From Serial to Parallel

A simple training using the Martix-Vector multiplication algorithm

Petros Anastasiadis
National Technical University of Athens

The problem: Dense Matrix-Vector Multiplication

- Appears in multiple simple daily applications
- Also part of many state-of-the-art algorithms in multiple fields (bioinformatics, networks, machine learning etc..)
- An Embarrassing Parallel algorithm

Dense Matrix-Vector Multiplication formula

Matrix-vector product

To define multiplication between a matrix *A* and a vector **x** (i.e., the matrix-vector product), we need to view the vector as a column matrix. We define the matrix-vector product only for the case when the number of columns in *A* equals the number of rows in **x**. So, if *A* is an *m*×*n* matrix (i.e., with *n* columns), then the product *A***x** is defined for *n*×1 column vectors **x**. If we let *A***x**=**b**, then **b** is an *m*×1 column vector. In other words, the number of rows in *A* (which can be anything) determines the number of rows in the product **b**.

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Development environment

- GRNET ARIS HPC (https://hpc.grnet.gr/)
- Utilized Hardware: http://doc.aris.grnet.gr/hardware/
  - CPUs
    - Ivy Bridge - Intel Xeon E5-2680v2
    - Haswell - Intel(R) Xeon(R) E5-2660v3
    - SandyBridge - Intel(R) Xeon(R) CPU E5-4650v2
  - GPUs
    - NVIDIA Tesla K40

Our approach

- CPU parallelization
  - Serial Implementation
  - Naïve OpenMP implementation
  - Affinity/socket sensitive OpenMP implementation
  - MPI multinode implementation
  - Hybrid Multi node/threaded MPI-OpenMP implementation

Our approach

- GPU parallelization
  - Cuda implementation ->
    - Naïve implementation
    - Coalesced memory access
    - Use of GPU shmem
  - cuBLAS library implementation
  - Hybrid MPI-Multi-GPU implementation

Matrix-Vector Multiplication Kernel

- We started from a serial implementation
- The code below performs the y = M*x operation for y[n], M[n*m], x[m]

```
register double yi;
for (k = 0; k < n; ++k) {
        yi = 0.0 ;
        for (j = 0; j < m; ++j)
                yi += M[n*k+j]*x[j];
        y[k] = yi;
}
```

OpenMP implementation

- We can easily parallelize the kernel to up to n different units
- We choose OMP_threads <= Hardware threads in our implementations.
- First Naïve-OpenMP implementation with parallel for:

```
register double yi;
#pragma omp parallel for private(j,yi) shared(n,m,M,y) schedule(dynamic) for (k = 0; k < n; ++k) {
        yi = 0.0 ;
        for (j = 0; j < m; ++j)
                yi += M[n*k+j]*x[j];
        y[k] = yi;
}
```

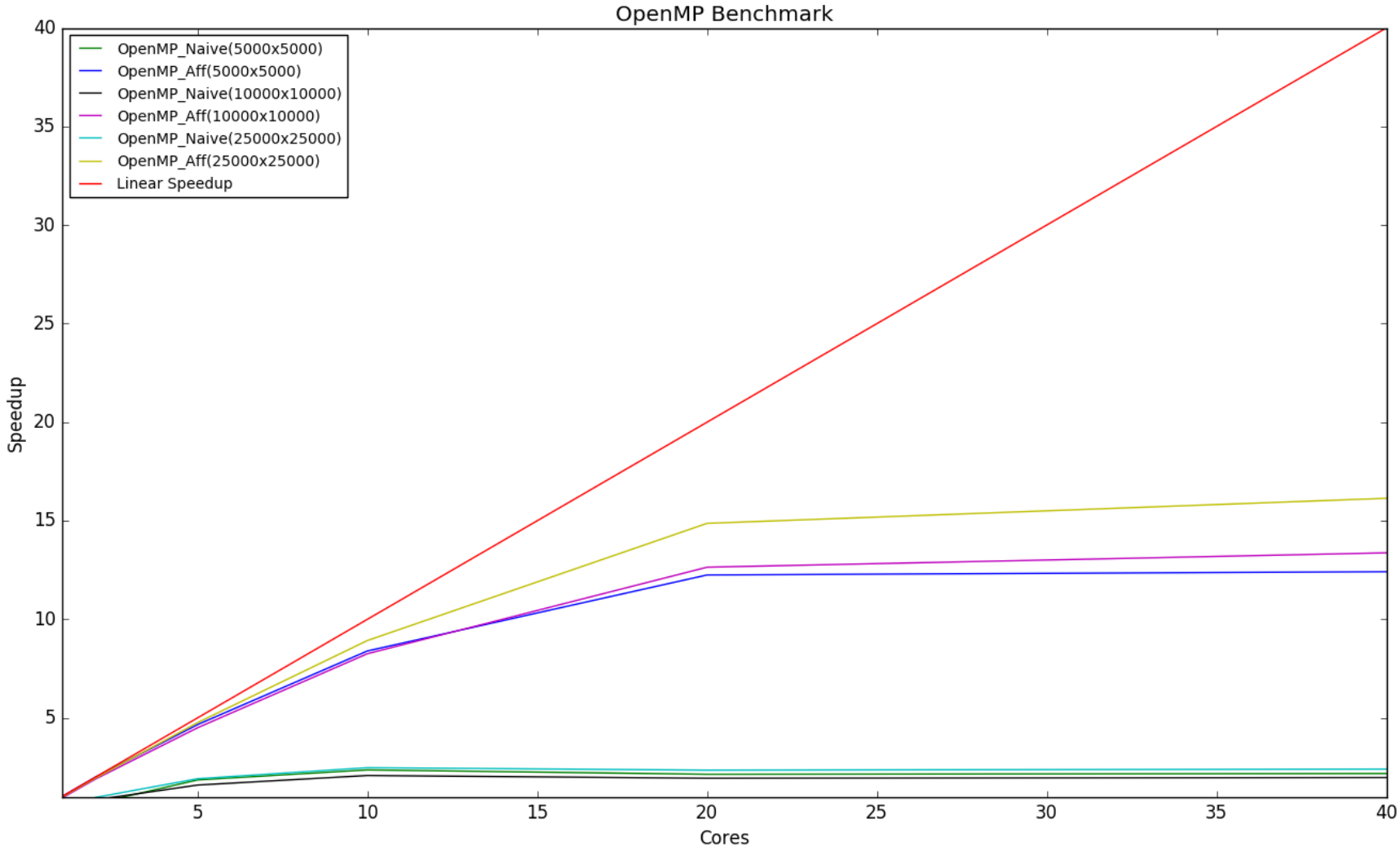
OpenMP implementation

- First problem: Socket transactions and thread movement limit performance.
- This is caused by the relatively **small operational intensity** of the matrix-vector multiplication kernel => performance greatly depends on memory bandwidth and cache utilization
- Flops :
  - m*n additions, m*n multiplications -> 2*m*n Flops
- Bytes:
  - m*n reads for x -> 8*m*n bytes (double precision)
  - m*n reads for M -> 8*n*m bytes (double precision)
  - n writes for y -> 8*n bytes (double precision)
- Operational intensity = Flops/Bytes = m*n / [(8*m+4)*n]

OpenMP implementation

- We want to limit socket transactions and better utilize caches
- We bind each OMP_thread to a physical core
  - export OMP_PROC_BIND=spread
- Each thread initializes its part of the M array ->
  - Memory initialized with first touch will be allocated to current thread's bound core socket
  - Each core's cache now will contain only the elements it needs for its part of the computation

```
#pragma omp parallel for schedule(static)
  for( i=0 ; i<n ; ++i){
      for ( j=0 ; j<m ; ++j) M[i*m+j]=0.0;
  }
```


OpenMP Benchmark
OpenMP_Naive(5000x5000)
OpenMP_Aff(5000x5000)
OpenMP_Naive(10000x10000)
OpenMP_Aff(10000x10000)
OpenMP_Naive(25000x25000)
OpenMP_Aff(25000x25000)
Linear Speedup
Speedup
Cores
40
35
30
25
20
15
10
5
5
10
15
20
25
30
35
40

MPI implementation

- Modern architectures support huge multinode clusters
- Matrix-Vector Multiplication for huge arrays can easily utilize multiple nodes for further parallel computation
- We chose MPI (Message passing interface) for our multinode implementation.
- 2 versions:
  - Multinode MPI
  - Hybrid Multi node/threaded MPI-OpenMP

MPI implementation

- We now have multiple processes instead of a single process who spawns multiple threads
- Non-shared memory model
- Inter-process communication is required -> MPI
- Rank 0 process distributes equal chunks of data to all others
  - MPI_Scatter for M array equal distribution
  - MPI_broadcast for x vector
- Each process computes part of the y vector (Process_num *Serial Kernels)
- Rank 0 gathers the y vector parts
  - MPI_Gather

Hybrid MPI-OpenMP implementation

- Using MPI to spawn a process for each core ignores each node's shared memory
- We can utilize this shared memory to reduce MPI communication
- Thus we use OpenMP for each node and MPI for inter-node communication (1 proc/node with OMP_threads/proc)
- Same with MPI implementation, but now each process computes its part in parallel using OpenMP
- While the achieved speedup is satisfying, MPI communication time is much bigger than computation time.
- We require a more compute intensive kernel in order to bypass this cost, or multiple iterative computations on fewer data.

GPU implementation

- Matrix-Vector Multiplication is a SIMD (single instruction multiple data) algorithm, and thus eligible for GPU parallelization.
- Its huge memory bandwidth requirements fit well with the high-bandwidth GPU memories.
- Its operational simplicity makes it rather easy to implement as a GPU kernel.
- In our approach, we start with a naïve GPU version, and improve it step by step to better fit the GPU logic.

Naïve Cuda Implementation

- In our first version, we simply convert our multiplication loop to device code.
- Each warp executes the same code in different data:

```
int tid = get_global_tid();
double yi = 0.0;
if(tid >= n)
        return ;
for ( int j = 0 ; j < n; j++ )
        yi += + a[tid*n+j]*x[j];
y[tid]=yi;
```

Coalesced Cuda Implementation

- The naïve version performs very bad in big arrays where memory bandwidth is critical, because the memory transactions are slow.
- For this reason we change the array format (by transposing it) and make the kernel column major.

```
int tid = get_global_tid();
double yi = 0.0;
if(tid >= n)
        return ;
for ( int j = 0 ; j < n; j++ )
        yi += + a[n*j+ tid]*x[j];
y[tid]=yi;
```

- Now, the threads in each warp require contiguous elements of a, and thus the memory transactions are coalesced, resulting in huge bandwidth improvement.

Shmem Cuda Implementation

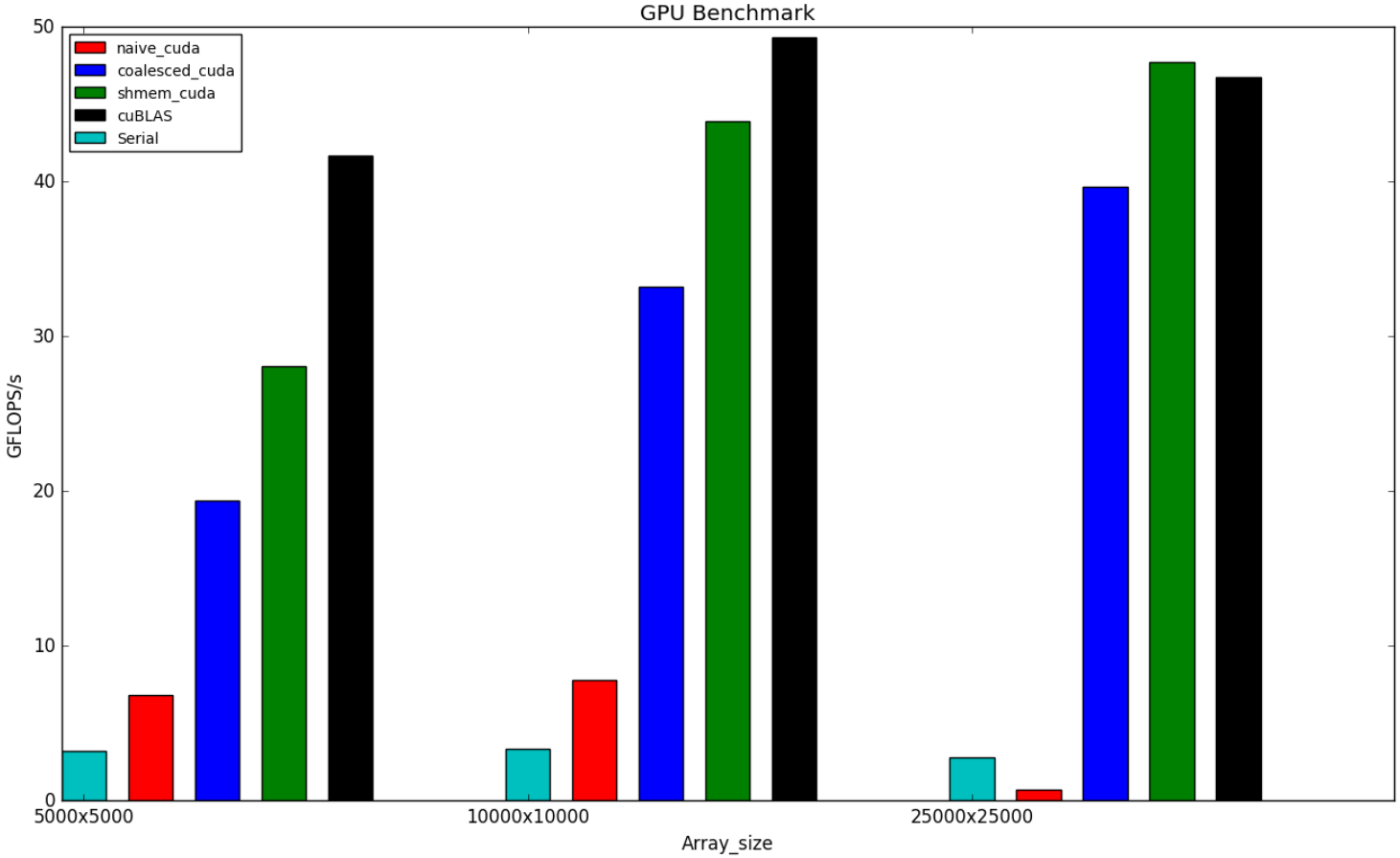
- To further improve our coalesced version, we load the x vector before the computation part into the GPU Shmem (block shared memory)
- Now the x loading is also coalesced, and no memory bandwidth is expanded during the computation part in order to fetch the (now locally available) x vector.
- Since the x vector is probably bigger than the block shmem, we split the above in parts loading the x vector part we need each time.

Shmem Cuda Implementation

```
extern __shared__ float shmem_buff[] ;
int tid = get_global_tid(), i, j;
double yi = 0.0;
if(tid >= n)
                return ;
int block_s=blockDim.x*blockDim.y;
int lid=get_local_tid(), last_id = n/block_s ;
for( j = 0; j< last_id; j++) {
                shmem_buff[lid] = x[block_s*j + lid];
                __syncthreads();
                for( i = 0 ; i < block_s; i++ ) {
                                yi += a[tid+ (i+j*block_s)*n]*shmem_buff[i];
                }
                __syncthreads();
}
y[tid]=yi;
```

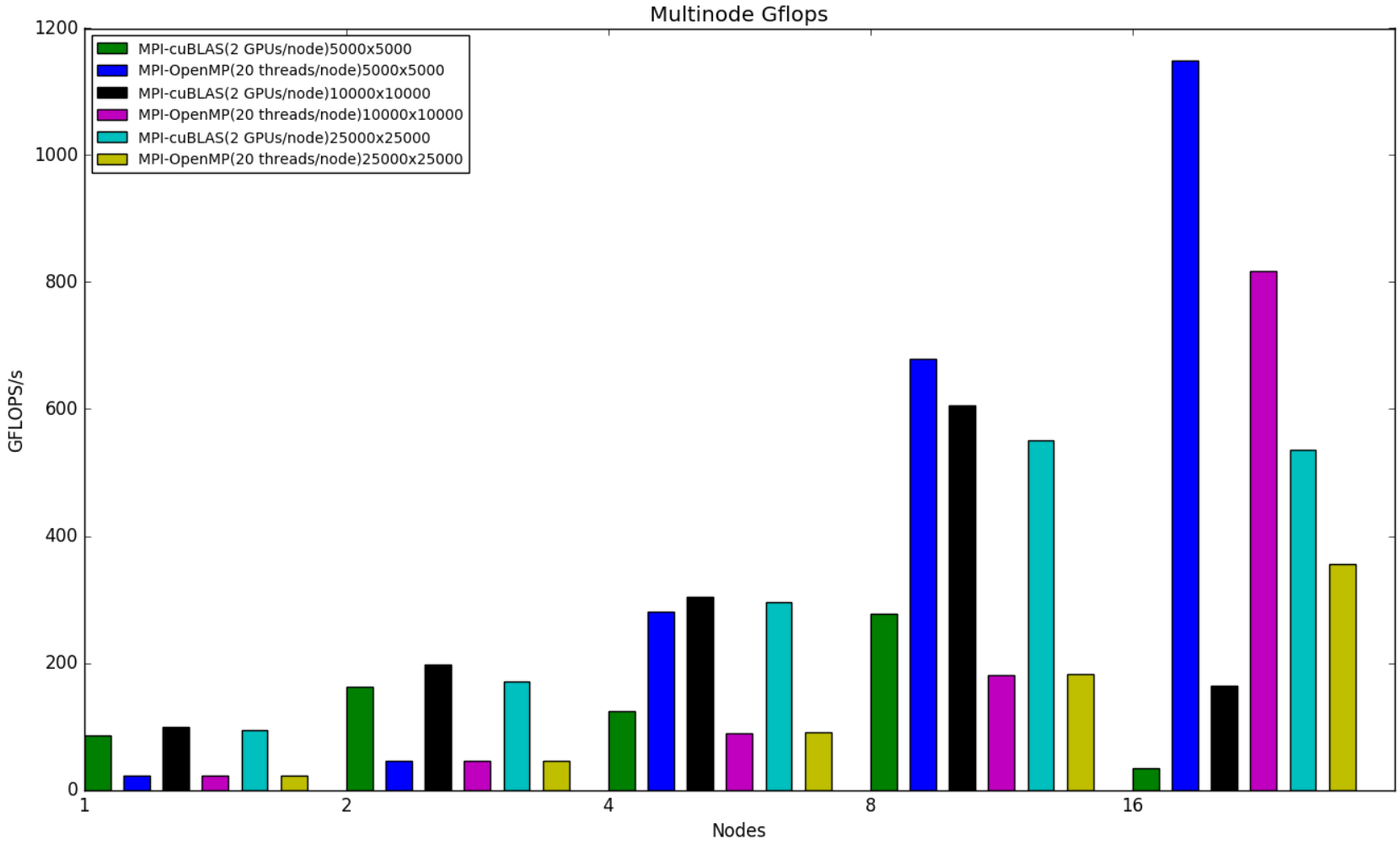
cuBLAS Implementation

- cuBLAS is the optimized blas library implementation for Nvidia GPUs
- It is internally designed to run optimally for almost every type of array
- We also created a basic cuBLAS implementation in order to rate our implementations
- The results are shown in the graph below:


GPU Benchmark
naive_cuda
coalesced_cuda
shmem_cuda
cuBLAS
Serial
GFLOPS/s
0
10
20
30
40
50
5000x5000
10000x10000
25000x25000
Array_size

MPI-cuBLAS Hybrid

- To conclude with our approach, we implement a hybrid GPU-Multinode implementation, using cuBLAS for the computation part and MPI in order to split the work in multiple GPUs.
- We can now compare our 2 best multinode implementations
- We test their performance in 3 different array sizes


Multinode Gflops
MPI-cuBLAS(2 GPUs/node)5000x5000
MPI-OpenMP(20 threads/node)5000x5000
MPI-cuBLAS(2 GPUs/node)10000x10000
MPI-OpenMP(20 threads/node)10000x10000
MPI-cuBLAS(2 GPUs/node)25000x25000
MPI-OpenMP(20 threads/node)25000x25000
GFLOPS/s
0
200
400
600
800
1000
1200
1
2
4
8
16
Nodes

Conclusion

- Each implementation has its own pros and cons:
  - OpenMP is fast no matter the data size if it is correctly optimized, but it is limited to shared memory architectures. Its simplicity makes it ideal for new programmers.
  - MPI supports inter-communication between processes , so it can utilize multinode architectures, but the communication cost is heavy for memory-bound kernels. Its use requires a bit more programming experience.
  - GPUs offer a good memory bandwidth and an ideal environment for SIMD kernels, but work well with big chunks of data and perform poorly for small ones. Cuda programming is even more complex for a beginner, but cuBLAS offers a huge variety of functions that can be called from the CPU.

Conclusion

- Each implementation has its own pros and cons:
  - OpenMP is fast no matter the data size if it is correctly optimized, but it is limited to shared memory architectures. Its simplicity makes it ideal for new programmers.
  - MPI supports inter-communication between processes , so it can utilize multinode architectures, but the communication cost is heavy for memory-bound kernels. Its use requires a bit more programming experience.
  - GPUs offer a good memory bandwidth and an ideal environment for SIMD kernels, but work well with big chunks of data and perform poorly for small ones. Cuda programming is even more complex for a beginner, but cuBLAS offers a huge variety of functions that can be called from the CPU.

THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu