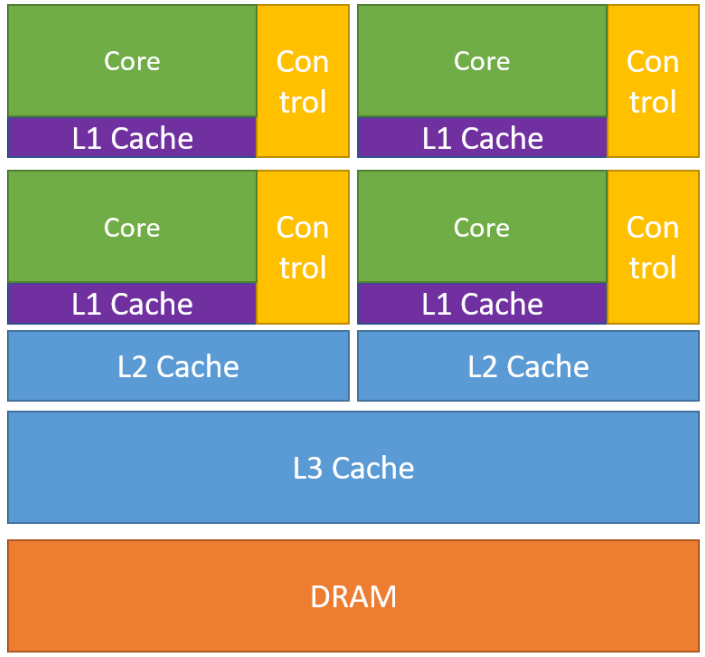
CUDA Introduction

俞锦程

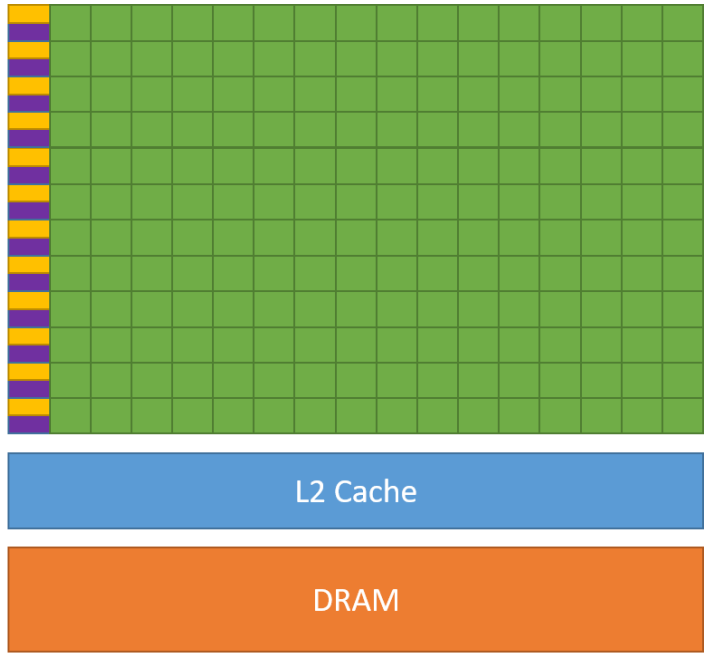
2021-9-26

What is GPU

- Graphics Processing Unit (CPU: Central Processing Unit)
  - 特殊类型的处理器，核数很大，适合并行运行大量计算
  - 图形渲染
  - 高性能计算（HPC）
  - 深度学习，机器学习



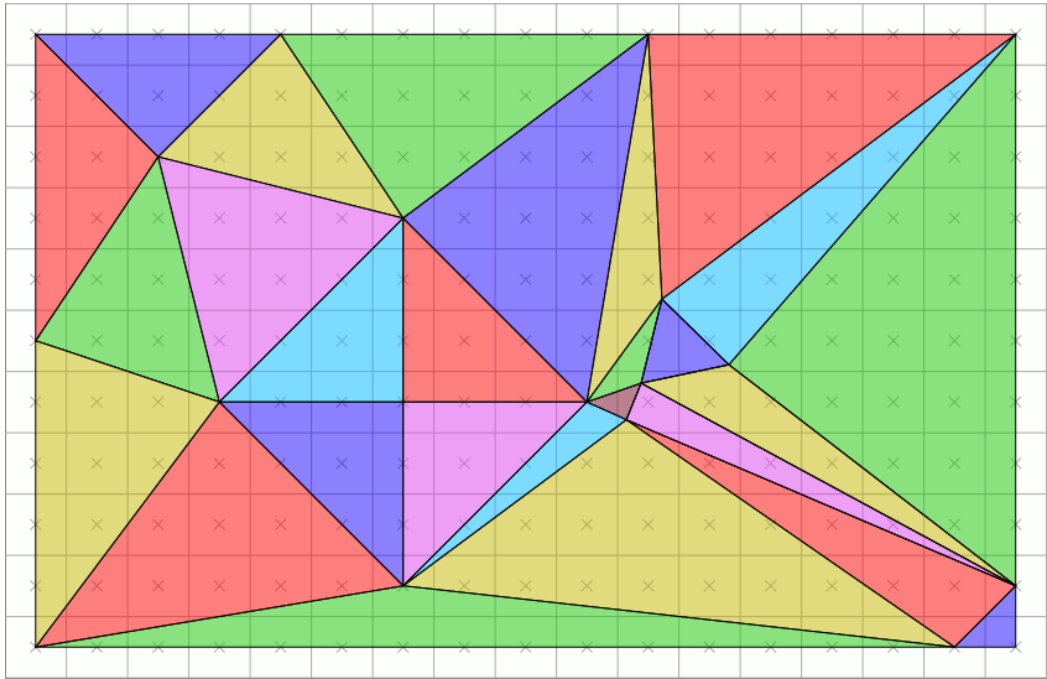
CPU



GPU

Some histories

- 最初的图形渲染由软件完成，效率太低
- 顶点计算、光栅化交由硬件完成
- 图形渲染的处理器 – GPU
- 大量的核，并行计算
- GPGPU (General Purpose GPU)
  - GPU从专用处理器→ “通用”处理器



Brief Timeline

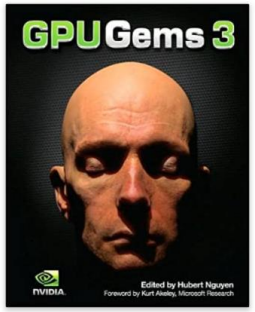
A *scandalously* brief history of GPUs

Year	Transistors	Model	Tech
1999	25M	GeForce 256	DX7, OpenGL
2001	60M	GeForce 3	Programmable Shader
2002	125M	GeForce FX	Cg programs
2006	681M	GeForce 8800	C for CUDA
2008	1.4G	GeForce GTX 280	IEEE FP
2010	3.0G	Fermi	Cache, C++

Histories of GPGPU

- 用OpenGL/DirectX API实现计算 (2001)
  - 数组 = 纹理 (texture)
  - 内核 = 着色器 (shader)
  - 运算 = 绘图
- CUDA (2006)
  - 通用计算的原生支持

Books › Computers & Technology › Digital Audio, Video & Photography



GPU Gems 3 (text only) by H.Nguyen Hardcover – January 1, 2007

by H.Nguyen (Author)

9 ratings

See all formats and editions

Hardcover
$920.99

4 Used from $95.76
1 New from $920.99

价格平稳 降价提醒

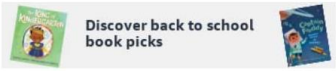
GPU Gems 3 [Hardcover]Hubert Nguyen (Author)

Publisher: Addison-Wesley Professional

Publication date: January 1, 2007

See all details

See this image



Frequently bought together

+

Total price: $1,004.38

Add both to Cart

One of these items ships sooner than the other. Show details

arXiv.org > hep-lat > arXiv:0811.2111

High Energy Physics - Lattice

[Submitted on 13 Nov 2008]

GPU computing for 2-d spin systems: CUDA vs OpenGL

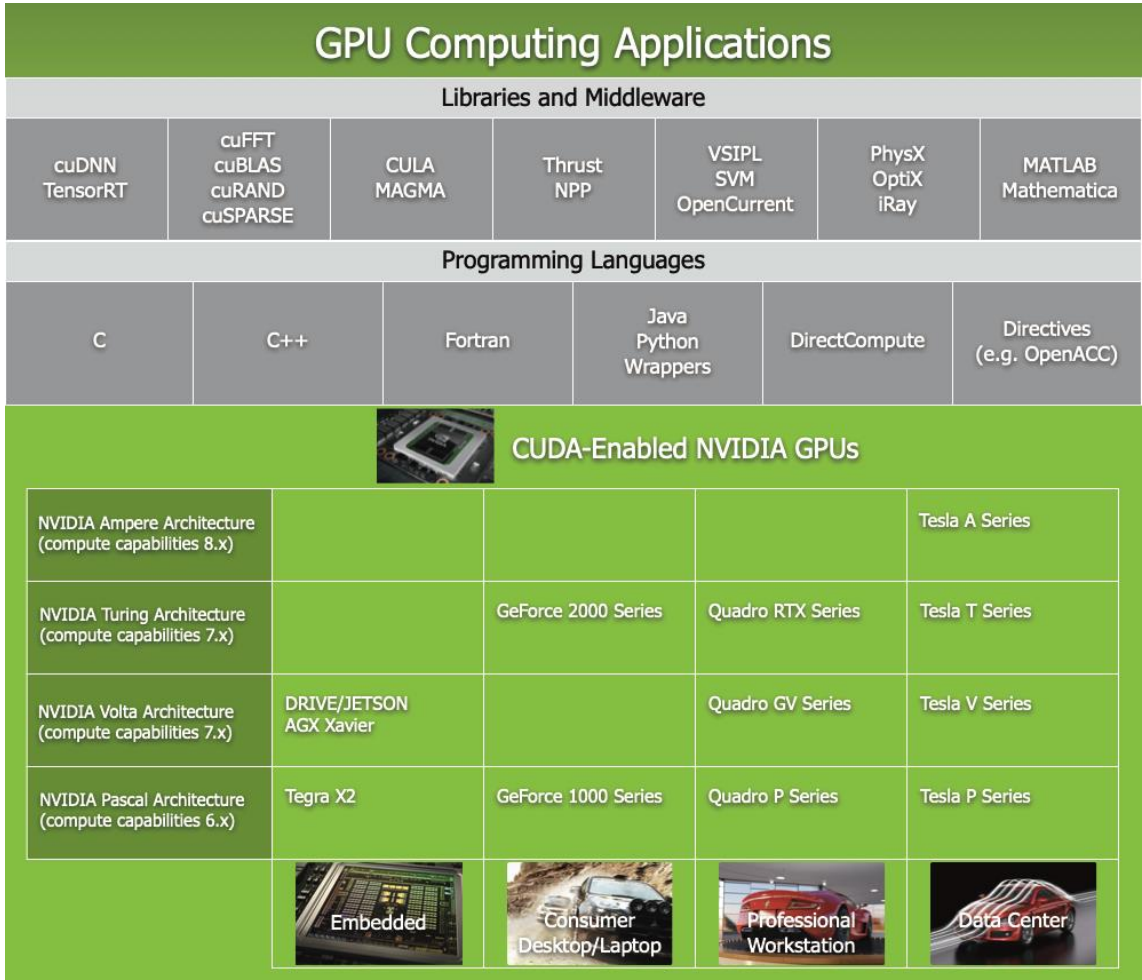
Viola Anselmi, Giovanni Conti, Francesco Di Renzo

Lattice size	Gain (CUDA)	Gain (OpenGL
128	33	1
256	37	4
512	38	14
1024	30	41
2048	30	n.a.

Table 1: Gain factors (ratios of execution times) with respect to the serial code.

What is CUDA, why CUDA

- Compute Unified Device Architecture
  - 异构的编程语言，CPU部分，GPU部分
  - 分散读取
  - 统一虚拟内存
  - 共享内存
  - 更大带宽

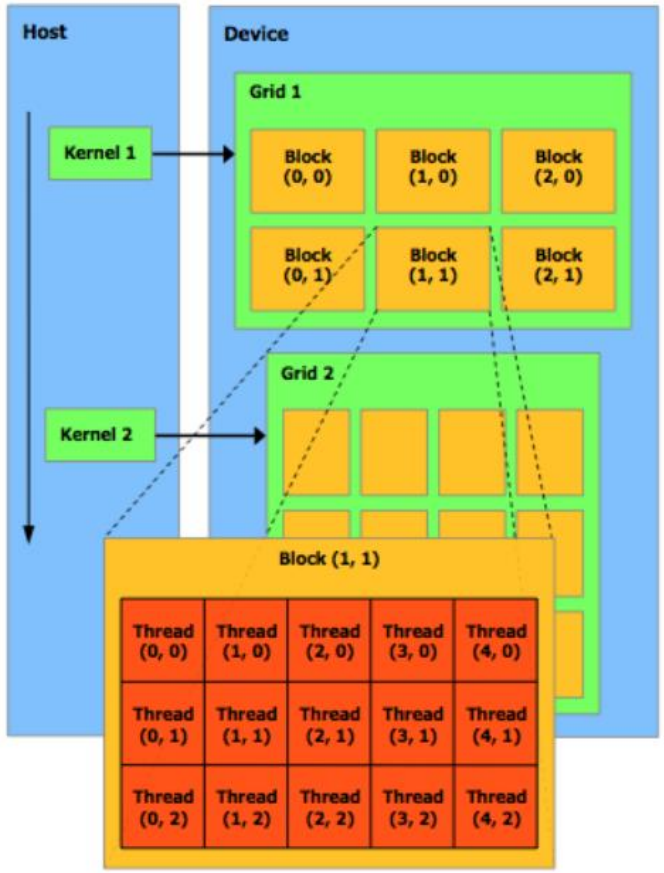


GPGPU

- CUDA (NVidia, AMD, **Iluvatar**, …)
  - Many libraries: cuBLAS, cuFFT, cuDNN…
- OpenCL
  - Open source, on many different platforms
- HIP (AMD GPU)
  - CUDA-like

A simple CUDA case

- CUDA code中的for loop去哪了

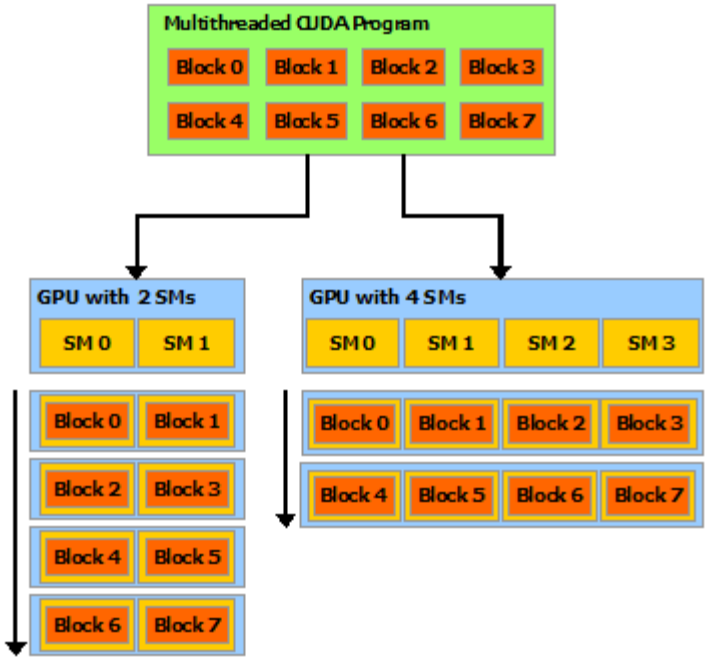



```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
  for (int i = 0; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
// Invoke the serial function:
saxpy_serial(n, 2.0, x, y);


__global__
void saxpy_cuda(int n, float alpha, float *x, float *y)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
// Invoke the cuda kernel:
int nblocks = (n + 255) / 256;
saxpy_cuda<<<nblocks, 256>>>(n, 2.0, x, y);
```

Programming model

- 异构的编程模型，CPU部分，GPU部分
- 硬件上有多个流多处理器SM (Streaming Multi-processor)，每个SM有多个核 (CUDA core)
- 编程模型上将任务分为一定大小的grid，每个grid有若干个block，每个block包含若干个thread



For loop隐藏在这!

Case study (I) – nobody simulation

- Chamomile Scheme (Hamada, T., & Iitaka, T., 2007, arXiv:astro-ph/0703100)

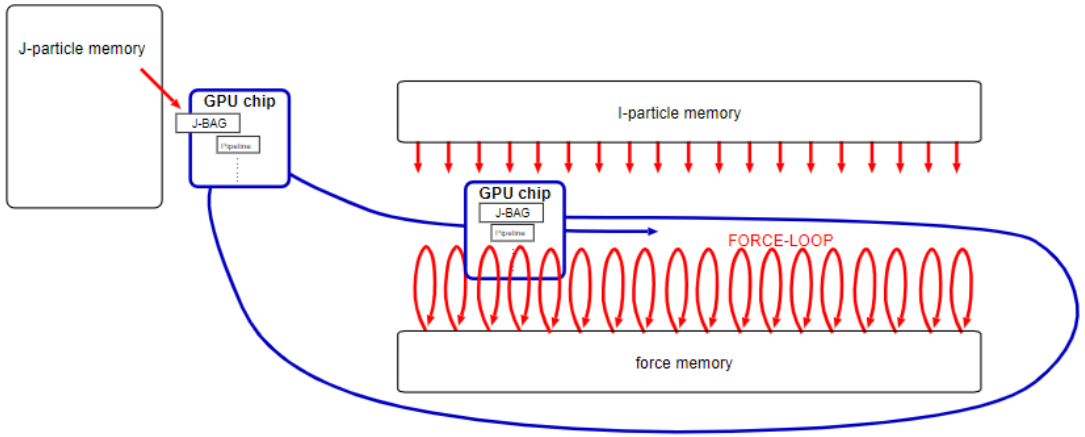
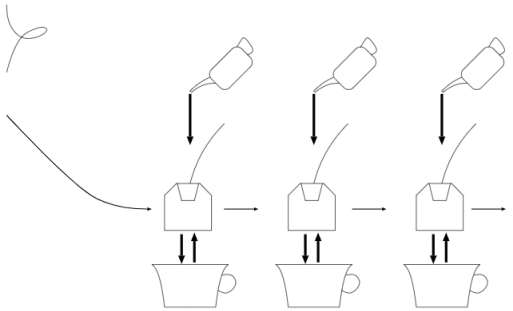


Fig. 3. Overview of the Chamomile Scheme.



```
__device__ float3
TwoBodyInteraction(struct star si, struct star sj, float3 fi)
{
    // calculate (unit mass) force (accelaration) of star i from star j
    // f.x = m_j * r_ij.x / r_ij^1.5
    float3 r;
    r.x = sj.x - si.x;
    r.y = sj.y - si.y;
    r.z = sj.z - si.z;
    float dis_sq = r.x * r.x + r.y * r.y + r.z * r.z;
    float dis = sqrt(dis_sq);
    fi.x += sj.m * r.x / (dis_sq * dis);
    fi.y += sj.m * r.y / (dis_sq * dis);
    fi.z += sj.m * r.z / (dis_sq * dis);
    return fi;
}

__global__ void
force(int n, struct star *x, float3 *a)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) {
        for (int i = 0; i < n; ++i) {
            a[tid] = TwoBodyInteraction(x[tid], x[i], a[tid]);
        }
    }
}
```

Case study (II) – likelihood function

- Likelihood: likelihood for each obs data (i) over all model data (j).

$$\Sigma_j\left(e^{-0.5b\times\left(x_{obs}^{(i)}-x_{model}^{(j)}\right)^2}\right)$$

cpu code

```
void likelihood_f(int nm, float *xm, float *ym, int nd, float *xd, float *yd, float *xsig, f
loat *ysig, float *p)
{
    for (int i = 0; i < nd; ++i) {
        float z = 0.0f;
        float t = 2 * PI * xsig[i] * ysig[i];
        for (int j = 0; j < nm; ++j) {
            float temp = (xd[i] - xm[j]) * (xd[i] - xm[j]) + (yd[i] - ym[j]) * (yd[i] - ym[j
]);
            z += expf(-0.5f * temp) / t;
        }
        p[i] = z;
    }
}
```

```
// one data point: one thread
void __global__
likelihood_kernel1(int nm, float *xm, float *ym, float *wm, int nd, float *xd, float *yd, fl
oat *xs, float *ys, float rc2, float *p)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < nd) {
        float a = 2 * PI * xs[tid] * ys[tid];
        for (int i = 0; i < nm; ++i) {
            float b = SQ(xd[tid] - xm[i]) + \
                      SQ(yd[tid] - ym[i]);
            if (b > rc2) continue;
            float c = expf(-0.5f * b) * wm[i] / a;
            p[tid] += c;
        }
    }
}
```

Host code

```
extern "C"
void likelihood_gpu(int nm, float *xm, float *ym, float *wm, int nd, float *xd, float *yd, f
loat *xs, float *ys, float rc, float *p)
{
    if (BLOCKSIZE % NT != 0) {
        fprintf(stderr, "BLOCKSIZE and NT should be aligned\n");
    } else if (NT > BLOCKSIZE) {
        fprintf(stderr, "warning: %d (NT) > %d (BLOCKSIZE)\n", NT, BLOCKSIZE);
    }
    int BLOCKNUM = (nd + BLOCKSIZE - 1) / BLOCKSIZE * NT;
    float rc2 = rc*rc;
    float *dxm, *dym, *dwm;
    float *dxd, *dyd, *dxs, *dys;
    float *dp;
    cudaMalloc((void **)&dxm, nm * sizeof(float));
    cudaMalloc((void **)&dym, nm * sizeof(float));
    cudaMalloc((void **)&dwm, nm * sizeof(float));
    cudaMalloc((void **)&dxd, nd * sizeof(float));
    cudaMalloc((void **)&dyd, nd * sizeof(float));
    cudaMalloc((void **)&dxs, nd * sizeof(float));
    cudaMalloc((void **)&dys, nd * sizeof(float));
    cudaMalloc((void **)&dp,  nd * sizeof(float));

    cudaMemcpy(dxm, xm, nm * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dym, ym, nm * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dwm, wm, nm * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dxd, xd, nd * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dyd, yd, nd * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dxs, xs, nd * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dys, ys, nd * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemset(dp, 0, nd * sizeof(float));
    likelihood_kernel3<<<BLOCKNUM, BLOCKSIZE>>>(nm, dxm, dym, dwm, nd, dxd, dyd, dxs, dys, r
c2, dp);
    cudaMemcpy(p, dp, nd * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(dxm);
    cudaFree(dym);
    cudaFree(dwm);
    cudaFree(dxd);
    cudaFree(dyd);
    cudaFree(dxs);
    cudaFree(dys);
    cudaFree(dp);
}
```

Shared memory

- 读写速度更快，延迟更低
  - 对于一个block内需要反复用到的数据可以充分利用shared memory
- 大小有限制，需要合理规划block大小

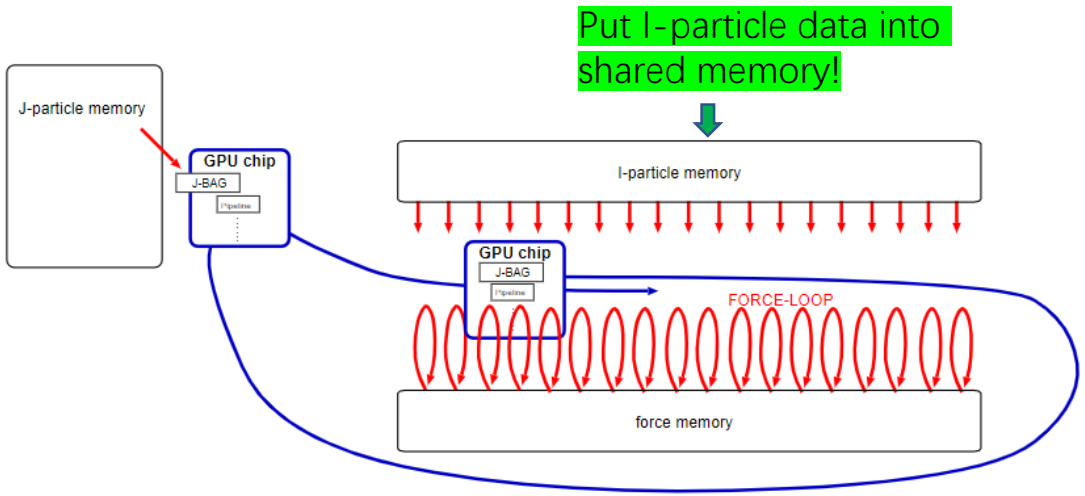


Fig. 3. Overview of the Chamomile Scheme.

Advanced scheme

```
// one data point: 2 threads
void __global__
likelihood_kernel2(int nm, float *xm, float *ym, float *wm, int nd, float *xd, float *yd, fl
oat *xs, float *ys, float rc2, float *p)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int data_idx = tid >> 1;
    int d_offset = threadIdx.x >> 1;
    int m_offset = threadIdx.x & 1;
    //extern __shared__ float psub[];
    __shared__ float psub[BLOCKSIZE/2];
    if (m_offset == 0) {
        psub[d_offset] = 0;
    }
    if (data_idx < nd) {
        float a = 2 * PI * xs[data_idx] * ys[data_idx];
        for (int i = 0; i < nm; i += NT) {
            if (i + m_offset >= nm) continue;
            float b = SQ(xd[data_idx] - xm[i+m_offset]) + \
                      SQ(yd[data_idx] - ym[i+m_offset]);
            if (b > rc2) continue;
            float c = expf(-0.5f * b) * wm[i+m_offset] / a;
            atomicAdd(psub + d_offset, c);
        }
        p[data_idx] = psub[d_offset];
    }
}
```

Advanced scheme

```
// one data point: multiple threads [4, 8, 16, ..., BLOCKSIZE/2]
void __global__
likelihood_kernel3(int nm, float *xm, float *ym, float *wm, int nd, float *xd, float *yd, fl
oat *xs, float *ys, float rc2, float *p)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int data_idx = tid / NT;
    int d_offset = threadIdx.x / NT;
    int m_offset = threadIdx.x % NT;
    __shared__ float psub[BLOCKSIZE];
#ifndef NOUSE_SHM
    __shared__ float xdsub[BLOCKSIZE/NT], ydsub[BLOCKSIZE/NT];
    __shared__ float xssub[BLOCKSIZE/NT], yssub[BLOCKSIZE/NT];
    if (m_offset == 0) {
        xdsub[d_offset] = xd[data_idx];
        ydsub[d_offset] = yd[data_idx];
        xssub[d_offset] = xs[data_idx];
        yssub[d_offset] = ys[data_idx];
    }
    __syncthreads();
#endif
    psub[threadIdx.x] = 0;
    if (data_idx < nd) {
#ifdef NOUSE_SHM
        float a = 2 * PI * xs[data_idx] * ys[data_idx];
#else
        float a = 2 * PI * xssub[d_offset] * yssub[d_offset];
#endif
        for (int i = 0; i < nm; i += NT) {
            if (i + m_offset >= nm) continue;
#ifdef NOUSE_SHM
            float b = SQ(xd[data_idx] - xm[i+m_offset]) + \
                      SQ(yd[data_idx] - ym[i+m_offset]);
#else
            float b = SQ(xdsub[d_offset] - xm[i+m_offset]) + \
                      SQ(ydsub[d_offset] - ym[i+m_offset]);
#endif
            if (b > rc2) continue;
            float c = expf(-0.5f * b) * wm[i+m_offset] / a;
            psub[threadIdx.x] += c;
        }
        // reduce sum
        for (int i = NT / 2; i > 0; i>>=1) {
            __syncthreads();
            if (m_offset < i) {
                psub[threadIdx.x] += psub[threadIdx.x + i];
            }
        }
        if (m_offset == 0) {
            p[data_idx] = psub[threadIdx.x];
        }
    }
}
```

Advanced scheme

```
// one data point: one block
void __global__
likelihood_kernel4(int nm, float *xm, float *ym, float *wm, int nd, float *xd, float *yd, fl
oat *xs, float *ys, float rc2, float *p)
{
    int data_idx = blockIdx.x;
    int m_offset = threadIdx.x;
    //extern __shared__ float psub[];
    __shared__ float psub[BLOCKSIZE];
    __shared__ float xdsub, ydsub;
    __shared__ float xssub, yssub;
    if (threadIdx.x == 0) {
        xdsub = xd[data_idx];
        ydsub = yd[data_idx];
        xssub = xs[data_idx];
        yssub = ys[data_idx];
    }
    __syncthreads();
    psub[threadIdx.x] = 0;
    if (data_idx < nd) {
        //float a = 2 * PI * xs[data_idx] * ys[data_idx];
        float a = 2 * PI * xssub * yssub;
        for (int i = 0; i < nm; i += BLOCKSIZE) { // or i += blockDim.x
            if (i + m_offset >= nm) continue;
            float b = SQ(xdsub - xm[i+m_offset]) + \
                      SQ(ydsub - ym[i+m_offset]);
            if (b > rc2) continue;
            float c = expf(-0.5f * b) * wm[i+m_offset] / a;
            psub[threadIdx.x] += c;
        }
        // reduce sum
        for (int i = BLOCKSIZE / 2; i > 0; i >>= 1) {
            __syncthreads();
            if (threadIdx.x < i) {
                psub[threadIdx.x] += psub[threadIdx.x + i];
            }
        }
        if (threadIdx.x == 0) {
            p[data_idx] = psub[threadIdx.x];
        }
    }
}
```

Performance guidelines

- 提高并行度以提高硬件资源利用
- 优化内存访问
- 优化指令
- 优化内存传输

我不会C/C++，只会python…

- Numba for CUDA GPUs
  - Vector add

```python
import numpy as np
from numba import cuda

@cuda.jit
def gpu_add(a, b, result, n):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if idx < n:
        result[idx] = a[idx] + b[idx]

def main():
    n = 1e6
    x = np.random.rand(n)
    y = np.random.rand(n)

    gpu_res = np.zeros(n)
    cpu_res = np.zeros(n)

    nthreads = 256
    nblocks  = np.ceil(n / nthreads).astype(int)
    gpu_add[nblocks, nthreads](x, y, gpu_res, n)
    cuda.synchronize()

    # cpu add
    cpu_res = np.add(x, y) # x + y
```

Numba case (III) – matrix multiplication

```python
from numba import cuda, float32

# Controls threads per block and shared memory usage.
# The computation will be done on blocks of TPBxTPB elements.
TPB = 16

@cuda.jit
def fast_matmul(A, B, C):
    # Define an array in the shared memory
    # The size and type of the arrays must be known at compile time
    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x    # blocks per grid

    if x >= C.shape[0] and y >= C.shape[1]:
        # Quit if (x, y) is outside of valid C boundary
        return

    # Each thread computes one element in the result matrix.
    # The dot product is chunked into dot products of TPB-long vectors.
    tmp = 0.
    for i in range(bpg):
        # Preload data into shared memory
        sA[tx, ty] = A[x, ty + i * TPB]
        sB[tx, ty] = B[tx + i * TPB, y]

        # Wait until all threads finish preloading
        cuda.syncthreads()

        # Computes partial product on the shared memory
        for j in range(TPB):
            tmp += sA[tx, j] * sB[j, ty]

        # Wait until all threads finish computing
        cuda.syncthreads()

    C[x, y] = tmp
```

- 对用户并不友好，需要知道硬件细节（以及懂C/C++）……

Frameworks

- Tensorflow (机器学习框架，支持CUDA)
- Pytorch （机器学习框架，支持CUDA)
- Caffe （深度学习框架，支持CUDA)
- Paddlepaddle (百度深度学习框架，支持CUDA，ROCm)
- ...

Case study (IV)

- FFT

```
import torch
t = torch.arange(4)
torch.fft.fft(t)

cuda_device = torch.device("cuda")
x_d = torch.randn(16, 32, device=cuda_device)
torch.fft.fft(x_d)
```