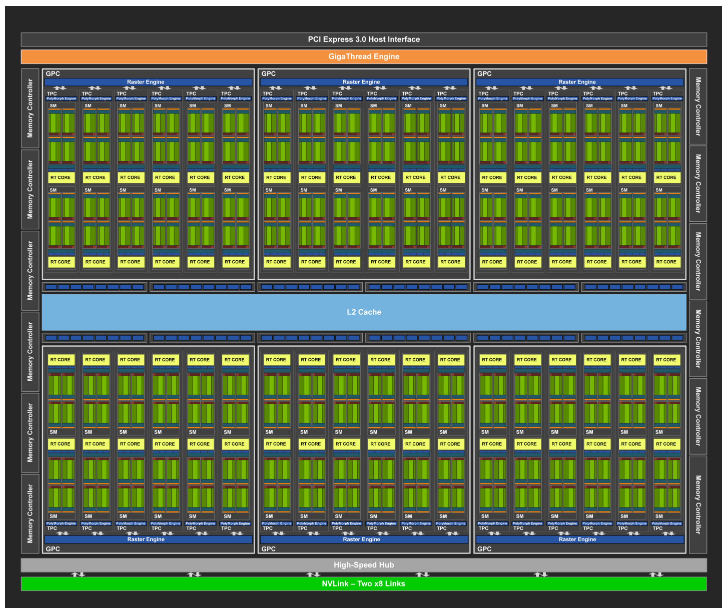
- Epyc 7601, introduced 2017.06 with 19.2×10^9 transistors
- M1 Max, introduced 2021.08 with 57.0×10^9 transistors
- Radeon MI 250, introduced 2021.09 with 58.2×10^9 transistors
- https://github.com/karlrupp/microprocessor-trend-data

Layout of the Graphics Processing Unit (GPU) on silicon

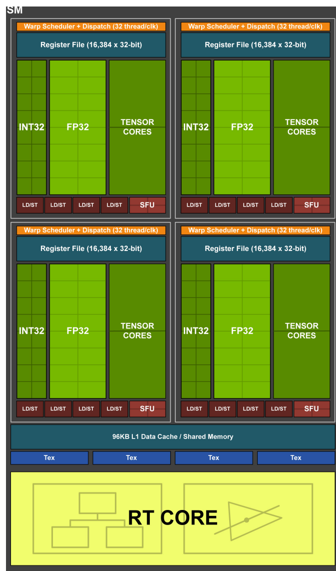


- TU102 GPU, Source: NVIDIA Turing Architecture Whitepaper
- 6 Graphics Processing Clusters (GPCs)
- 12 Streaming Multiprocessors (SMs) per GPC
- 64 CUDA cores per SM (4608 for the GPU)
- Extrapolating Eilmer speed-up t1/tn=2882.
- A$249 buys a GTX 1650 graphics card with a Tu117 GPU and 896 CUDA cores.

Layout of the GPU – Top-level structure



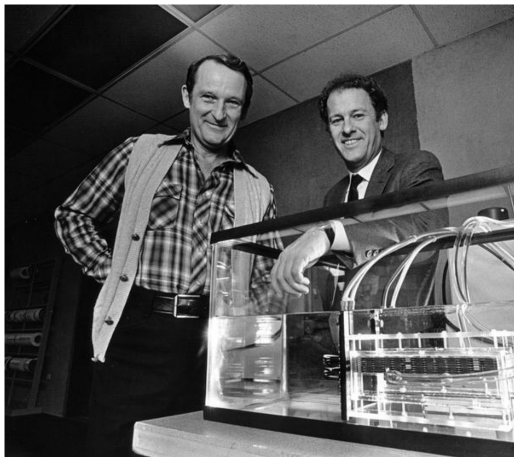
Layout of the GPU – The Streaming Multiprocessor

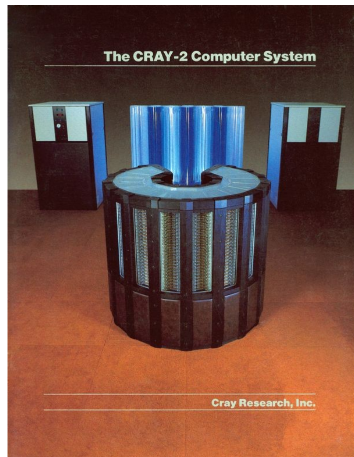


- 64 CUDA cores per SM with a cache of shared memory.
- The cores have integer and floating-point functional units.
- Register file is local memory for the threads (but limited to 64kB)
- Warp scheduler runs sets of 32 threads of code (warps) in parallel on each SM.
- Many more threads are scheduled to run concurrently.
- It is cheap to swap warps (sets) of 32 threads if they get stalled, waiting for data to be delivered from global memory.

HPC God Fathers – Seymour Cray and John Rollwagen

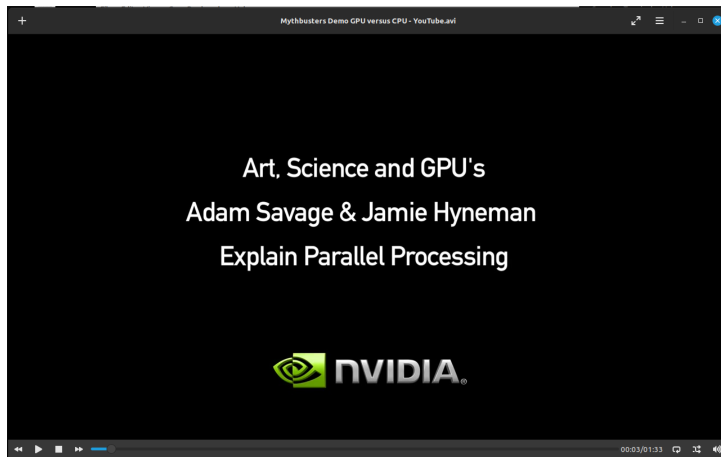
Do you want your wagon pulled by 4 hefty bullocks or 1024 chickens?





27 Cray-2 computers were built and sold.
https://www.computerhistory.org/revolution/supercomputers/10/68/273
https://www.computerhistory.org/revolution/supercomputers/10/68/86

Free-piston-driven GPU



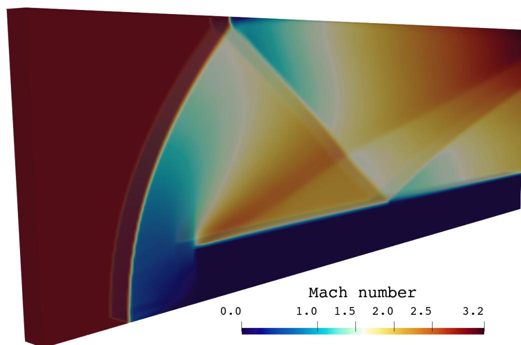
- NVIDIA, 2009

GPU activities in recent years

https://tuckdbpostcards.org/items/35560 (Artist: Wally Fiakowska)

- GPU-powered chemical-update in Eilmer (Kyle Damm).
- Shocked flows with Walsh functions (Jamie Border).
- *Spatz*, a Cartesian-grid based flow solver (Christine Mittler).
- *Chicken*, port the “traditional” gas-dynamics core of Eilmer.

The *Chicken* compressible-flow code



- Time-marching, shock-capturing, laminar, 3D flow.
- Ideal gas model with simple combustion.
- Design goal: simple enough code to write in a month or two.
- To use thousands of cores, we have to partition the calculations into small, independent parcels of work.

Gas-dynamic equations

Conservation statements for mass, momentum and energy determine what happens next:

$$\frac{\partial}{\partial t}\int_V U dV = -\oint_S \overline{F}_c \cdot \hat{n}\ dA + \int_V Q dV \ ,$$

For a two-species (α, β) gas, the array (U) of conserved quantities and convective flux vectors (F) in 2D are

$$U = \begin{bmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho E \\ \rho Y_\beta \end{bmatrix} \ , \overline{F}_c = \begin{bmatrix} \rho v_x \\ \rho v_x^2 + p \\ \rho v_y v_x \\ \rho E v_x + p v_x \\ \rho Y_\beta v_x \end{bmatrix} \hat{i} + \begin{bmatrix} \rho v_y \\ \rho v_x v_y \\ \rho v_y^2 + p \\ \rho E v_y + p v_y \\ \rho Y_\beta v_y \end{bmatrix} \hat{j} \ ,$$

Conserved quantities in the U vector are per unit volume.
Total specific energy is $E = u + \frac{1}{2}v^2$.
The chemical reactions appear as source terms for Y_β and energy.

Flux calculators

To compute the flux of mass, momentum and energy at each face, we work in the local coordinate frame of the face, $\hat{n}$ is the unit normal, $\hat{t}$ is the corresponding unit tangent.

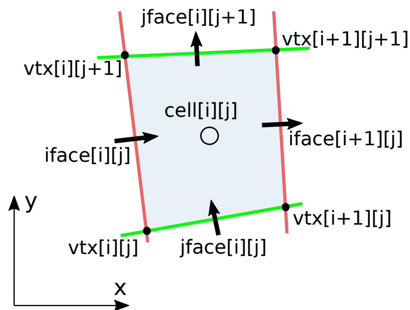
We usually think of the flux calculator as incorporating some approximate solution to the Riemann problem for the interaction of a left (L) flow state and a right (R) flow state, with the flux values being estimated at the initial location of the interface.

The flux for a uniform supersonic flow is

$$\overline{F}_c = \begin{bmatrix} \dot{m} \\ \dot{m}\, v_n + p \\ \dot{m}\, v_t \\ \dot{m}\,(E + p/\rho) \\ \dot{m}\, Y_\beta \end{bmatrix} \hat{n}$$

where $\dot{m} = \rho\, v_n$ is the mass flux.

Cells, faces, vertices and indexing



vertex: geometric location

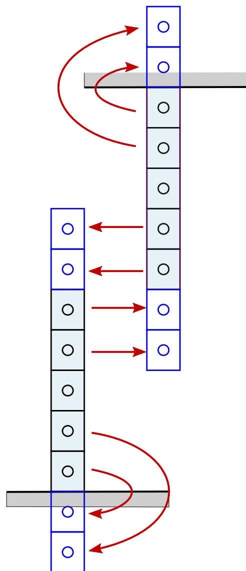
face:

- Defined by vertices at ends.
- We calculate fluxes of mass, momentum and energy across each face.

cell:

- Defined by bounding faces.
- Flow state associated with cell centre.
- Each finite-volume cell is the basic unit over which the gas-dynamic equations are applied.

Boundary conditions, multiple blocks



The picture at left shows strips of cells within two blocks. They are aligned even though they are shown offset.

- Each block has two "ghost" cells associated with each end of the strip.
- Convective boundary conditions are implemented by copying appropriate data into the ghost cells. Red arrows indicate data transfer.
- An adjacent block interacts through a simple copy of the flow data.
- There is a fixed ordering of the blocks (in the i, j and k index directions).
- Interaction with a solid wall involves reflection of the flow data in the frame of the boundary face.

Overview of the calculation process

Given a discretized domain, an initial flow at $t = 0$, and a set of boundary conditions, do:

1. Set up ghost-cell flow states to effect the boundary conditions.
2. For all faces, compute flux vectors.
3. For all cells, update conserved quantities and flow states over small dt.
4. Maybe save the flow data for postprocessing.
5. Move on to time $t + dt$.
6. If $t < t_{max}$, go back to step 1.
7. We are done.

Gas-dynamic update code for CPU (only)

```
// Gas-dynamic update over three stages with TVD-RK3 weights.
int bad_cell_count = 0;
// Stage 1.
// number t = SimState::t; // Only needed if we have time-dependent source terms or BCs.
apply_boundary_conditions_for_convective_fluxes();
for (auto& bcc : bad_cell_counts) bcc = 0;
#pragma omp parallel for
for (int ib=0; ib < Config::nFluidBlocks; ib++) {
    BConfig& cfg = blk_configs[ib];
    Block& blk = fluidBlocks[ib];
    if (cfg.active) {
        blk.calculate_convective_fluxes(Config::flux_calc, Config::x_order);
        if (Config::viscous) {
            apply_viscous_boundary_conditions(blk, cfg);
            blk.add_viscous_fluxes();
        }
        bad_cell_counts[ib] = blk.update_stage_1(cfg, SimState::dt);
    }
}
for (auto bcc : bad_cell_counts) bad_cell_count += bcc;
if (bad_cell_count > 0) {
    throw runtime_error("Stage 1 bad cell count: "+to_string(bad_cell_count));
}
```

- Runs everything on the CPU.

Gas-dynamic update code for GPU

```
// Gas-dynamic update over three stages with TVD-RK3 weights.
bad_cell_count = 0;
status = cudaMemcpy(bad_cell_count_on_gpu, &bad_cell_count, sizeof(int), cudaMemcpyHostToDevice);
if (status) throw runtime_error("Stage 0, could not copy bad_cell_count to gpu.");
//
// Stage 1.
// number t = SimState::t; // Only needed if we have time-dependent source terms or BCs.
for (int ib=0; ib < Config::nFluidBlocks; ib++) {
    BConfig& cfg = blk_configs[ib];
    Block& blk = fluidBlocks[ib];
    if (!cfg.active) continue;
    Block& blk_on_gpu = fluidBlocks_on_gpu[ib];
    BConfig& cfg_on_gpu = blk_configs_on_gpu[ib];
    //
    int nGPUblocks = cfg.nGPUblocks_for_faces;
    int nGPUthreads = Config::threads_per_GPUblock;
    calculate_fluxes_on_gpu<<<nGPUblocks,nGPUthreads>>>(blk_on_gpu, cfg_on_gpu, flowStates_on_gpu, fluidBlocks_on_gpu,
                                                         Config::flux_calc, Config::x_order, Config::viscous);
    auto cudaError = cudaGetLastError();
    if (cudaError) throw runtime_error(cudaGetErrorString(cudaError));
    //
    nGPUblocks = cfg.nGPUblocks_for_cells;
    nGPUthreads = Config::threads_per_GPUblock;
    update_stage_1_on_gpu<<<nGPUblocks,nGPUthreads>>>(blk_on_gpu, cfg_on_gpu, Config::source_terms,
                                                       SimState::dt, bad_cell_count_on_gpu);
    cudaError = cudaGetLastError();
    if (cudaError) throw runtime_error(cudaGetErrorString(cudaError));
}
status = cudaMemcpy(&bad_cell_count, bad_cell_count_on_gpu, sizeof(int), cudaMemcpyDeviceToHost);
if (status) throw runtime_error("Stage 1, could not copy bad_cell_count from gpu to host cpu.");
if (bad_cell_count > 0) {
    throw runtime_error("Stage 1, bad cell count: "+to_string(bad_cell_count));
}
```

- This code runs on CPU and launches many threads of kernel functions that run on the GPU.
- Those threads are where the work is done.

Update-stage-1 code on GPU

```
__global__
void update_stage_1_on_gpu(Block& blk, const BConfig& cfg, int isrc, number dt, int* bad_cell_count)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < cfg.nActiveCells) {
        FVCell& c = blk.cells_on_gpu[i];
        ConservedQuantities dUdt0;
        c.eval_dUdt(dUdt0, blk.faces_on_gpu, isrc);
        blk.dQdt_on_gpu[i] = dUdt0; // keep
        ConservedQuantities U = blk.Q_on_gpu[i]; // U0
        for (int j=0; j < CQI::n; j++) { U[j] += dt*dUdt0[j]; }
        blk.Q_on_gpu[cfg.nActiveCells + i] = U; // keep update
        int bad_cell_flag = c.fs.decode_conserved(U);
        atomicAdd(bad_cell_count, bad_cell_flag);
        if (bad_cell_flag) {
            printf("Stage 1 update, Bad cell at pos x=%g y=%g z=%g\n", c.pos.x, c.pos.y, c.pos.z);
        }
    }
}
```

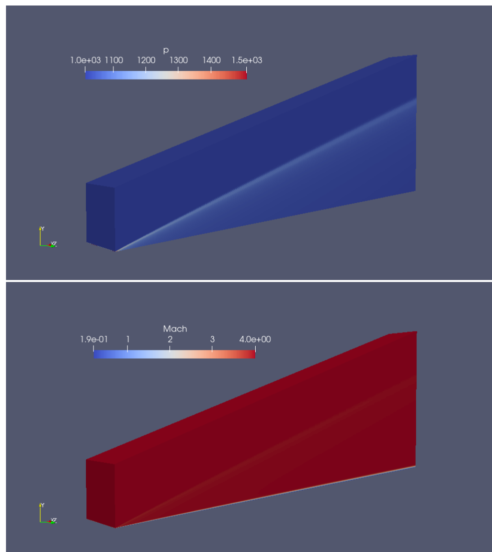
- This code is a kernel function that runs in a thread of code execution on GPU.
- It is started with thread-specific context.
- Each thread works on a single finite-volume cell.
- Calls other functions that run in the same thread on the GPU.

eval-dUdt code on GPU or CPU

```
__host__ __device__
void eval_dUdt(ConservedQuantities& dUdt, FVFace faces[], int isrc)
// These are the spatial (RHS) terms in the semi-discrete governing equations.
{
    number vol_inv = one/volume;
    auto& fim = faces[face[Face::iminus]];
    auto& fip = faces[face[Face::iplus]];
    auto& fjm = faces[face[Face::jminus]];
    auto& fjp = faces[face[Face::jplus]];
    auto& fkm = faces[face[Face::kminus]];
    auto& fkp = faces[face[Face::kplus]];
    // Introducing local variables for the data helps
    // promote coalesced global memory access on the GPU.
    number area_im = fim.area; ConservedQuantities F_im = fim.F;
    number area_ip = fip.area; ConservedQuantities F_ip = fip.F;
    number area_jm = fjm.area; ConservedQuantities F_jm = fjm.F;
    number area_jp = fjp.area; ConservedQuantities F_jp = fjp.F;
    number area_km = fkm.area; ConservedQuantities F_km = fkm.F;
    number area_kp = fkp.area; ConservedQuantities F_kp = fkp.F;
    //
    for (int i=0; i < CQI::n; i++) {
        // Integrate the fluxes across the interfaces that bound the cell.
        number surface_integral = area_im*F_im[i] - area_ip*F_ip[i]
            + area_jm*F_jm[i] - area_jp*F_jp[i] + area_km*F_km[i] - area_kp*F_kp[i];
        // Then evaluate the derivatives of conserved quantity.
        // Note that conserved quantities are stored per-unit-volume.
        dUdt[i] = vol_inv*surface_integral;
    }
    //
    if (isrc != SourceTerms::none) add_source_terms(dUdt, isrc);
    return;
} // end eval_dUdt()
```

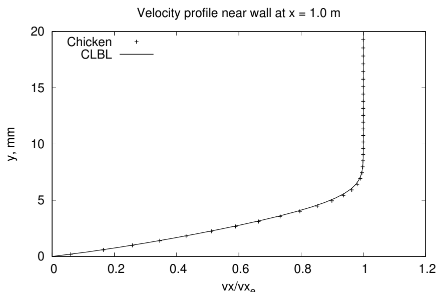
- Used for both CPU and GPU flavours of *Chicken*.

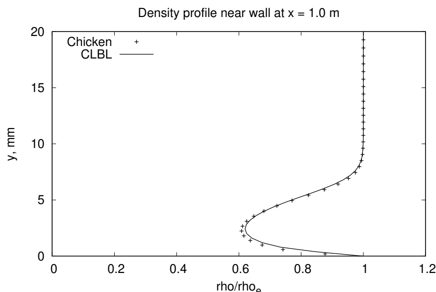
Flow along a flat plate - 1



- Mach 4 free stream, 1.1 m plate, 440 mm high domain.
- $220 \times 2 \times 192 = 84480$ cells
- At x=1 m, δ=3.4 mm
- 198 ns/cell/update-stage on GTX1650
- 14 ns/cell/update-stage on A100
- 20 μs/cell/Euler-step for 2D `cns4u` code on Cray-Y/MP back in 1991

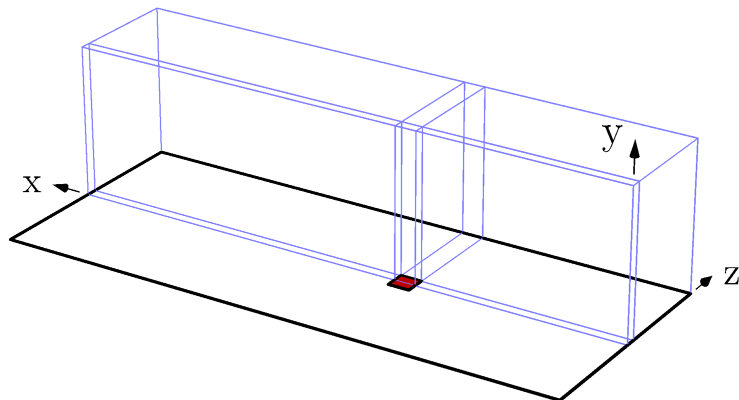
Flow along a flat plate - 2





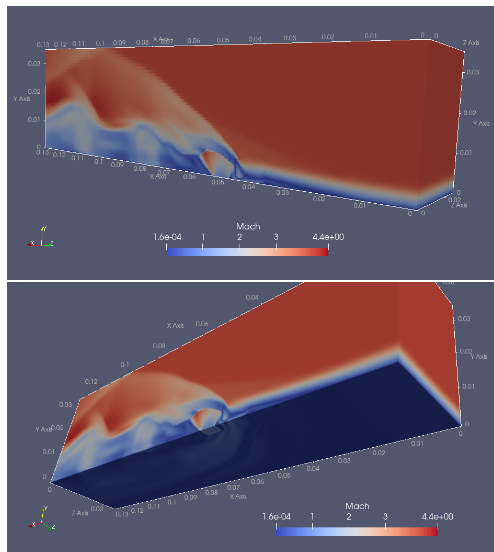
- Compare profile through boundary layer at x=1 m.
- Reference data from Schetz' compressible, laminar boundary-layer (CLBL) program.

Transverse injection into a Mach 4 flow - 1



- Mach 4 flow in x-direction, boundary-layer profile at x=0.
- Sonic injection, in y-direction, through square port.
- 6 structured-grid blocks for half of the domain.

Transverse injection into a Mach 4 flow - 2



- Inflow with boundary-layer profile from self-similar solution (Anderson's text book).
- 1,176,000 cells, 2.6GB memory allocated on CPU and GPU
- Run time 1.6 hours on GTX1650
- 199 ns/cell/update-stage

GPU hackathon team in front of Gadi

- November 2022
- NVIDIA mentor: Wei Fang