GPU algorithms for speeding up a 2D Fluid Solver

John Callow

California State University Long Beach

john.j.callow@gmail.com

December 14, 2015

Navier-Stokes (no viscosity or external forces for now)

$$\frac{d\vec{u}}{dt} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = 0$$

Such that

$$\nabla \cdot \vec{u} = 0$$

Instead of directly solving, split into simpler problems

Advection

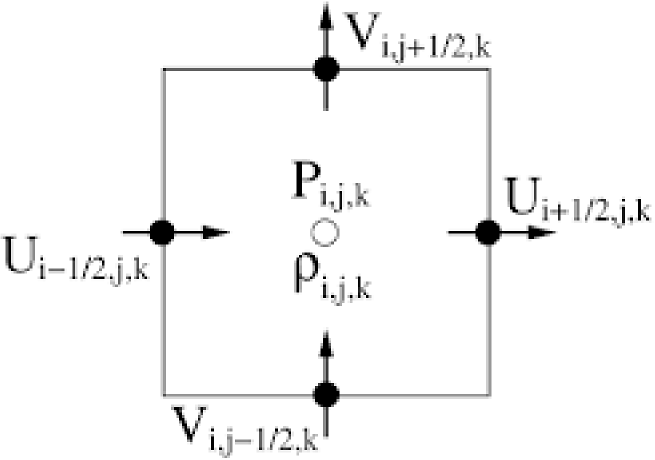
$$\frac{d\vec{u}}{dt} + \vec{u} \cdot \nabla \vec{u} = 0$$

Pressure/Incompressibility

$$\frac{d\vec{u}}{dt} + \frac{1}{\rho} \nabla p = 0$$

Such that

$$\nabla \cdot \vec{u} = 0$$


$V_{i,j+1/2,k}$
$P_{i,j,k}$
$U_{i+1/2,j,k}$
$U_{i-1/2,j,k}$
$\rho_{i,j,k}$
$V_{i,j-1/2,k}$

Finite difference methods unstable. Instead use semi-Lagrangian (Start at grid point, look back in time to see what ends up at this point).

```
__global__
void advect(GridData* data, float timestep, float delx, GridData *u, GridData *v) {
    int index = threadIdx.x + blockIdx.x*blockDim.x;

    int x = index % data->width;
    int y = index/data->width;

    if (x < data->width && y < data->height) {

        float ox = x + data->offset_x;
        float oy = y + data->offset_y;

        rungeKutta3(data, ox, oy, timestep, delx, u, v);

        data->dst[index] = cubic_interpolation(data, ox, oy);
    }
}
```

Pressure Correction to enforce $\nabla \cdot \vec{u} = 0$

- calculate negative divergence $\vec{b} = -\nabla \cdot \vec{u}_{adv}$
- Set entries of finite difference pressure coefficient matrix A
- Construct MIC(0) - Modified Incomplete Cholesky preconditioner
- Solve $Ap = b$ with MICCG(0) - Conjugate Gradient with MIC(0)
- set $\vec{u}^{n+1} = \vec{u}_{adv} - \Delta t \frac{1}{p} \nabla p$

```
__global__
void buildRHS(GridData *u, GridData *v, float *r, float delx, int width, int height) {
    float scale = 1.0/delx;
    int id = threadIdx.x + blockIdx.x*blockDim.x;

    int x = id % width;
    int y = id/width;

    if (x < width && y < height) {
        r[id] = -scale*(u->get(x+1,y) - u->get(x,y) + v->get(x, y+1) - v->get(x,y));
    }

}
```

Finite Difference applied to pressure update equations

$$u_{i+1/2,j}^{n+1} = u_{i+1/2,j}^{adv} - \Delta t \frac{1}{p} \frac{p_{i+1,j} - p_{i,j}}{\Delta x}$$

$$v_{i,j+1/2}^{n+1} = v_{i,j+1/2}^{adv} - \Delta t \frac{1}{p} \frac{p_{i,j+1} - p_{i,j}}{\Delta x}$$

Finite Differences to calculate divergence

$$\nabla \cdot \vec{u}_{i,j} = \frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta x}$$

Combining the above, taking the divergence, setting to 0, and solving for p

$$\frac{\Delta t}{\rho}\left(\frac{4p_{i,j} - p_{i+1,j} - p_{i,j+1} - p_{i-1,j} - p_{i,j-1}}{\Delta x}\right) = -\left(\frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta x}\right)$$

This can be represented as the problem $Ap = b$ with A being a matrix of the above pressure coefficients, p the pressures as a vector, and b the vector of divergence values

Algorithm Preconditioned Conjugate Gradient

p = 0
r = b
if infNorm(r) = 0 **then**
  return p
end if
z = applyPreconditioner(r)
s = z
while *iterations* < *max* **do**
  $z = MatrixVectorProduct(A, s)$
  $\alpha = \sigma / dotProduct(z, s)$
  $p = p + \alpha s$
  $r = r - \alpha s$
  if infNorm(r) ¡ tol **then**
    return p
  end if
  $z = applyPreconditioner(r)$
  $\sigma_{new} = dotProduct(z, r)$
  $\beta = \sigma_{new} / \sigma$
  $s = z + \beta s$
  $\sigma = \sigma_{new}$
end while
return p

```
__global__
void dotProduct_Device(float *a, float *b, float *result, int length, int nearest2pow) {
    extern __shared__ float tmp[];
    int id = threadIdx.x + blockIdx.x*blockDim.x;
    int locId = threadIdx.x;

    if (id < length) {
        tmp[locId] = a[id]*b[id];
    } else {
        tmp[locId] = 0;
    }
    __syncthreads();

    for (int i = nearest2pow/2; i > 0; i /= 2) {
        if (locId < i && locId + i < blockDim.x)
            tmp[locId] += tmp[locId + i];
        __syncthreads();
    }

    if (locId == 0)
        result[blockIdx.x] = tmp[0];
}
```

```
__host__
float dotProduct(float *a, float *b, float* dev_partial, int ↩
    length, int blocks, int threads, int nearest2pow) {

  dotProduct_Device<<<blocks, threads, blocks*sizeof(float)>>>(a↩
      , b, dev_partial, length, nearest2pow);

  float partial[blocks];

  CUDA_CHECK_RETURN(cudaMemcpy(partial, dev_partial, blocks*↩
      sizeof(float), cudaMemcpyDeviceToHost));

  float dotproduct = 0;
  for (int i = 0; i < blocks; i++) {
    dotproduct += partial[i];
  }

  return dotproduct;
}
```

```
__global__
void infinityNorm_Device(float *a, float *result, int length, int nearest2pow) {
    extern __shared__ float tmp[];

    int id = threadIdx.x + blockIdx.x*blockDim.x;
    int locId = threadIdx.x;

    if (id < length)
        tmp[locId] = abs(a[id]);
    else
        tmp[locId] = 0;

    for (int i = nearest2pow/2; i > 0; i /= 2) {
        if (locId < i && locId + i < blockDim.x) {
            tmp[locId] = max(tmp[locId + i], tmp[locId]);
        }
        __syncthreads();
    }

    if (locId == 0) {
        result[blockIdx.x] = tmp[0];
}
```

Infinity Norm Host

```
__host__
float infinityNorm(float* a, float *dev_partial, int length, int blocks, int threads, int nearest2pow) {
    infinityNorm_Device<<<blocks, threads, blocks*sizeof(float)>>>(a, dev_partial, length, nearest2pow);

    float partial[blocks];

    CUDA_CHECK_RETURN(cudaMemcpy(partial, dev_partial, blocks*sizeof(float), cudaMemcpyDeviceToHost));

    float infinity_norm = 0;
    for (int i = 0; i < blocks; i++) {
        infinity_norm = max(infinity_norm, partial[i]);
    }

    return infinity_norm;
}
```

```
__global__
void matrixVectorProduct(float *aDiag, float *aPlusX, float *aPlusY, float *dst, float *b, int width, int height) {
    int id = threadIdx.x + blockIdx.x*blockDim.x;
    int x = id % width;
    int y = id/width;

    if (x < width && y < height) {
        float t = aDiag[id]*b[id];

        if (x > 0)
            t += aPlusX[id-1]*b[id-1];
        if (y > 0)
            t += aPlusY[id - width]*b[id - width];
        if (x < width - 1)
            t += aPlusX[id]*b[id + 1];
        if (y < height - 1)
            t += aPlusY[id]*b[id + width];
        dst[id] = t;
    }
}
```

Algorithm Preconditioner Pseudocode

$\tau = 0.97$
$\phi = 0.25$
for $i = 1$ to *width* **do**
  for $j = 1$ to *height* **do**
    $px1 = Aplusi[i-1,j] * precon[i-1,j]$
    $px2 = Aplusi[i,j-1] * precon[i,j-1]$
    $py1 = Aplusj[i-1,j] * precon[i-1,j]$
    $py2 = Aplusj[i,j-1] * precon[i,j-1]$
    $e = Adiag[i,j] - px1^2 - py1^2 - \tau(px1py1 + px2py2)$
    if $e < \sigma * Adiag[i,j]$ **then**
      $e = Adiag[i,j]$
    end if
    $precon[i,j] = \frac{1}{\sqrt{e}}$
  end for
end for

```
__global__
void buildPreconditioner(float *aDiag, float *aPlusX, float *aPlusY, float *preconditioner, int width, int height) {
   float tau = 0.97;
   float sigma = 0.25;
   int id = threadIdx.x;
   int x, y, matrixId;
   int limit = max(width, height);
   for (int i = 0; i < 2*limit; i++) {
      x = i - id;
      y = id;
      matrixId = x + y*width;
      if (x >= 0 && x < width && y >= 0 && y < height) {
         float e = aDiag[matrixId];
         if (x > 0) {
            float px = aPlusX[matrixId - 1]*preconditioner[matrixId-1];
            float py = aPlusY[matrixId-1]*preconditioner[matrixId-1];
            e = e - (px*px + tau*px*py);
         }
```

```
            if (y > 0) {
                float px = aPlusX[matrixId - width]*preconditioner[matrixId-width];
                float py = aPlusY[matrixId-width]*preconditioner[matrixId-width];
                e = e - (py*py + tau*px*py);
            }
            if (e < sigma*aDiag[matrixId]) {
                e = aDiag[matrixId];
            }
            preconditioner[matrixId] = 1.0/sqrt(e);
        }
        __syncthreads();
    }
}
```

Algorithm Solves $Lq = r$ then $L^T z = q$ to get $L(L^T z) = r$

for $i = 1$ to *width* **do**
  for $j = 1$ to *height* **do**
    $t = r[i,j] - Aplusi[i-1,j] * precon[i-1,j] * q[i-1,j] -$
    $Aplusj[i,j-1] * precon[i,j-1] * q[i,j-1]$
    $q[i,j] = t * precon[i,j]$
  end for
end for
for $i = width$ down to 1 **do**
  for $j = height$ down to 1 **do**
    $t = q[i,j] - Aplusi[i,j] * precon[i,j] * z[i+1,j] -$
    $Aplusj[i,j] * precon[i,j] * z[i,j+1]$
    $z[i,j] = t * precon[i,j]$
  end for
end for

```
__global__
void applyPreconditioner(float *aDiag, float *aPlusX, float *aPlusY, float *preconditioner, float *dst, float *a, int width, int height) {
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  int x, y, matrixId;
  int limit = max(width, height);

  for (int i = 0; i < 2*limit; i++) {
    x = i - id;
    y = id;
    matrixId = x + y*width;
    if (x >= 0 && x < width && y >= 0 && y < height) {
      float t = a[matrixId];
      if (x > 0)
        t -= aPlusX[matrixId -1]*preconditioner[matrixId-1]*dst[matrixId-1];

      if (y > 0)
        t -= aPlusY[matrixId - width]*preconditioner[matrixId-width]*dst[matrixId-width];
      dst[matrixId] = t*preconditioner[matrixId];
    }
```

GPU code MIC Preconditioner

```
        __syncthreads();
    }
    for (int i = 2*limit-1; i >= 0; i--) {
        x = i - limit + id;
        y = limit-1-id;
        matrixId = x + y*width;
        if (x >= 0 && x < width && y >= 0 && y < height) {
            float t = dst[matrixId];
            if (x < width-1)
                t -= aPlusX[matrixId]*preconditioner[matrixId]*dst[matrixId+1];
            if (y < height-1)
                t -= aPlusY[matrixId]*preconditioner[matrixId]*dst[matrixId+width];
            dst[matrixId] = t*preconditioner[matrixId];
        }
        __syncthreads();
    }
}
```

Currently, the gpu apply preconditioner is bad, so I've use host apply for both tests tests. The cpu code tested against is by tunabrain (see sources). His/her code is also based on Robert Bridson's book.

512x512
CPU - 3.1 seconds per frame
GPU - 2.2 seconds per frame

1024x1024
CPU 35 seconds per frame
Gpu 15 seconds per frame

- Bridson, Robert. Fluid Simulation for Computer Graphics. Wellesley, MA: A K Peters Ltd. 2008. Print.
- Tunabrain, https://github.com/tunabrain/incremental-fluids