Optimizing DPD Pairwise Kernels for Petascale Microfluidic Simulations

Yu-Hang Tang, Diego Rossinelli, Mauro Bison, George Karniadakis

San Jose | GTC 2016 | April 6, 2016
uDeviceX
http://udevicex.github.io/

Sub-cellular-resolution simulations of cell suspensions in microfluidic devices using Dissipative Particle Dynamics.

94 mm³ microfluidic space

1.4×10⁹ blood cells

40% peak of TITAN

The In-Silico Lab-on-a-Chip: Petascale and High-throughput Simulations of Microfluidics at Cell Resolution, SC 15
Detection of rare circulating tumor cells *in silico*

Video Credit: Christian Conti @ ETH Zurich
Dissipative Particle Dynamics

a stochastic, coarse-grained particle method for simulating mesoscopic fluids.

DPD Pairwise Force

Conservative
\[ F_{ij}^C = a_{ij} \left[ 1 - \frac{r}{r_0} \right] \]

Dissipative
\[ F_{ij}^D = -\gamma_{ij} w_D(r_{ij})(e_{ij} \cdot v_{ij})e_{ij} \]

Random
\[ F_{ij}^R = \sigma_{ij} w_R(r_{ij})\xi_{ij}\delta t^{-\frac{1}{2}}e_{ij} \]

60% of total simulation time
Cell list-based algorithm

Particles ... 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 ...

Cells ...

... 24 27 31 35 38 42 ...

Each warp cooperatively computes the interactions between a cell and its neighbors.
Implementation 0 - Baseline

shared start[32], count[32]
lane = threadIdx.x mod 32

for cell i = id_{warp} to N_{cells} stride N_{warps} // inter-warp loop
    if lane < 27:
        ncid = id of lane^{th} neighbor cell
        count[lane] = cell_size[ncid]
        start[lane+1] = prefix( count[lane] )

    for j = 0 to count[27] stride 32 // intra-warp loop
        k = trisection( count, j )
        s = start[k] + j - count[k] // id of j^{th} neighbor

    for each particle d in cell i
        if |x_d-x_s| < r_c
            f = force( s, d )
            f_{total} = shuffle( f )
            if lane == 0: force[d] += f_{total}

64^3 Cells, 1M Particles

<table>
<thead>
<tr>
<th>GPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20</td>
<td>12.82 ms</td>
</tr>
<tr>
<td>TITAN X</td>
<td>5.80 ms</td>
</tr>
</tbody>
</table>
Optimization 1 - Enforcing Newton’s 3\textsuperscript{rd} Law

Calculate each local pair of force only once

Forces between MPI ranks (bipartite) still calculated twice

atomicAdd appears fast enough

Result not really needed $\rightarrow$ atomic reduction (SASS: RED) does not block thread execution unless throttled
Implementation 1

shared start[32], count[32]
lane = threadIdx.x mod 32
for cell i = id\_warp to N\_cells stride N\_warps
    if lane < 27:
        ncid = id of lane\(^{th}\) neighbor cell
        count[lane] = cell\_size[ncid]
        start[lane+1] = prefix( count[lane] )
    for j = 0 to count[27] stride 32
        k = trisection( count, j )
        s = start[k] + j - count[k]
    for each particle d in cell i
        if \(|x_d - x_s| < r_c\)
            f = force( s, d )
            f\_total = shuffle( f )
        if lane == 0: force[d] += f\_total

shared start[16], count[16]
lane = threadIdx.x mod 32
for cell i = id\_warp to N\_cells stride N\_warps
    if lane < 14:
        ncid = id of lane\(^{th}\) neighbor cell
        count[lane] = cell\_size[ncid]
        start[lane+1] = prefix( count[lane] )
    for j = 0 to count[14] stride 32
        k = trisection( count, j )
        s = start[k] + j - count[k]
    for each particle d in cell i
        if \(|x_d - x_s| < r_c\)
            f = force( s, d )
            f\_total = shuffle( f )
        if lane == 0: force[d] += f\_total
        if s < d:
            atomic( force[s]-=f )
## Implementation 1

<table>
<thead>
<tr>
<th>Baseline vs. OPT 1</th>
<th>Inst. # (M)</th>
<th>Baseline</th>
<th>Atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20 12.82 ms → 9.40 ms ↓27%</td>
<td>FP32</td>
<td>1171</td>
<td>658</td>
</tr>
<tr>
<td>TITAN X 5.80 ms → 4.26 ms ↓27%</td>
<td>INT</td>
<td>3340</td>
<td>1872</td>
</tr>
<tr>
<td></td>
<td>LD/ST</td>
<td>602</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>COMM</td>
<td>42</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>ATOMIC</td>
<td>0</td>
<td>46</td>
</tr>
</tbody>
</table>
Optimization 2

An in-situ RNG is needed to compute the pairwise random force
Popular choices for DPD: SARU, TEA, etc.

Tiny Encryption Algorithm

```c
uint32_t v0=v[0], v1=v[1], sum=0, i;
uint32_t delta=0x9e3779b9;
uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
for (i=0; i < NROUNDS; i++) {
    sum += delta;
    v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
    v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
}
v[0]=v0; v[1]=v1;
```

SARU

```c
seed3 ^= ( seed1 << 7 ) ^ ( seed2 >> 6 );
seed2 += ( seed1 >> 4 ) ^ ( seed3 >> 15 );
seed1 ^= ( seed2 << 9 ) + ( seed3 << 8 );
seed3 ^= 0xA5366B4D * ( ( seed2 >> 11 ) ^ ( seed1 << 1 ) );
seed2 += 0x72BE1579 * ( ( seed1 << 4 ) ^ ( seed3 >> 16 ) );
seed1 ^= 0X3F38A6ED * ( ( seed3 >> 5 ) ^ ( ( ( signed int )seed2 ) >> 22 ) );
seed2 += seed1 * seed3;
seed1 += seed3 ^ ( seed2 >> 2 );
seed2 ^= ( ( signed int )seed2 ) >> 17;
int state = 0x79dedea3 * ( seed1 ^ ( ( ( signed int )seed1 ) >> 14 ) );
int wstate = ( state + seed2 ) ^ ( ( ( signed int )state ) >> 8 );
state = state + ( wstate * ( wstate ^ 0xdddf97f5 ) );
wstate = 0xABCB96F7 + ( wstate >> 1 );
state = 0x4beb5d59 * state + 0x2600e1f7;
wstate = wstate + 0x8009d14b + ( ( ( ( signed int )wstate ) >> 31 ) & 0xda879add );
unsigned int v = ( state ^ ( state >> 26 ) ) + wstate;
unsigned int r = ( v ^ ( v >> 20 ) ) * 0x6957f5a7;
```

Yu-Hang Tang yuhang_tang@brown.edu Karniadakis Group Brown University | GPU Tech Conf 2016 S6140 April 6, 2016
Optimization 2

Both RNGs fires a lot of integer ops.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Kepler</th>
<th>Maxwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/MUL/FMA</td>
<td># 192</td>
<td>% 100%</td>
</tr>
<tr>
<td>ADD</td>
<td>160</td>
<td>83%</td>
</tr>
<tr>
<td>MUL</td>
<td>32</td>
<td>17%</td>
</tr>
<tr>
<td>SHIFT</td>
<td>64</td>
<td>33%</td>
</tr>
<tr>
<td>BITWISE</td>
<td>160</td>
<td>83%</td>
</tr>
<tr>
<td>CLZ/FFS</td>
<td>32</td>
<td>17%</td>
</tr>
<tr>
<td>POPC</td>
<td>32</td>
<td>17%</td>
</tr>
<tr>
<td>MUL</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SHUFFLE</td>
<td>32</td>
<td>17%</td>
</tr>
<tr>
<td>CMP</td>
<td>160</td>
<td>83%</td>
</tr>
</tbody>
</table>

```
seed3 ^= ( seed1 << 7 ) ^ ( seed2 >> 6 );
seed2 += ( seed1 >> 4 ) ^ ( seed3 >> 15 );
seed1 ^= ( seed2 << 9 ) + ( seed3 << 8 );
seed3 ^= 0xA5366B4D * ( ( seed2 >> 11 ) ^ ( seed1 << 1 ) );
seed2 += 0x72BE1579 * ( ( seed1 << 4 ) ^ ( seed3 >> 16 ) );
seed1 ^= 0x3F38A6ED * ( ( seed3 >> 5 ) ^ ( ( signed int )seed2 ) >> 22 );
seed2 += seed1 * seed3;
seed1 += seed3 ^ ( seed2 >> 2 );
seed2 ^= ( ( signed int )seed2 ) >> 17;
int state = 0x79dedea3 * ( seed1 ^ ( ( signed int )seed1 ) >> 14 );
int wstate = ( state + seed2 ) ^ ( ( signed int )state ) >> 8 );
state = state + ( wstate ^ ( wstate ^ 0xdddf97f5 ) );
wstate = 0xABCB96F7 + ( wstate >> 1 );
state = 0x4beb5d59 * state + 0x2600e1f7;
wstate = wstate + 0x8009d14b + ( ( ( signed int )wstate ) >> 31 ) & 0xda879add ;
unsigned int v = ( state ^ ( state >> 26 ) ) + wstate;
unsigned int r = ( v ^ ( v >> 20 ) ) * 0x6957f5a7;
```
Optimization 2 – Floating Point-based RNG

The Logistic Map $X_{n+1} = 4X_n(1-X_n)$, $X \in (0,1)$

Very simple (easy to implement), yet chaotic (unpredictable)
Optimization 2 - FPRNG

Scaling the recurrence relation onto \([-1,1]\) simplifies the iterated formula. This can be explained by recognizing that the iterated logistic map is essentially the Chebyshev Polynomial of the First Kind, while the scaling eliminates the odd-ordered components from the formula.

Let
\[ Y_n = 1 - 2X_n, Y_n \in (-1,1) \]

Then
\[
\begin{align*}
Y_{n+1} &= 1 - 2X_{n+1} \\
&= 8X_n^2 - 8X_n + 1 \\
&= 2Y_n^2 - 1
\end{align*}
\]

\[
\begin{align*}
X_{n+1} &= -4(X_n^2 - X_n) \\
X_{n+2} &= -64X_n^4 + 128X_n^3 - 80X_n^2 + 16X_n \\
X_{n+3} &= -16384X_n^8 + 65536X_n^7 - \cdots - 1344X_n^2 + 64X_n
\end{align*}
\]

\[
\begin{align*}
Y_{n+1} &= 2Y_n^2 - 1 \\
Y_{n+2} &= 8Y_n^4 - 8Y_n^2 + 1 \\
Y_{n+3} &= 128Y_n^8 - 256Y_n^6 + 160Y_n^4 - 32Y_n^2 + 1
\end{align*}
\]
Optimization 2 - FPRNG

Pairwise random force requires stateless in-situ RNG

// Seeding with particle local id + per-rank root #
// g, s are irrational numbers e.g. 0.618..., 0.414...

float logi(int i, int j, float root) {
    float x = (root + i * g + j * s) % 1
    return logistic(x, N);
}

Randomness verified by computing mutual information (MI)

\[ MI(X, Y) = \int_x \int_y p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \, dx \, dy \]

18 Logistic passes need \(\rightarrow\) 18 FMA + 9 MUL
## Implementation 2

<table>
<thead>
<tr>
<th>OPT 1 vs. OPT 1:2</th>
<th>Inst. # (M)</th>
<th>Atomic</th>
<th>Atomic/Logistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20 9.40 ms → 8.99 ms</td>
<td>FP32</td>
<td>658</td>
<td>1003</td>
</tr>
<tr>
<td>TITAN X 4.26 ms → 4.10 ms</td>
<td>INT</td>
<td>1872</td>
<td>1234</td>
</tr>
<tr>
<td></td>
<td>LD/ST</td>
<td>347</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>COMM</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>ATOMIC</td>
<td>46</td>
<td>46</td>
</tr>
</tbody>
</table>
Optimization 3 – Divergence Suppression

On average
4 particles per cell
16 neighbors per particle
14 neighbor cells
56 interaction candidates

Only 29% percent of pairwise evaluation give non-zero forces.

**Warp Efficiency**

- Non-branch-deactivated: 38%
- Non-predicated: 37%

```
shared start[16], count[16]
lane = threadIdx.x mod 32
for cell i = id\_warp to N\_cells stride N\_warps
  if lane < 14
    ncid = id of lane\textsuperscript{th} neighbor cell
    count[lane] = cell\_size[ncid]
    start[lane+1] = prefix(count[lane])
  for j = 0 to count[14] stride 32
    k = trisection(count, j)
    s = start[k] + j - count[k]
  for each particle d in cell i
    if |x\_d-x\_s| < r\_c
      f = force(s, d)
      atomic(force[d]+=f)
    if s < d:
      atomic(force[s]-=f)
```
Optimization 3

Queue up force evaluations and compute only when all lanes could get work

\[
\begin{array}{cccccccccc}
\text{\textbf{\textit{stencil}}} & 2 & 3 & 4 & 7 & 8 & 9 & 12 & 13 \\
\text{r}<r_c? & \checkmark & \times & \checkmark & \checkmark & \times & \times & \times & \checkmark \\
\text{ballot} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\text{Thread 0 popc = 0} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\text{Thread 2 popc = 1} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\text{Thread 3 popc = 2} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\text{Thread 7 popc = 3} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\text{queue} & 2 & 4 & 7 & 13 & \text{} & \text{} & \text{} & \text{} \\
\end{array}
\]

\[
\begin{align*}
\text{shared} & \quad \text{start[16], count[16], queue[64]} \\
\text{nq} & \quad = 0 \\
\text{lane} & \quad = \text{threadIdx.x \ mod \ 32} \\
\text{for} & \quad \text{cell i = 0 to N\_cells \ stride N\_warps} \\
\text{if} & \quad \text{lane < 14} \\
\quad & \quad \text{...} \\
\text{for} & \quad j = 0 \text{ to count[14] stride 32} \\
\text{k} & \quad = \text{trisection( count, j )} \\
\text{s} & \quad = \text{start[k] + j - count[k]} \\
\text{for each} & \quad \text{particle d in cell i} \\
\text{p} & \quad = \text{bool( |x_d-x_s| < r_c )} \\
\text{hit} & \quad = \text{ballot( p )} \\
\text{if} & \quad \text{p:} \\
\text{queue[nq + popc( hit & lanemask.lt )]} & \quad = (s,d) \\
\text{nq} & \quad += \text{popc( hit )} \\
\text{if} & \quad \text{nq >= 32:} \\
\text{is, id} & \quad = \text{queue[ lane ]} \\
\text{f} & \quad = \text{force( is, id )} \\
\text{atomic( force[id]+=f )} \\
\text{if} & \quad \text{is < id:} \\
\text{atomic( force[is]-=f )} \\
\text{queue[ lane ]} & \quad = \text{queue[ lane + 32 ]} \\
\text{nq} & \quad -= 32
\end{align*}
\]
### Implementation 3 – Shared FIFO Queue

<table>
<thead>
<tr>
<th>OPT 1:2 vs. OPT 1:3</th>
<th>Inst. # (M)</th>
<th>Atomic/Logistic</th>
<th>SM Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>K20</strong> 8.99 ms → 6.31 ms</td>
<td>FP32</td>
<td>1003</td>
<td>1038</td>
</tr>
<tr>
<td><strong>TITAN X</strong> 4.10 ms → 3.78 ms</td>
<td>INT</td>
<td>1234</td>
<td>1737</td>
</tr>
<tr>
<td></td>
<td>LD/ST</td>
<td>347</td>
<td>430</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Atomic</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHMEM atomics version</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>K20</strong> 10.53 ms</td>
<td>ATOMIC 46</td>
<td>26</td>
</tr>
<tr>
<td><strong>TITAN X</strong> 3.83 ms</td>
<td>WARP % 38%</td>
<td>77%</td>
</tr>
<tr>
<td></td>
<td>WARP NP % 37%</td>
<td>75%</td>
</tr>
</tbody>
</table>
Locally Transposed Force Array

AOS (xyzxyz...) -> locally SOA (xx...xyy...yzz...z)

Helps coalesce global atomics

Only helps when using consolidated queue

Implication: Maxwell throttles more easily than Kepler

<table>
<thead>
<tr>
<th>OPT 1:1</th>
<th>AOS</th>
<th>SOA</th>
<th>OPT 1:3</th>
<th>AOS</th>
<th>SOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20</td>
<td>9.40 ms</td>
<td>9.73 ms</td>
<td>K20</td>
<td>6.56 ms</td>
<td>6.31 ms</td>
</tr>
<tr>
<td>TITAN X</td>
<td>4.26 ms</td>
<td>4.24 ms</td>
<td>TITAN X</td>
<td>4.15 ms</td>
<td>3.78 ms</td>
</tr>
</tbody>
</table>
Optimization 4 – Half Precision Floats

Use half-precision coordinates and arithmetic for distance checking

Force evaluation for non-zero forces still done in full precision

The benefit mostly comes from bandwidth saving

<table>
<thead>
<tr>
<th></th>
<th>OPT 1:3</th>
<th>OPT 1:4</th>
<th>↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20</td>
<td>6.31 ms</td>
<td>6.01 ms</td>
<td>5%</td>
</tr>
<tr>
<td>TITAN X</td>
<td>3.78 ms</td>
<td>3.76 ms</td>
<td>1%</td>
</tr>
</tbody>
</table>
Push for the last drop of performance

Suspected compiler inefficiency

- Shared LD/ST addressing
- Chained predication
- No C++ equivalence of a predicate register

**OPT 1:4 vs. OPT 1:5**

<table>
<thead>
<tr>
<th></th>
<th>OPT 1:4</th>
<th>OPT 1:5</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20</td>
<td>6.01 ms</td>
<td>5.64 ms</td>
<td>↓ 6%</td>
</tr>
<tr>
<td>TITAN X</td>
<td>3.76 ms</td>
<td>3.79 ms</td>
<td>↑ 1%</td>
</tr>
</tbody>
</table>

Solution: Inline PTX

**Examples**

<table>
<thead>
<tr>
<th>C Code</th>
<th>PTX Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>array[wid][9].y</td>
<td>ld.shared.s32 ..., [array + 9*8 + 4];</td>
<td>Avoids runtime address calculation</td>
</tr>
<tr>
<td>bool interacting = ...;</td>
<td>.reg .pred interacting</td>
<td>Eliminates predicate/Boolean conversion</td>
</tr>
<tr>
<td>overview = __ballot(interacting );</td>
<td>setp.s32 {...} interacting</td>
<td></td>
</tr>
<tr>
<td>if ( interacting ) queue[...] = ...;</td>
<td>vote.ballot.b32 overview, interacting</td>
<td></td>
</tr>
<tr>
<td>@interacting st.volatile.shared.s32 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Yu-Hang Tang  yuhang_tang@brown.edu  Karniadakis Group  Brown University  |  GPU Tech Conf 2016 S6140  April 6, 2016
Summary

Optimization techniques at both algorithmic and implementation levels were applied to speed up the Dissipative Particle Dynamics pairwise force kernel by 2.3x on Kepler and 1.5x on Maxwell.

Fast global atomics eliminates the need to compute each pair twice as concluded by earlier CUDA programming experience.

Fast FPRNGs may have potential applications in GPU-based scientific computing, graphics rendering etc.

Intra-warp communication vital for efficient work migrating and branching suppression.

More texture units need for particle-based simulations.
Thank you!

GK acknowledges support by the DOE INCITE project bip118 and by the National Institutes of Health (NIH) Grant U01HL114476. YHT appreciates the GPU hardware equipment from Mark Berge and Brad Nemire. We thank Jack Wells and Don Maxwell as well as all the Titan support staff at the Oak Ridge National Laboratory (ORNL) that made this possible. This research used resources of the Oak Ridge Leadership Computing Facility at the ORNL, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.