

# GPU Parallelization



## 00\_hello.cu

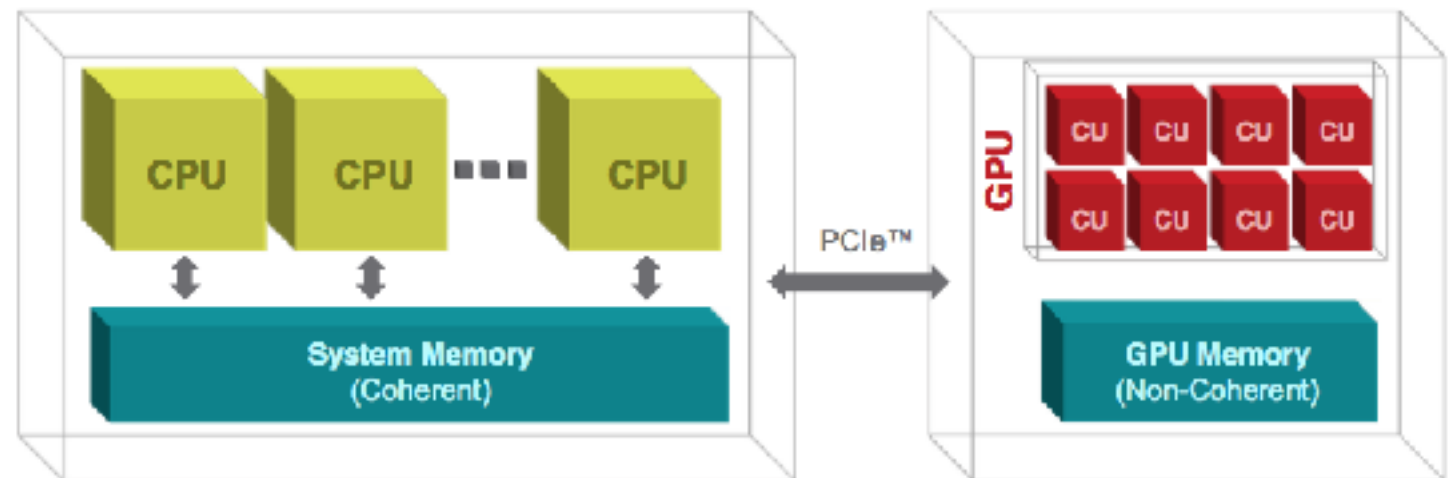
```
__global__ void print(void) {  
    printf("Hello GPU\n");  
}  
  
int main() {  
    printf("Hello CPU\n");  
    print<<<1,1>>>();  
}
```

nvcc -arch=sm\_60 00\_hello.cu

# 01\_memcpy.cu

```
__global__ void kernel(float *a) {  
    a[threadIdx.x] = threadIdx.x;  
}
```

```
int main(void) {  
    int size = 4 * sizeof(float);  
    float *a, *b = (float*) malloc(size);  
    cudaMalloc(&a, size);  
    kernel<<<1,4>>>(a);  
    cudaMemcpy(b, a, size, cudaMemcpyDeviceToHost);  
    for (int i=0; i<4; i++) printf("%f\n",b[i]);  
    cudaFree(a);  
    free(b);  
    return 0;  
}
```

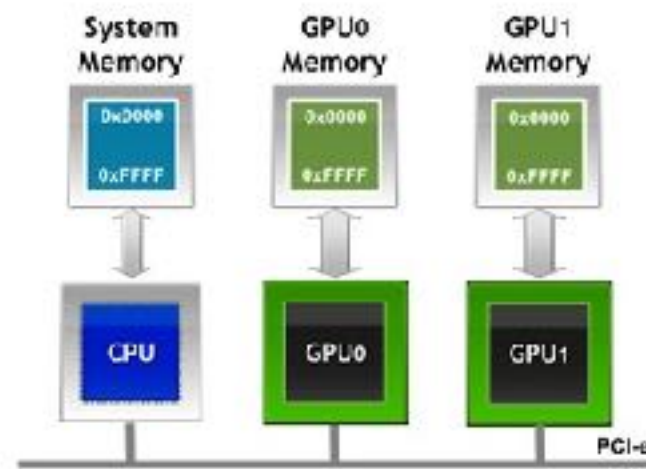


## 02\_unified.cu

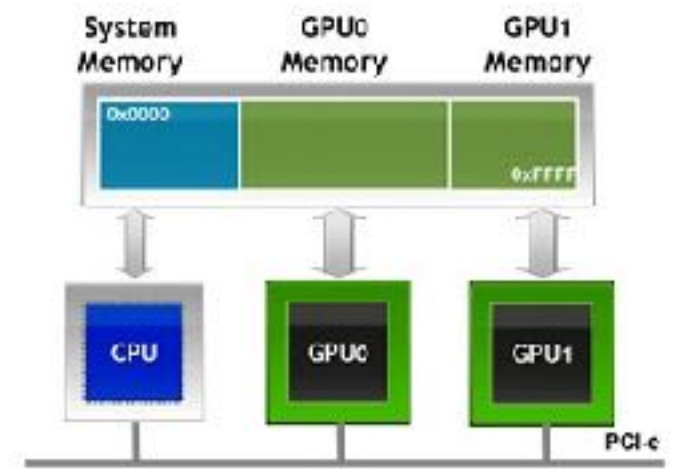
```
__global__ void kernel(float *a) {  
    a[threadIdx.x] = threadIdx.x;  
}
```

```
int main(void) {  
    int size = 4 * sizeof(float);  
    float *a;  
    cudaMallocManaged(&a, size);  
    kernel<<<1,4>>>(a);  
    cudaDeviceSynchronize();  
    for (int i=0; i<4; i++) printf("%f\n",a[i]);  
    cudaFree(a);  
    return 0;  
}
```

*No UVA: Multiple Memory Spaces*



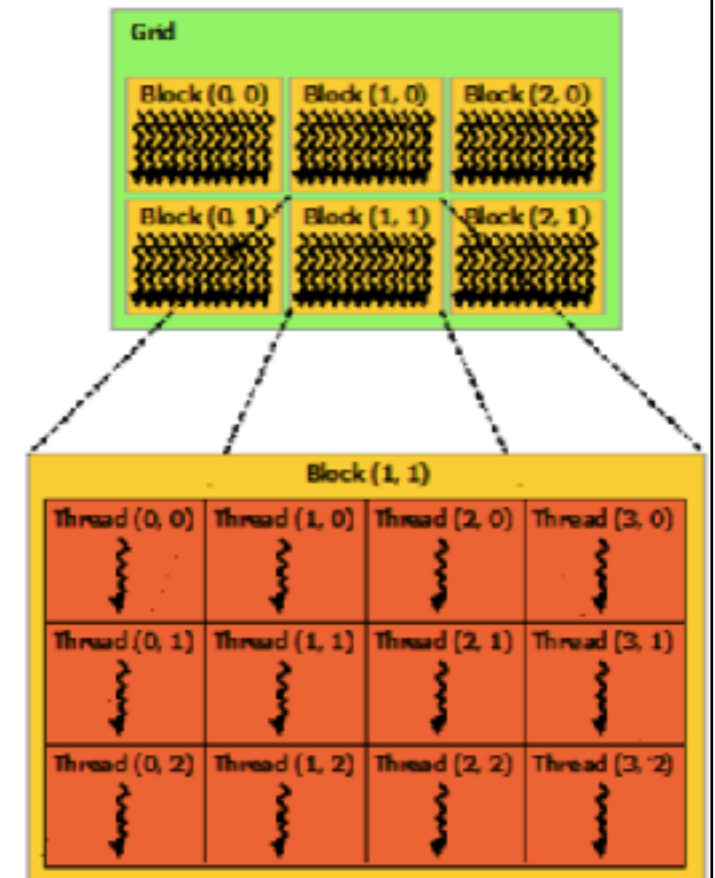
*UVA: Single Address Space*



# 03\_thread\_block.cu

```
__global__ void kernel(float *a) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    a[i] = 10 * blockIdx.x + threadIdx.x;  
}
```

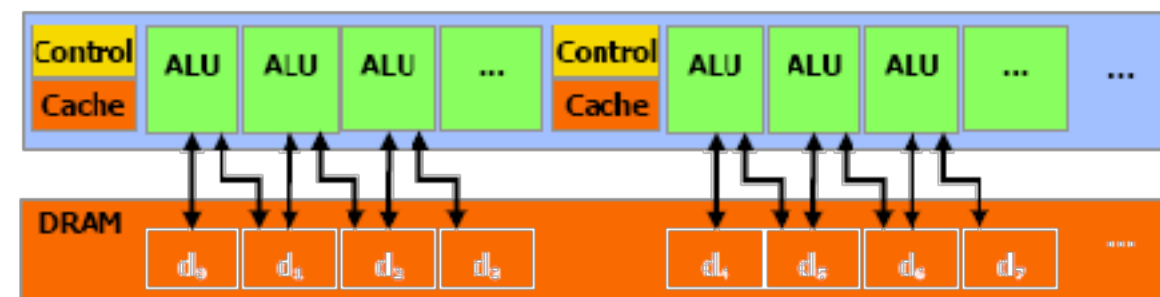
```
int main(void) {  
    int size = 4 * sizeof(float);  
    float *a;  
    cudaMallocManaged(&a, size);  
    kernel<<<2,2>>>(a);  
    cudaDeviceSynchronize();  
    for (int i=0; i<4; i++) printf("%f\n",a[i]);  
    cudaFree(a);  
    return 0;  
}
```



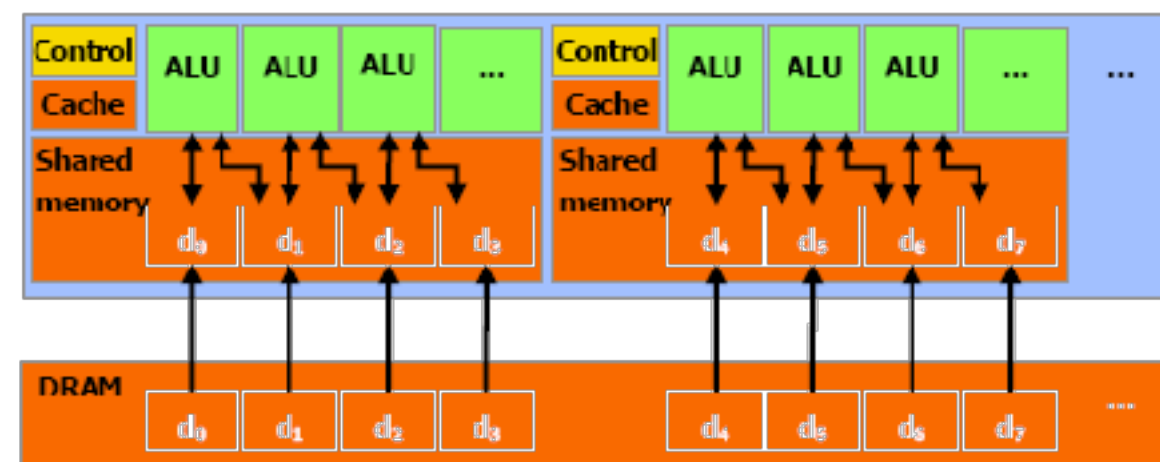
## 04\_shared.cu

```
__global__ void kernel(float *a) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    __shared__ b[2];  
    b[threadIdx.x] = 10 * blockIdx.x + threadIdx.x;  
    __syncthreads();  
    a[i] = b[threadIdx.x];  
}
```

```
int main(void) {  
    int size = 4 * sizeof(float);  
    float *a;  
    cudaMallocManaged(&a, size);  
    kernel<<<2,2>>>(a);  
    cudaDeviceSynchronize();  
    for (int i=0; i<4; i++) printf("%f\n", a[i]);  
    cudaFree(a);  
    return 0;  
}
```



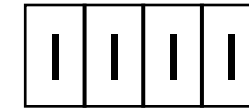
Without shared memory



With shared memory

## 05\_sum.cu

```
__global__ void kernel(float *a, float *sum) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    atomicAdd(sum, a[i]);  
}
```

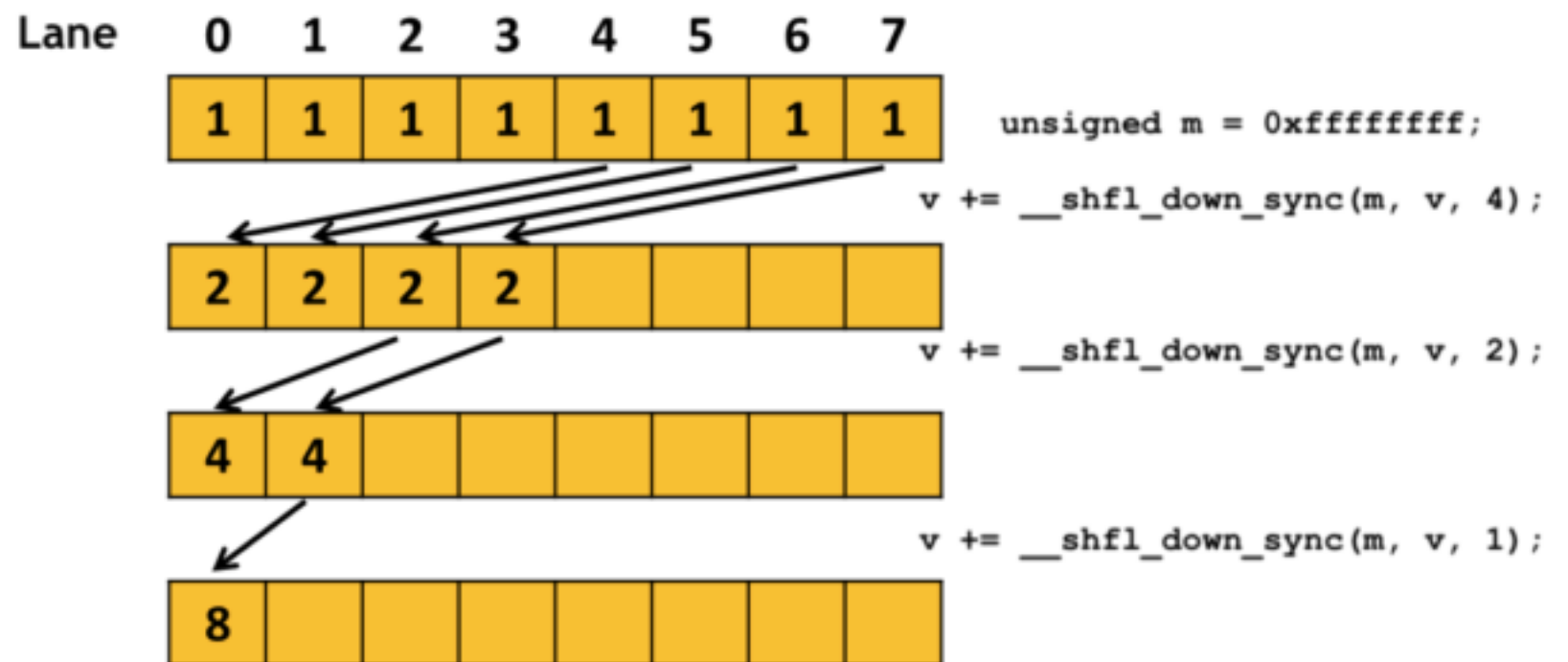
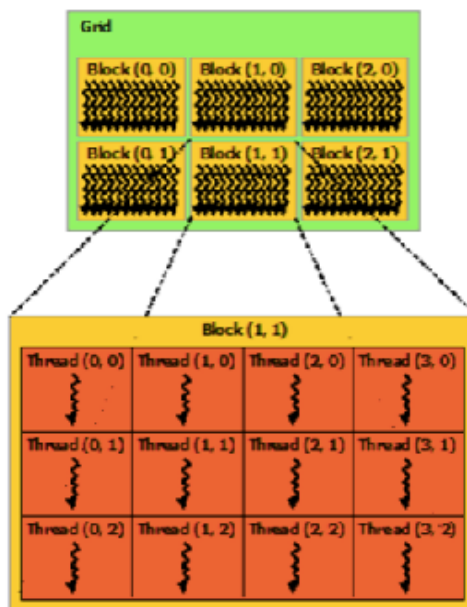


$$1+1+1+1=4$$

```
int main(void) {  
    int size = 4 * sizeof(float);  
    float *a, *sum;  
    cudaMallocManaged(&a, size);  
    cudaMallocManaged(&sum, sizeof(float));  
    for (int i=0; i<4; i++) a[i] = 1;  
    kernel<<<2,2>>>(a, sum);  
    cudaDeviceSynchronize();  
    printf("%f\n", *sum);  
    cudaFree(a);  
    cudaFree(sum);  
    return 0;  
}
```

# 06\_warp\_sum.cu

```
__inline__ __device__
int warpSum(float sum) {
    for (int offset=16; offset>0; offset >>= 1)
        sum += __shfl_down_sync(0xffffffff, sum, offset);
    return sum;
}
```

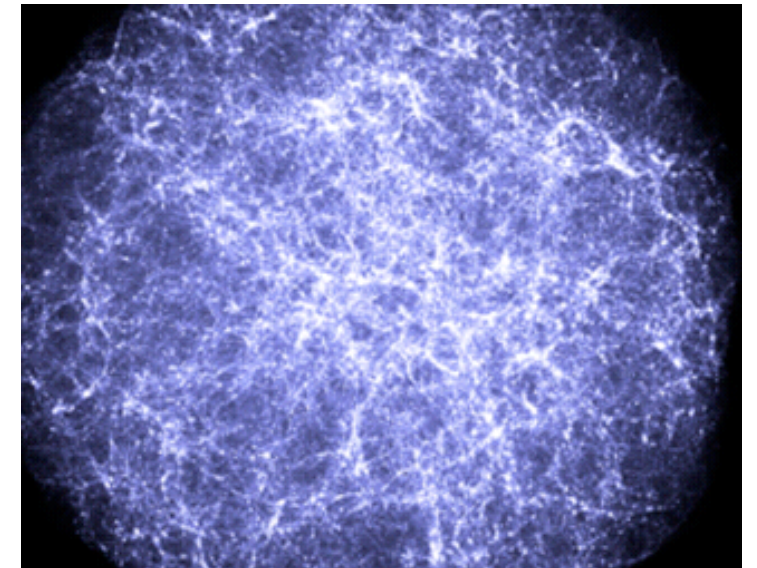


```
__global__ void kernel(float *a, float *sum) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float b = warpSum(a[i]);
    if ((threadIdx.x & 31) == 0) atomicAdd(sum, b);
}
```



# 07\_nbody.cu

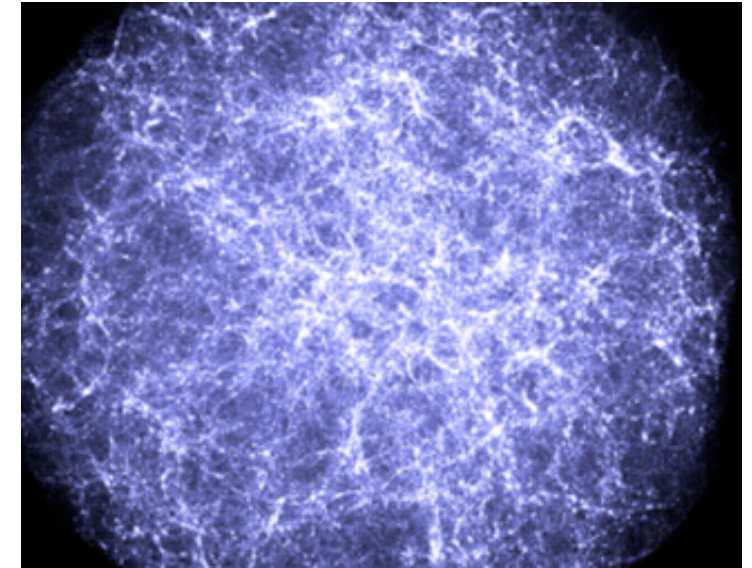
```
__global__ void GPUkernel(int N, float * x, float * y, float * z, float * m,
                          float * p, float * ax, float * ay, float * az, float eps2) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float pi = 0;
    float axi = 0;
    float ayi = 0;
    float azi = 0;
    float xi = x[i];
    float yi = y[i];
    float zi = z[i];
    for( int j=0; j<N; j++ ) {
        float dx = x[j] - xi;
        float dy = y[j] - yi;
        float dz = z[j] - zi;
        float R2 = dx * dx + dy * dy + dz * dz + eps2;
        float invR = rsqrtf(R2);
        pi += m[j] * invR;
        float invR3 = m[j] * invR * invR * invR;
        axi += dx * invR3;
        ayi += dy * invR3;
        azi += dz * invR3;
    }
    p[i] = pi;
    ax[i] = axi;
    ay[i] = ayi;
    az[i] = azi;
}
```



```
nvcc -arch=sm_60 -Xcompiler "-Wall -fopenmp -O3 -ffast-math" 07_nbody.cu
```

# 08\_nbody\_shared.cu

```
__global__ void GPUkernel(int N, float * x, float * y, float * z, float * m,
                          float * p, float * ax, float * ay, float * az, float eps2) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float pi = 0;
    float axi = 0;
    float ayi = 0;
    float azi = 0;
    float xi = x[i];
    float yi = y[i];
    float zi = z[i];
    __shared__ float xj[THREADS], yj[THREADS], zj[THREADS], mj[THREADS];
    for ( int jb=0; jb<N/blockDim.x; jb++ ) {
        __syncthreads();
        xj[threadIdx.x] = x[jb*blockDim.x+threadIdx.x];
        yj[threadIdx.x] = y[jb*blockDim.x+threadIdx.x];
        zj[threadIdx.x] = z[jb*blockDim.x+threadIdx.x];
        mj[threadIdx.x] = m[jb*blockDim.x+threadIdx.x];
        __syncthreads();
#pragma unroll
        for( int j=0; j<blockDim.x; j++ ) {
            float dx = xj[j] - xi;
            float dy = yj[j] - yi;
            float dz = zj[j] - zi;
            float R2 = dx * dx + dy * dy + dz * dz + eps2;
            float invR = rsqrtf(R2);
            pi += mj[j] * invR;
            float invR3 = mj[j] * invR * invR * invR;
            axi += dx * invR3;
            ayi += dy * invR3;
            azi += dz * invR3;
        }
    }
    p[i] = pi;
    ax[i] = axi;
    ay[i] = ayi;
    az[i] = azi;
}
```



# 00\_kernel.cpp

```
int main(void) {
    int size = 4 * sizeof(float);
    float *a = (float*) malloc(size);
#pragma acc kernels
    for (int i=0; i<4; i++) a[i] = i;
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    free(a);
    return 0;
}
```

```
__global__ void kernel(float *a) {
    a[threadIdx.x] = threadIdx.x;
}

int main(void) {
    int size = 4 * sizeof(float);
    float *a;
    cudaMallocManaged(&a, size);
    kernel<<<1,4>>>(a);
    cudaDeviceSynchronize();
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    cudaFree(a);
    return 0;
}
```

# 00\_kernel.cpp

OpenACC

```
int main(void) {
    int size = 4 * sizeof(float);
    float *a = (float*) malloc(size);
#pragma acc kernels
    for (int i=0; i<4; i++) a[i] = i;
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    free(a);
    return 0;
}
```

pgc++ -acc -Minfo=accel 00\_kernel.cpp

CUDA

```
__global__ void kernel(float *a) {
    a[threadIdx.x] = threadIdx.x;
}

int main(void) {
    int size = 4 * sizeof(float);
    float *a;
    cudaMallocManaged(&a, size);
    kernel<<<1,4>>>(a);
    cudaDeviceSynchronize();
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    cudaFree(a);
    return 0;
}
```

nvcc -arch=sm\_60 02\_unified.cu

# 01\_gang.cpp

OpenACC

```
int main(void) {
    int size = 4 * sizeof(float);
    float *a = (float*) malloc(size);
#pragma acc kernels loop gang(2) vector(2)
    for (int i=0; i<4; i++) a[i] = i;
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    free(a);
    return 0;
}
```

CUDA

```
int main(void) {
    int size = 4 * sizeof(float);
    float *a;
    cudaMallocManaged(&a, size);
    kernel<<<2,2>>>(a);
    cudaDeviceSynchronize();
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    cudaFree(a);
    return 0;
}
```

**gang=block, worker=warp, vector=thread**

## 02\_copy.cpp

OpenACC

```
int main(void) {
    int size = 4 * sizeof(float);
    float *a = (float*) malloc(size);
#pragma acc kernels loop gang(2) vector(2) copyout(a)
    for (int i=0; i<4; i++) a[i] = i;
    for (int i=0; i<4; i++) printf("%f\n",a[i]);
    free(a);
    return 0;
}
```

CUDA

```
int main(void) {
    int size = 4 * sizeof(float);
    float *a, *b = (float*) malloc(size);
    cudaMalloc(&a, size);
    kernel<<<2,2>>>(a);
    cudaMemcpy(b, a, size, cudaMemcpyDeviceToHost);
    for (int i=0; i<4; i++) printf("%f\n",b[i]);
    cudaFree(a);
    free(b);
    return 0;
}
```

# 03\_nbody.cpp

OpenACC

```
#pragma acc kernels loop gang(N/threads) vector(threads)
for (int i=0; i<N; i++) {
    float pi = 0;
    float axi = 0;
    float ayi = 0;
    float azi = 0;
    float xi = x[i];
    float yi = y[i];
    float zi = z[i];
    for (int j=0; j<N; j++) {
        float dx = x[j] - xi;
        float dy = y[j] - yi;
        float dz = z[j] - zi;
        float R2 = dx * dx + dy * dy + dz * dz + EPS2;
        float invR = 1.0f / sqrtf(R2);
        float invR3 = m[j] * invR * invR * invR;
        pi += m[j] * invR;
        axi += dx * invR3;
        ayi += dy * invR3;
        azi += dz * invR3;
    }
    p[i] = pi;
    ax[i] = axi;
    ay[i] = ayi;
    az[i] = azi;
}
```