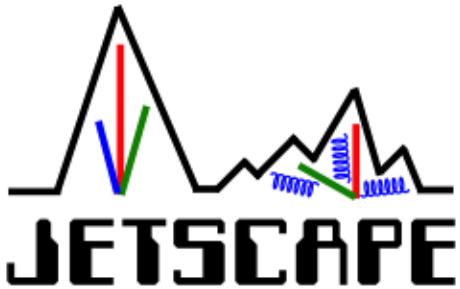
Viscous Hydro On GPU

Long-Gang Pang

UC Berkeley and LBL

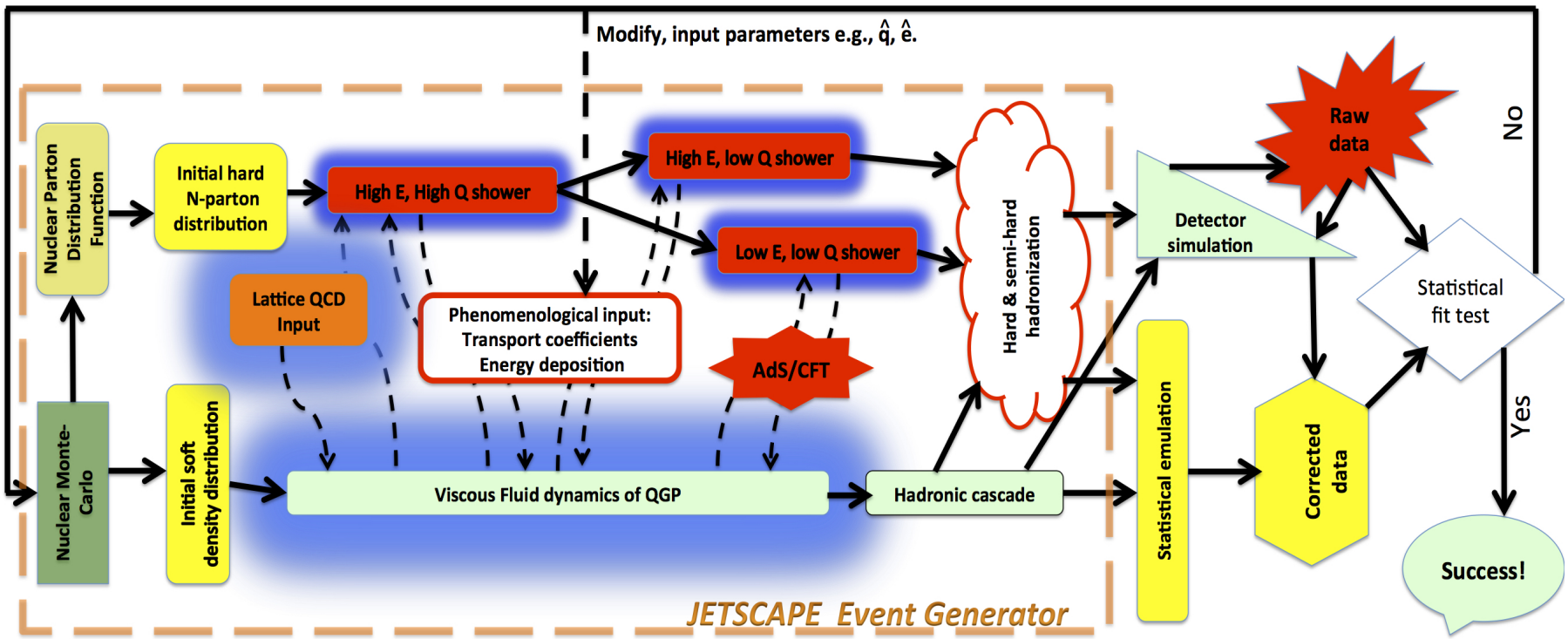
January 5th, 2018



Outline

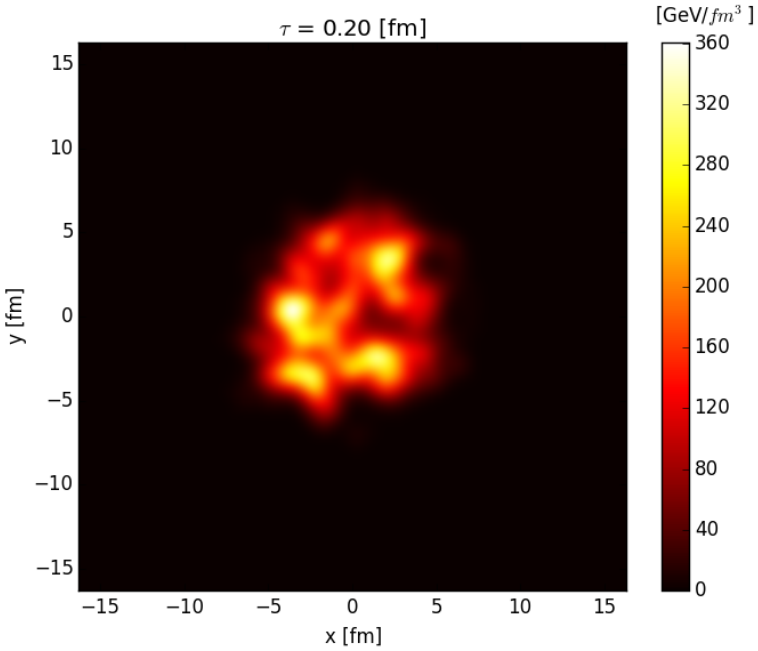
1. Motivation: computing demands
2. Introduction to GPU parallel computing
3. 3+1D viscous hydrodynamics on GPU
4. Conclusion

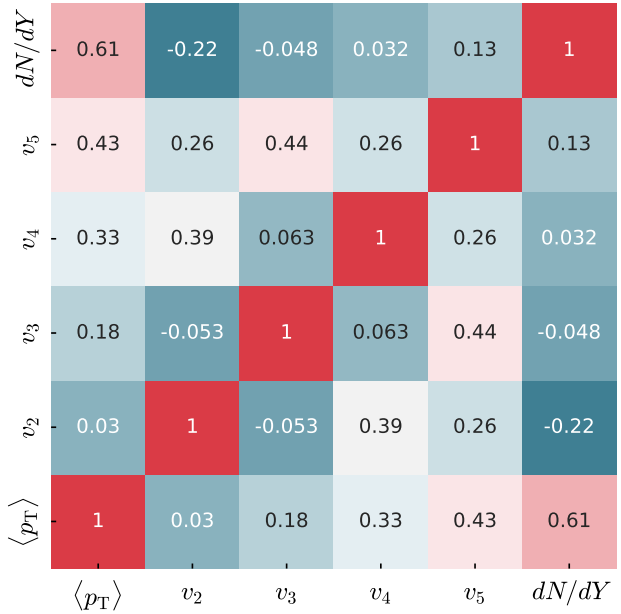
Hydro is the bottleneck in Jet + Hydro concurrent running



- Jet shower propagation takes less than 1 second CPU time per event.
- (3+1)D viscous hydrodynamics takes $O(10)$ CPU hours per event.

Big data analysis requests huge number of events





- Event-by-event simulations with fluctuating initial conditions to get: event averages, probability distributions, correlation matrix.
- Bayesian analysis needs $O(10^{5\sim7})$ viscous hydrodynamic events.
- Deep learning studies need $O(10^{5\sim7})$ viscous hydrodynamic events.

(3+1)D viscous hydrodynamic equations are complicated

$$\nabla_\mu T^{\mu\nu} = 0, \quad \nabla_\mu N^\mu = 0, \tag{1}$$

with the energy-momentum tensor $T^{\mu\nu} = \varepsilon u^\mu u^\nu - (p + \Pi)\Delta^{\mu\nu} + \pi^{\mu\nu}$, where ε is the energy density, p the pressure, u^μ the fluid four-velocity normalized as $u^\mu u_\mu = 1$ and $\Delta^{\mu\nu} = g^{\mu\nu} - u^\mu u^\nu$ the projection operator which is orthogonal to the fluid velocity, and the net charge current $N^\mu = n u^\mu + d^\mu$ where d^μ is the charge diffusion current.

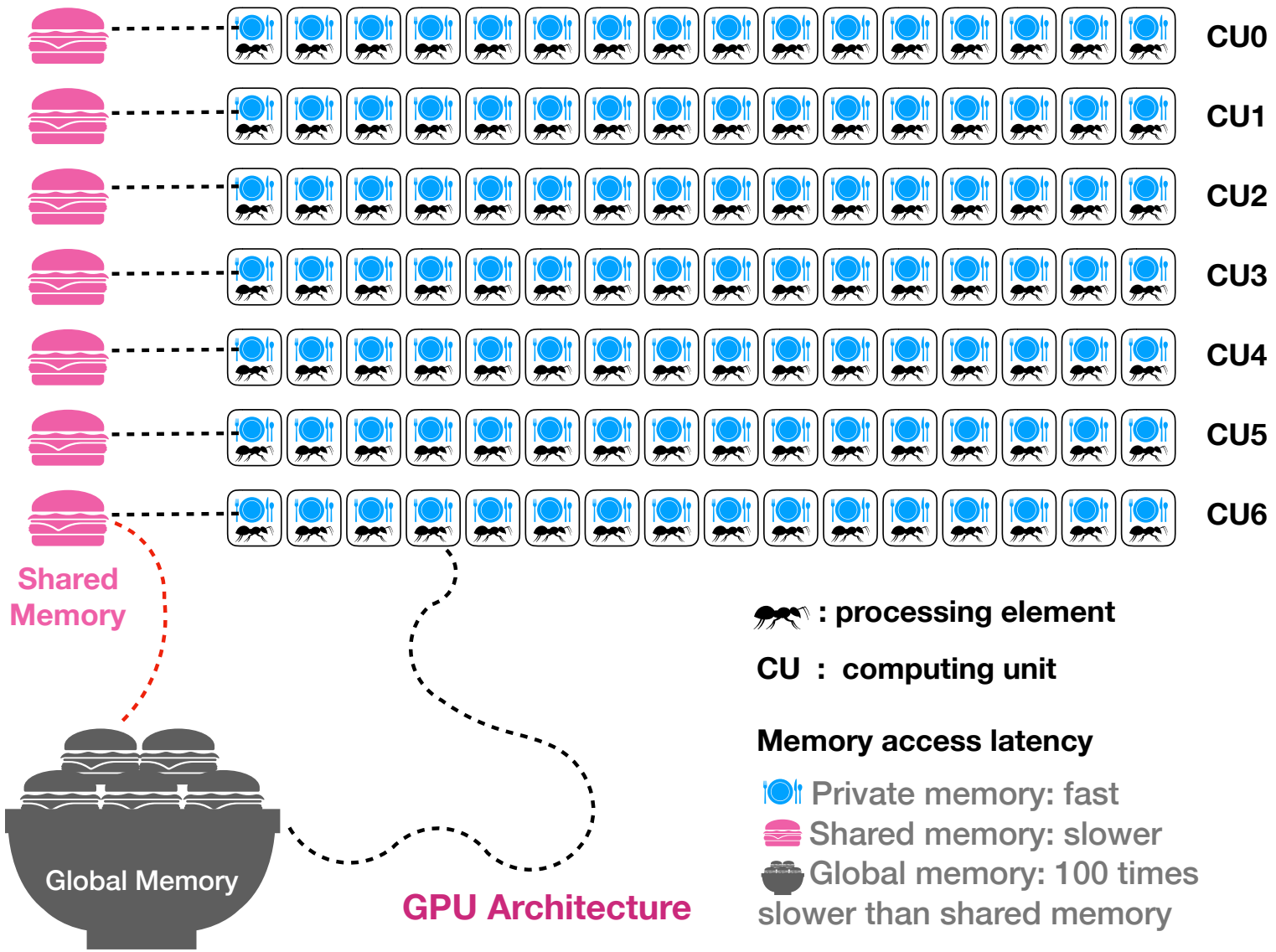
$$\Pi = -\zeta\theta - \tau_\Pi \left[u^\lambda \nabla_\lambda \Pi + \frac{4}{3}\Pi\theta \right] \tag{2}$$

$$\pi^{\mu\nu} = \eta_v \sigma^{\mu\nu} - \tau_\pi \left[\Delta^\mu_\alpha \Delta^\nu_\beta u^\lambda \nabla_\lambda \pi^{\alpha\beta} + \frac{4}{3}\pi^{\mu\nu}\theta \right] - \lambda_1 \pi_\lambda^{\langle\mu} \pi^{\nu\rangle\lambda} - \lambda_2 \pi_\lambda^{\langle\mu} \Omega^{\nu\rangle\lambda} - \lambda_3 \Omega_\lambda^{\langle\mu} \Omega^{\nu\rangle}$$

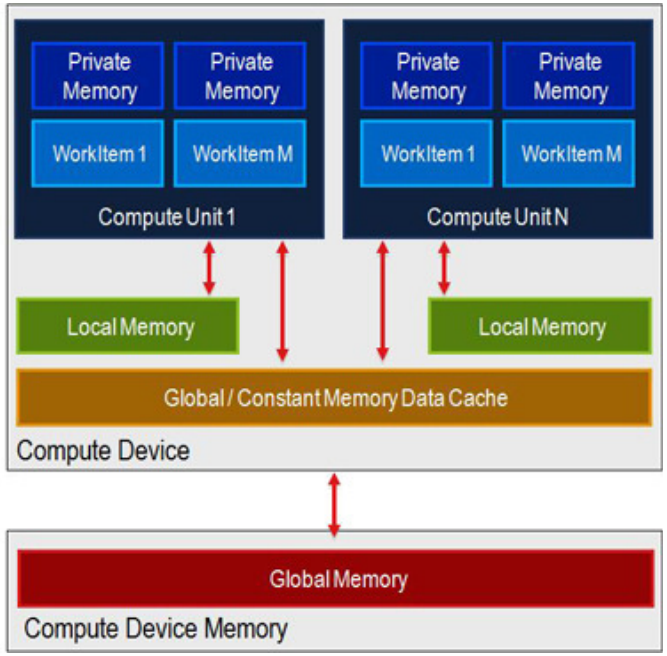
with the expansion rate θ, symmetric shear tensor $\sigma^{\mu\nu}$ and the antisymmetric vorticity tensor $\Omega^{\mu\nu}$ defined as

$$\begin{aligned}
\theta &\equiv \nabla_\mu u^\mu, \\
\sigma^{\mu\nu} &\equiv 2\nabla^{\langle\mu} u^{\nu\rangle} \equiv 2\Delta^{\mu\nu\alpha\beta}\nabla_\alpha u_\beta, \\
\Omega^{\mu\nu} &\equiv \frac{1}{2}\Delta^{\mu\alpha}\Delta^{\nu\beta}(\nabla_\alpha u_\beta - \nabla_\beta u_\alpha), \\
\Delta^{\mu\nu\alpha\beta} &\equiv \frac{1}{2}(\Delta^{\mu\alpha}\Delta^{\nu\beta} + \Delta^{\mu\beta}\Delta^{\nu\alpha}) - \frac{1}{3}\Delta^{\mu\nu}\Delta^{\alpha\beta},
\end{aligned} \tag{3}$$

Cartoon diagram of GPU architecture



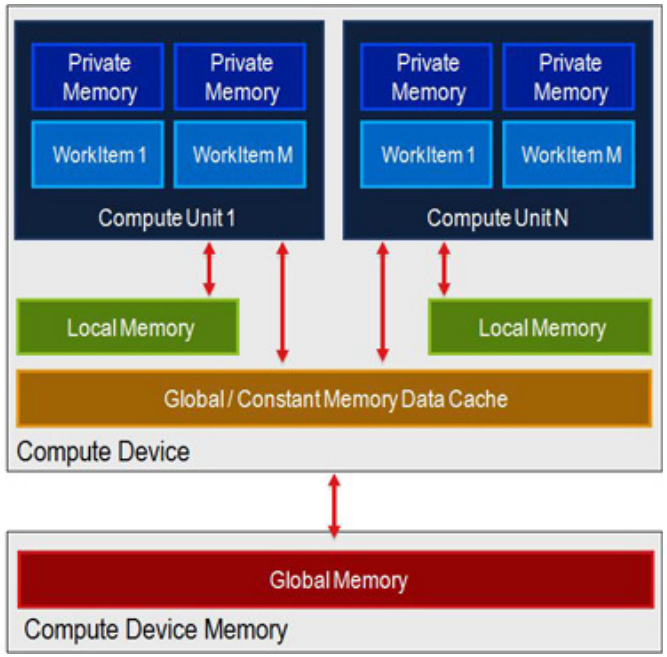
Memory access on GPU



Global Memory

- Global memory: GPU side, $1-12$ GB, speed $100-300$ GB/s, latency 400 clock cycles.
- 400 clock cycles == (400 +) or (100 *) or (20-40 square root).
- Use more workitems per workgroup to hide latency (warp switching).
- Do extra calculation other than Global memory access.
- Slowest

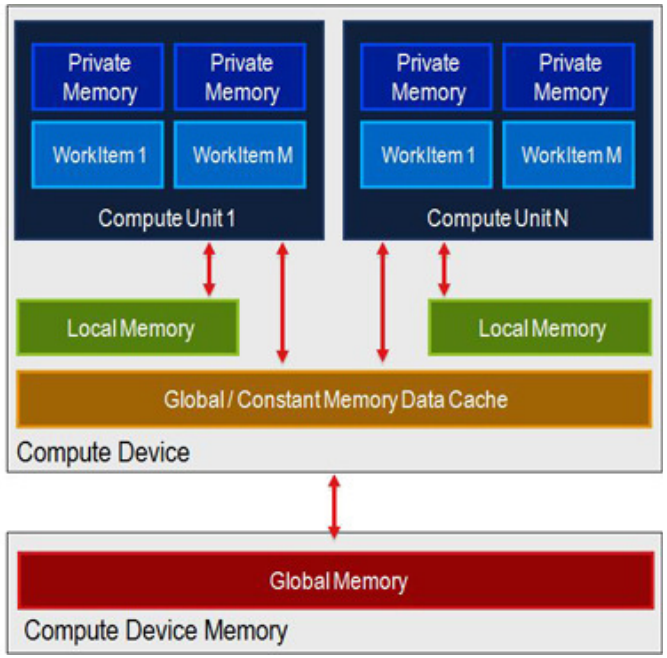
Memory access on GPU



Local Memory

- Local memory: on CU, $16 - 64KB$, speed $600 - 800$ GB/s, latency $1 - 40$ clock cycles
- Used when multi workitems in the same workgroup share data
- No data sharing, do not use local memory (slower than private memory).
- Faster

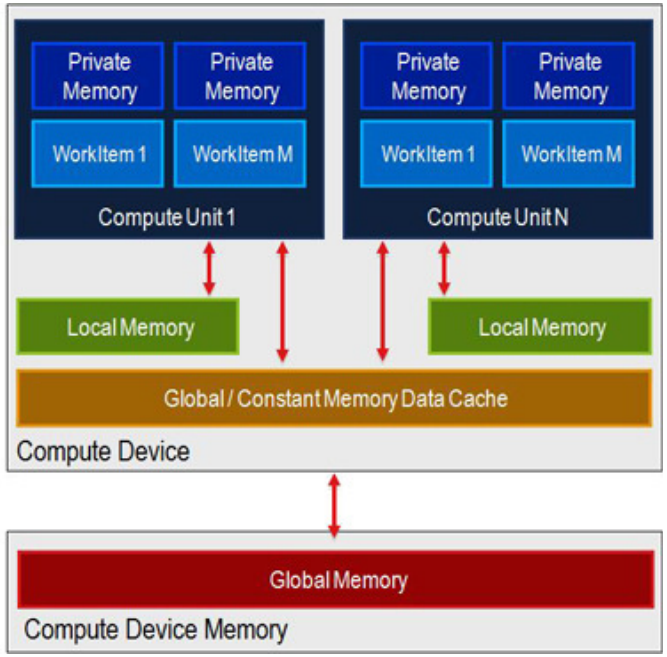
Memory access on GPU



Constant memory

- Shared by all PE on GPU.
- Fast. Speed is between global and local memory.

Memory access on GPU



Private memory

- Private memory: on PE, 16-64K per CU.
- Used if global/local/constant memory is accessed by one workitem multiple times.

Existing Code

- OpenCL-Shasta (Frankfurt Group), (3+1)D ideal hydro with simple EoS.
- CLVisc (CCNU-LBNL Group), (3+1)D viscous hydro using OpenCL.
- GPU-VH (OSU Group), (3+1)D viscous hydro using CUDA.
- Key ideas are the same – solving partial differential equations numerically can be parallelized on GPU using massive processing elements.
- This talk will focus on the OpenCL implementation CLVisc.

OpenCL (Used in CLVisc)

Open Computing Language (OpenCL) from wikipedia

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors.

- Open Standard maintained by Khronos org.
- Host language: C/C++/Python/Julia/Java.
- Device language: C99 (subset)

Example program on GPU using OpenCL: vector addition

CPU version

```
void vectorAdd(float *a, float *b, float *c, int N)
{
    for ( int i = 0; i < N; i++ ) {
        c[i] = a[i] + b[i];
    }
}
```

Example program on GPU using OpenCL: vector addition

CPU version

```
void vectorAdd(float *a, float *b, float *c, int N)
{
    for ( int i = 0; i < N; i++ ) {
        c[i] = a[i] + b[i];
    }
}
```

GPU version: in kernel source file vectorAdd.cl

```
__kernel void vectorAdd(__global float *a,
                        __global float *b,
                        __global float *c)
{
    // i is the thread id, along the 0th direction
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Example code from pyopencl

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from __future__ import absolute_import, print_function
import numpy as np
import pyopencl as cl

# a_np, b_np, res_np are vectors in CPU memory
a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)
res_np = np.empty_like(a_np)

# interactively choose computing devices and opencl platform
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

# open kernel *.cl file and compile
prg = cl.Program(ctx, open("vectorAdd.cl").read()).build()

# copy data to GPU global memory
mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)
res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)

# do vector addition in parallel
prg.vectorAdd(queue, a_np.shape, None, a_g, b_g, res_g)

# copy data from GPU global memory to host memory
cl.enqueue_copy(queue, res_np, res_g)

# Check on CPU with Numpy:
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
```

Hydrodynamic equations can be simplified to a general form

$$\partial_\tau Q + \partial_x F^x + \partial_y F^y + \partial_\eta F^\eta = S \tag{4}$$

where Q is the conservative variable, $F^{x,y,\eta}$ the flux along x, y, η directions and S the source term.

$$Q = \left[\tilde{T}^{\tau\nu}, \tilde{u}^\tau \tilde{\pi}^{\mu\nu}, \tilde{N}^\tau, \tilde{u}^\tau \tilde{\Pi}\right] \tag{5}$$

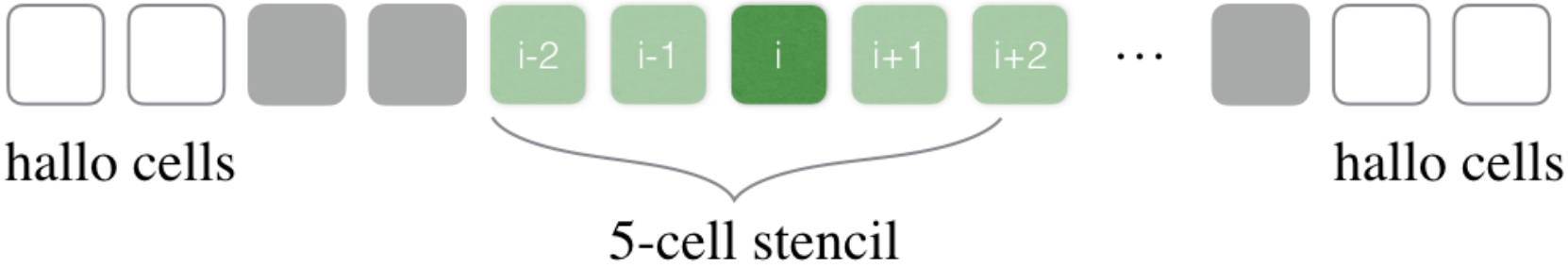
Simple finite difference

$$\partial_x f \approx \frac{f_{i+1} - f_i}{\Delta x} \approx \frac{f_i - f_{i-1}}{\Delta x} \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x} \tag{6}$$

Kurganov and Tadmor(KT) algorithm (finite volume),

$$\begin{aligned} Q^{n+1}_{i,j,k} \quad = \quad & KT(F^{x,n}_{i-2,j,k}, F^{x,n}_{i-1,j,k}, F^{x,n}_{i,j,k}, F^{x,n}_{i+1,j,k}, F^{x,n}_{i+2,j,k}, && (7)\\ & F^{y,n}_{i,j-2,k}, F^{y,n}_{i,j-1,k}, F^{y,n}_{i,j,k}, F^{y,n}_{i,j+1,k}, F^{y,n}_{i,j+2,k}, && (8)\\ & F^{\eta,n}_{i,j,k-2}, F^{\eta,n}_{i,j,k-1}, F^{\eta,n}_{i,j,k}, F^{\eta,n}_{i,j,k+1}, F^{\eta,n}_{i,j,k+2}, && (9)\\ & S^n_{i,j,k}). && (10) \end{aligned}$$

Stripe-by-stripe updating using shared memory in CLVisc



Global memory accessing is reduced from 13 times to 4 times.

```
    # CLVisc = python (host side) + opencl (device side)
    def IS_stepUpdate(self, step):
        NX, NY, NZ, BSZ = self.cfg.NX, self.cfg.NY, self.cfg.NZ, self.cfg.BSZ

        self.kernel_IS.visc_src_christoffel(self.queue, (NX*NY*NZ,), None,
                self.d_IS_src, self.d_pi[step], self.ideal.d_ev[step],
                self.ideal.tau, np.int32(step)).wait()
        # update along x direction
        self.kernel_IS.visc_src_alongx(self.queue, (BSZ, NY, NZ), (BSZ, 1, 1),
                self.d_IS_src, self.d_udx, self.d_pi[step], self.ideal.d_ev[step],
                self.eos_table, self.ideal.tau).wait()
        # update along y direction
        self.kernel_IS.visc_src_alongy(self.queue, (NX, BSZ, NZ), (1, BSZ, 1),
                self.d_IS_src, self.d_udy, self.d_pi[step], self.ideal.d_ev[step],
                self.eos_table, self.ideal.tau).wait()
        # update along space-time rapidity direction
        self.kernel_IS.visc_src_alongz(self.queue, (NX, NY, BSZ), (1, 1, BSZ),
                self.d_IS_src, self.d_udz, self.d_pi[step], self.ideal.d_ev[step],
                self.eos_table, self.ideal.tau).wait()
        # afterwards pi^{mu nu} terms are updated with d_IS_src
```

Profiling

block size	8	16	32	64	128
Ideal(s)-GPU	0.37	0.218	0.178	0.155	0.157
Visc(s)-GPU	3.12	1.65	1.17	1.01	1.17
Visc(s)-CPU	6.64	6.45	6.63	7.0	7.58

Table: Computing time for one time step on various computing devices for several different block sizes, (3+1D) viscous hydro $N_{\rm cell} = 385 \times 385 \times 115$ for 1600 time steps.

- CPU: Intel Xeon 2650v2 (10 cores, 20 threads)
- GPU: AMD S9150 (2496 processing elements)
- Block size: the number of processing elements assigned to process one workgroup of cells

Coalesced reading makes z-update 6 times faster

```
Line #  Hits         Time  Per Hit   %Time  Line Contents
==============================================================
165                                         @profile
166                                         def IS_stepUpdate(self, step):
                                                # NX = NY = NZ = 256
168     52          152       2.9       0.0       NX, NY, NZ, BSZ = self.cfg.NX, self.cfg.NY, self.cfg.NZ, self
      .cfg.BSZ
169
170     52       143954    2768.3       0.7       self.kernel_IS.visc_src_christoffel(self.queue, (NX*NY*NZ,),
      None,
171     52          134       2.6       0.0                 self.d_IS_src, self.d_pi[step], self.ideal.d_ev[step
      ],
172     52       581359   11180.0       3.0                 self.ideal.tau, np.int32(step)).wait()
173
174     52       159298    3063.4       0.8       self.kernel_IS.visc_src_alongx(self.queue, (BSZ, NY, NZ), (
      BSZ, 1, 1),
175     52          143       2.8       0.0                 self.d_IS_src, self.d_udx, self.d_pi[step], self.
      ideal.d_ev[step],
176     52      8055724  154917.8      41.9                 self.eos_table, self.ideal.tau).wait()
179
180     51       156991    3078.3       0.8       self.kernel_IS.visc_src_alongy(self.queue, (NX, BSZ, NZ), (1,
       BSZ, 1),
181     51          151       3.0       0.0                 self.d_IS_src, self.d_udy, self.d_pi[step], self.
      ideal.d_ev[step],
182     51      7381515  144735.6      38.4                 self.eos_table, self.ideal.tau).wait()
183
184                                             # data is continues along z(etas) direction
185     51       157382    3085.9       0.8       self.kernel_IS.visc_src_alongz(self.queue, (NX, NY, BSZ), (1,
       1, BSZ),
186     51          137       2.7       0.0                 self.d_IS_src, self.d_udz, self.d_pi[step], self.
      ideal.d_ev[step],
187     51      1329880   26076.1       6.9                 self.eos_table, self.ideal.tau).wait()
```

Spectra calculation on GPU (example from ideal hydro)

Perfect job for GPU

$$\frac{dN}{dY p_T dp_T d\phi} = \frac{g_s}{(2\pi)^3} \int_\Sigma p^\mu d\Sigma_\mu \frac{1}{\exp((p\cdot u - \mu)/T_{FO}) \pm 1} \tag{11}$$

- Up to $200,000$ small pieces of $d\Sigma_\mu$.
- Usually need 41 rapidity(Y) bins, 15 transverse momentum(p_T) bins, 48 azimuthal angle(ϕ) bins.
- More than 300 resonance particles.
- For each event, needs to calc. exp function $200,000 * 41 * 15 * 48 * 300$ times.

Spectra calculation on GPU (example from ideal hydro)

Perfect job for GPU

$$\frac{dN}{dY p_T dp_T d\phi} = \frac{g_s}{(2\pi)^3}\int_\Sigma p^\mu d\Sigma_\mu \frac{1}{\exp((p\cdot u-\mu)/T_{FO})\pm 1} \tag{11}$$

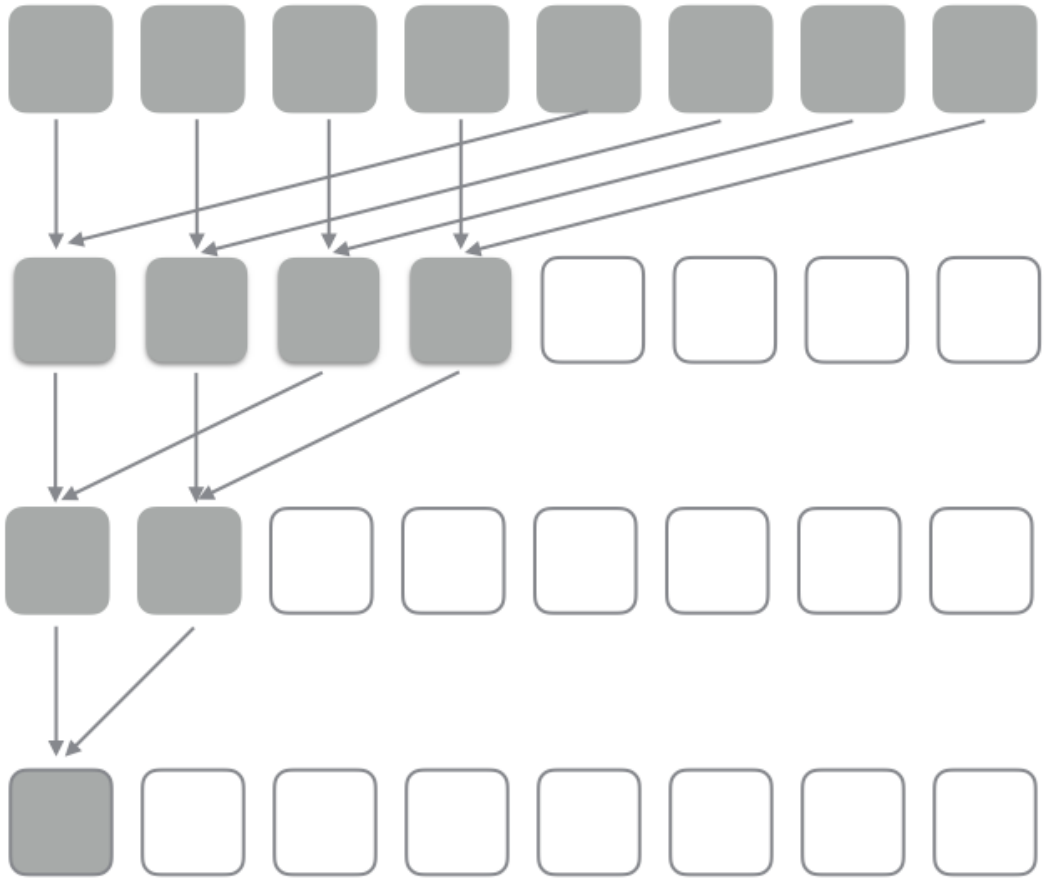
- Up to $200,000$ small pieces of $d\Sigma_\mu$.
- Usually need 41 rapidity(Y) bins, 15 transverse momentum(p_T) bins, 48 azimuthal angle(ϕ) bins.
- More than 300 resonance particles.
- For each event, needs to calc. exp function $200,000 * 41 * 15 * 48 * 300$ times.

Pb+Pb 2.76TeV/n, 20-25%

	CPU (i5-430M)	GPU (GT-240M)	GPU (K20)
Smooth spec. for π^+	7 minutes	30 seconds	0.5 seconds

Table: GPU(48 cuda cores) is 10-30 times faster than CPU. NVIDIA K20 GPU has 2496 cuda cores.

Widely used parallel reduction (spectra and max energy density calc.)



Parallel reduction kernel using OpenCL

```
int tid = get_local_id(0);
int localSize = get_local_size(0);
// spec in local/shared memory
__local real spec[NBlocks];
// read data to local/shared memory
spec[tid] = d_SubSpec[I];
// memory fence to make sure all data read in
barrier(CLK_LOCAL_MEM_FENCE);
//unroll loop: s = 32, 16, 8, 4, 2 if localSize=64
for (int s = localSize>>1; s>0; s >>= 1){
     if (tid < s) spec[tid] += spec[s + tid];
     barrier(CLK_LOCAL_MEM_FENCE);
}
int idx = pid*NY*NPT*NPHI + get_group_id(0);
if (tid == 0) d_Spec[idx] = spec[tid];
```

Computational graph for GPU parallelization (deep learning libraries)

```
# Creates a graph.
with tf.device('/device:GPU:1'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2,
        3], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3,
        2], name='b')
    c = tf.matmul(a, b)
# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=
    True))
# Runs the op.
print(sess.run(c))
```

- Modern deep learning libraries hide the GPU parallelization in the backend.
- E.g., tensorflow writes kernel programs from host side using computational graph.
- This is a new option to parallelize scientific programs on GPUs.

Conclusion

- Event-by-event hydro simulations demands high performance computing
- Correctly using memory is very important for GPU parallel programing
- Many processes in heavy ion collisions can be parallelized on GPUs: initial condition manipulation, hydro evolution, smooth spectra calculation
- More details (comparisons with analytical solutions and physical observables) are coming soon in the long CLVisc paper.

Backups

C++ version host side helper functions (used in smooth spectra calculation).

Data buffer for vector addition in OpenCL

Data in host memory

```
// c[i] = a[i] + b[i]
const int N;    //The length of the array
std::vector<cl_float> h_a(N);
std::vector<cl_float> h_b(N);
std::vector<cl_float> h_c(N);
```

Data buffer for vector addition in OpenCL

Data in host memory

```
// c[i] = a[i] + b[i]
const int N;    //The length of the array
std::vector<cl_float> h_a(N);
std::vector<cl_float> h_b(N);
std::vector<cl_float> h_c(N);
```

Data on device (GPU)

```
cl::Buffer d_a = cl::Buffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  N*sizeof(cl_float), h_a.data());

cl::Buffer d_b = cl::Buffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  N*sizeof(cl_float), h_b.data());

cl::Buffer d_c = cl::Buffer(context, CL_MEM_READ_WRITE, N*
    sizeof(cl_float));
```

Set up environment (Copy and Paste)

```
cl_int device_type = CL_DEVICE_TYPE_GPU;

/** associate devices, programs, kernels */
cl::Context context = CreateContext(device_type);

/** CL_DEVICE_TYPE_CPU/GPU/ALL */
std::vector<cl::Device> devices = context.getInfo<
    CL_CONTEXT_DEVICES>();

/** Bunch of kernel functions in *.cl kernel files*/
std::vector<cl::Program> programs;

programs.push_back(CreateProgram(''vectorAdd.cl''));

/** Read/Write Buffer and Run kernel on device*/
cl::CommandQueue queue = cl::CommandQueue(context, devices[0]);

/** kernel that run on GPU */
cl::Kernel kernel = cl::Kernel(programs.at(0), ''vectorAdd'');
/** Notice: CreateContext() and CreateProgram() are helper
    functions */
/** that can be found in backup slides*/
```

Run kernel on GPU (Copy and Paste)

```
/** Set kernel arguments */
kernel.setArg(0, d_a);
kernel.setArg(1, d_b);
kernel.setArg(2, d_c);
kernel.setArg(3, N);

/** Assign N jobs to workitems */
cl::NDRange globalSize = cl::NDRange( N );
cl::NDRange localSize = cl::NullRange;

cl::Event event;
/** Run kernel function on device */
queue.enqueueNDRangeKernel( kernel,
cl::NullRange,    // offset
globalSize,       // how many jobs
localSize,        // how does the job distribute
NULL,             //
&event);          // event for sync, profiling
/** Block until kernel completion, async command queue only*/
event.wait();
/** Copy data back to host memory h_c */
queue.enqueueReadBuffer(d_c, CL_TRUE, 0, N*sizeof(cl_float),
    h_c.data());
```

Create context and get devices

```
cl::Context CreateContext( const cl_int & device_type )
{
    std::vector<cl::Platform> platforms;
    cl::Platform::get( &platforms );
    if( platforms.size() == 0 ){
        std::cerr<<"no platform found\n";
        exit(EXIT_FAILURE);
    } else {
        for( int i=0; i < platforms.size(); i++ ){
            std::vector<cl::Device> supportDevices;
            platforms.at(i).getDevices( CL_DEVICE_TYPE_ALL, & supportDevices );
            for(int j=0; j < supportDevices.size(); j++ ){
                if( supportDevices.at(j).getInfo<CL_DEVICE_TYPE>() == device_type ) {
                    std::cout<<"Found device "<<device_type<<" on platform "<<i<<std::endl;
                    cl_context_properties properties[] =
                    { CL_CONTEXT_PLATFORM,
                        (cl_context_properties) (platforms.at(i))(),
                        0 };
                    return cl::Context( device_type, properties );
                }// Found supported device and platform
            }// End for devices
        }// End for platform
        //// if no platform support device_type, exit
        std::cerr<<"no platform support device type"<<device_type<<std::endl;
        exit(EXIT_FAILURE);
    }
}
```

Read kernel files *.cl

```
cl::Program CreateProgram(const char * fname)
{ /** Helper function to load *.cl program*/
    std::ifstream kernelFile(fname);
    if( !kernelFile.is_open() ) std::cerr<<"Open␣"<<fname << "␣failed!"<<std::endl;
    std::string sprog(std::istreambuf_iterator<char> (kernelFile),
                      (std::istreambuf_iterator<char> ()));

    cl::Program::Sources prog(1, std::make_pair(sprog.c_str(), sprog.length()));

    return cl::Program(context, prog);
}
```

Compile programs on device

```
void BuildPrograms(std::vector<cl::Program> programs, const char * compile_options)
{ //// build programs and output the compile error if there is
    for ( std::vector<cl::Program>::size_type i = 0; i != programs.size(); i++ ) {
        try {
            programs.at(i).build(devices, compile_options);
        } catch(cl::Error & err) {
            std::cerr<<err.what()<<"("<<err.err()<<")\n"<<programs.at(i).getBuildInfo<CL_PROGRAM_BUILD_LOG
                  >(devices[0]);
        }
    }
}
```

Important comments

Online resources and books for OpenCL

- Khronos.org website
- OpenCL specification and headers
- OpenCL c++ bindings
- BOOK: OpenCL In Action
- BOOK: OPENCL PROGRAMMING GUIDE
- BOOK: Heterogeneous Computing With OpenCL