TASKING IN ACCELERATORS

Cuda Graphs and OpenACC programming models

Leonel Toledo, Antonio J. Peña.

The EPEEC project has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement Nº 801051

INTRODUCTION

Introduction

- GPU compute capabilities have been significantly increasing; however, applications and scalability of algorithms still face some important challenges. One important problem regarding scalability is hardware resource assignment.

- Tasking, on the other hand makes it possible for algorithms with run-time dependent execution flow, to be parallelized.

Introduction

- Dynamic Parallelism in CUDA
  - Launch kernels from threads running on the device.
    - Significant Overhead

- CUDA Streams for asynchronous calls
  - Queueing system.
    - Sometimes is difficult to handle a large number of streams by both the programmer and the hardware.
- A combination of both CDP + CUDA streams.
- CUDA Graph API.

Introduction

- This work explores different approaches to run several tasks with data dependencies on the same GPU envisioning a future integration of tasking oriented programming models and NVIDIA GPUs.

- Different performance tests
  - Dynamic Parallelism
  - CUDA streams
    - From both host and device
  - Cuda Graph API.

Introduction - Motivation

- Matrix operations are relevant within scientific and engineering computing operations.

- In this work, we present the results of evaluating DGEMM operations on the GPU using dynamic parallelism and CUDA graph in different configurations using the following heterogeneous system: 2x IBM power9 8335-GTH at 2.4 Ghz, 32 GB RAM memory, and a NVIDIA V100 (Volta) GPU with 16GB HBM2 and NVLink2 for high-bandwidth communication between CPU and GPU.

Introduction - Contribution

- The contribution of this work is the analysis and evaluation of the current CUDA features for tasking in latest GPU architecture, this is a preliminary study and analysis for a future integration of NVIDIA GPUs and tasking-oriented programming models.

RELATED WORK

EPEEC
PRODUCTIVITY AT EXASCALE

Related Work

- Task based programming models.
  - StarPU
  - Thread Building Blocks
  - High Performance ParallelX
  - OpenMP
  - OmpSs
- CUDA Dynamic Parallelism and CUDA Graph.
- Linear Algebra Libraries
  - CuBlAS
  - CuSparse
  - LAS

IMPLEMENTATION

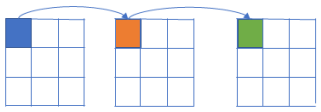
Implementation

- For this study we propose four different approaches to benchmark dynamic parallelism and task oriented programming on the GPU.
- The first approach uses only dynamic parallelism, and tasks are generated sequentially. Second, we introduce streams within the GPU, that way we take advantage of the several execution queues that are available in the device. Next, we compare the same approach, but instead of instantiating jobs within the GPU, we create tasks from the host using multiple streams at the same time. Finally, we test task performance using the CUDA graph API.

Experiment Design

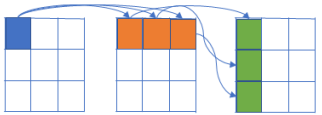
- We define three different scenarios with different degrees of parallelism, synchronization and tasks executed every time.

```
for (int i =0; i<TILES; i++){
        for (int j=0; j< TILES; j++){
                for (int k=0; k< TILES; k++){
                Dgemm(…)
                synch();
                }
        }
}
```



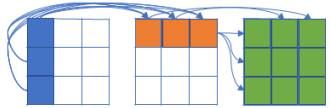
A[i][k] * B[k][j] = C[i][j]

```
for (int j =0; j<TILES; j++){
        for (int k=0; k< TILES; k++){
                for (int i=0; i< TILES; i++){
                Dgemm(…)
                }
        synch();
        }
}
```



A[i][k] * B[k][j] = C[i][j]

```
for (int k =0; k<TILES; k++){
        for (int i=0; i< TILES; i++){
                for (int j=0; j< TILES; j++){
                Dgemm(…)
                }
        }
        synch();
}
```



A[i][k] * B[k][j] = C[i][j]

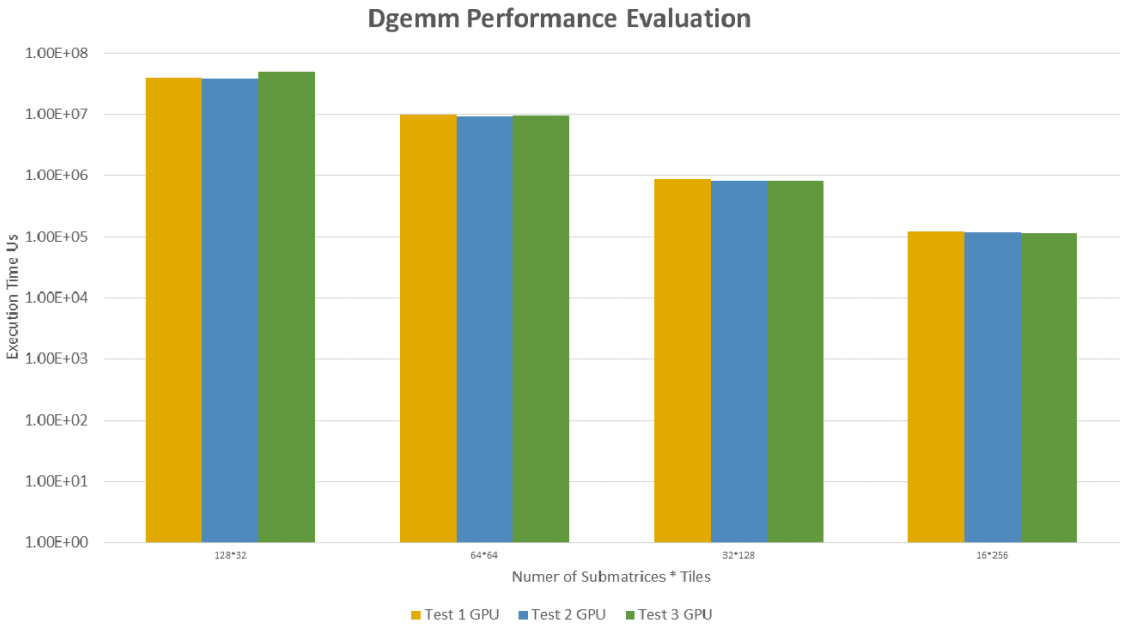
Dynamic Parallelism

- The main advantage of this feature is the capacity of the GPU to constantly create work without returning to the host.
- As expected, kernels that made the most parallel calculations had a better performance overall, the best cases had performance gains about 30%.
- However, there are important considerations that must be addressed.
  - Even though performance improves from test to test, it is not as significant as we expected and this is mainly due because two main reasons.

Dynamic Parallelism

- First, even though it is not necessary to return to the host to generate more work, most of the memory fetching is from global memory.
  - The bottleneck of the execution is how the hardware handles the creation and destruction of kernels in between launches from the device.

- Second, there is a significant overhead that is involved with creating kernels within the GPU which directly impacts the performance. To mitigate that, we added CUDA streams to maximize the amount of asynchronous work.


Dgemm Performance Evaluation
1.00E+08
1.00E+07
1.00E+06
1.00E+05
1.00E+04
1.00E+03
1.00E+02
1.00E+01
1.00E+00
Execution Time Us
128*32
64*64
32*128
16*256
Numer of Submatrices * Tiles
Test 1 GPU
Test 2 GPU
Test 3 GPU

CUDA Streams with Dynamic Parallelism

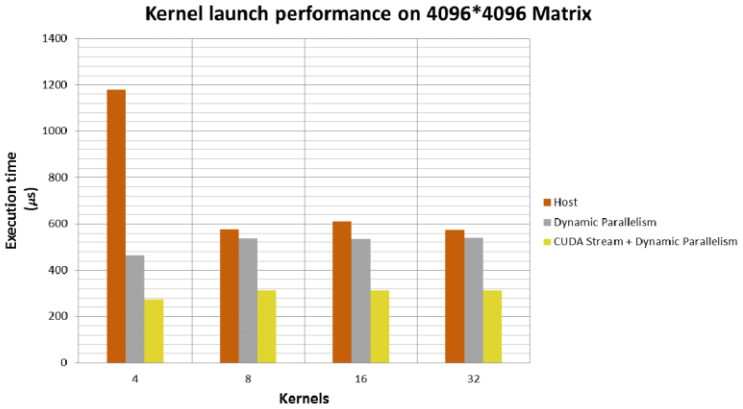
- By including streams into the evaluation, it was observed that there was a significant improvement in terms of performance.

- The experiment took less time to complete in all the configurations that were tested against using only dynamic parallelism without explicit concurrency.

  - However, speaking of how the GPU handles several tasks at the same time, this approach shows limitations.

CUDA Streams with Dynamic Parallelism

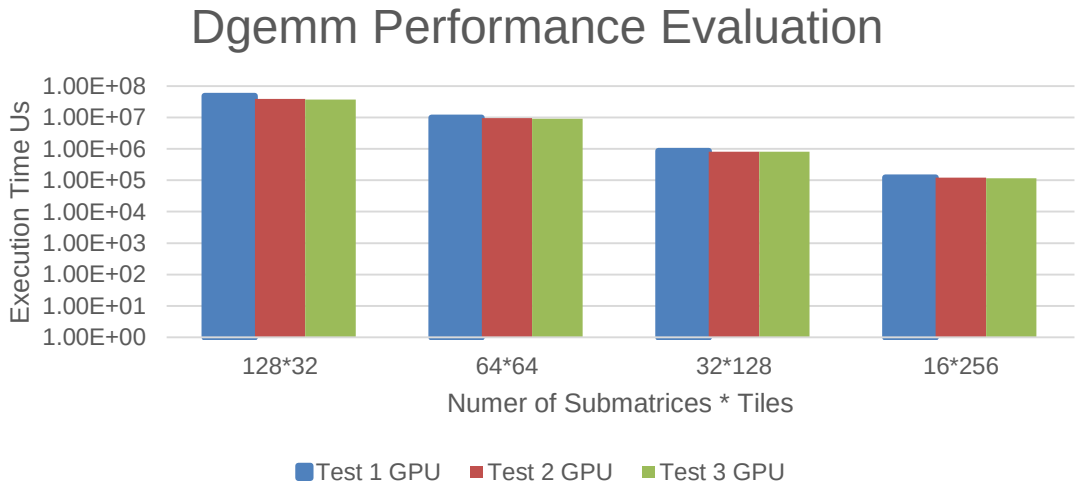
- Regardless of the configuration, the GPU did not execute efficiently the different tasks that needed to execute, even when the third configuration design had much more parallelism and a higher potential for overlapped calculations, it not outperformed significantly the first configuration that was almost sequential



*Tests were performed using 4 streams.

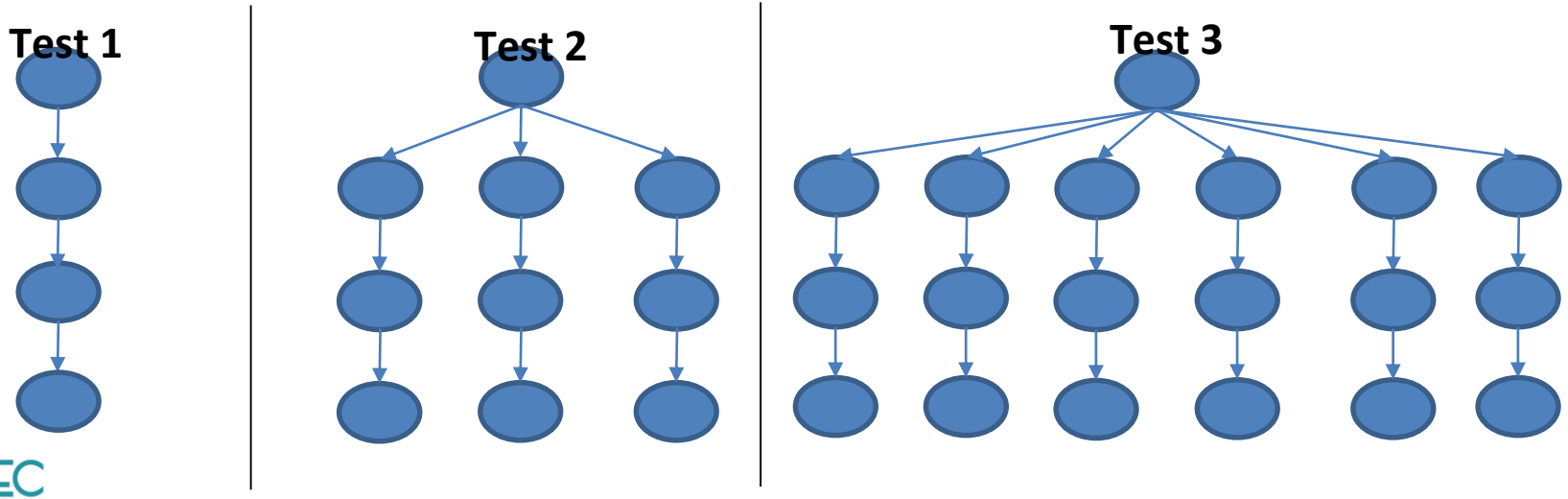
CUDA Streams from Host

- Based on the previous results, it is important to address that using dynamic parallelism, whether it is alone or with CUDA streams, has an important bottleneck that is directly related to the creation of kernels within the GPU.



Cuda Graph

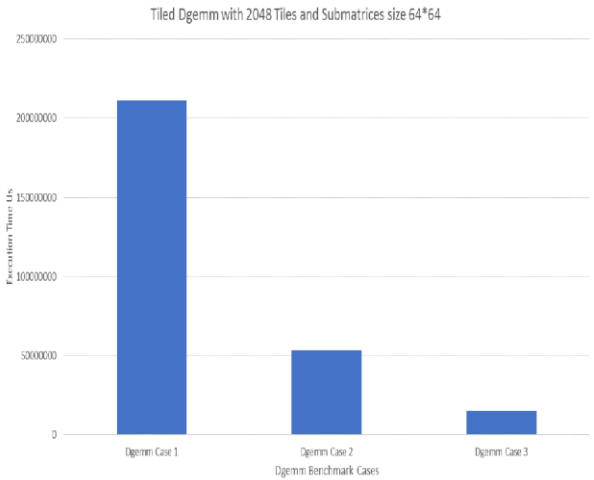
- Many HPC applications such as deep neural network training and scientific simulations have an iterative structure where the same workflow is executed repeatedly. CUDA streams require that the work be resubmitted with every iteration, which consumes both time and CPU resources. Graphs enable a *define-once-run-repeatedly* execution flow.



EPEEC
PRODUCTIVITY AT EXASCALE

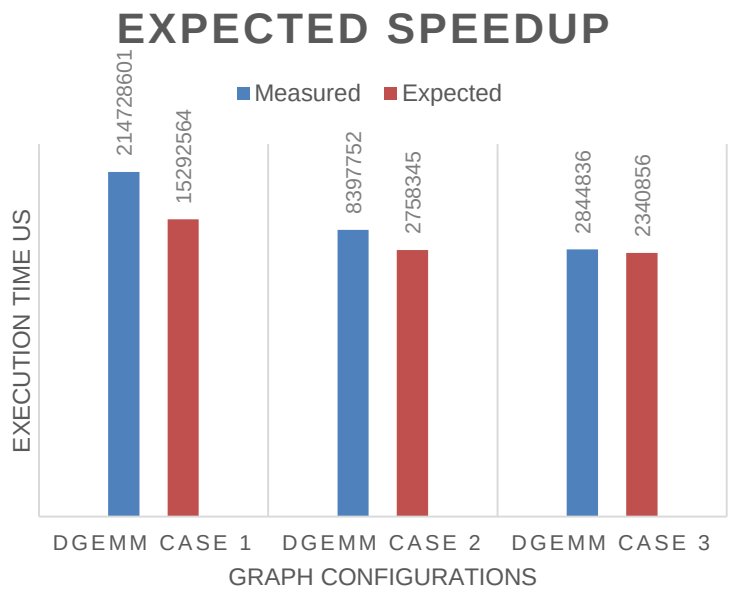
Tiled Dgemm using CUDA Graph

- We were able to achieve a better degree of parallelism and task overlapping.
- For instance, between benchmark case 1 - 2 we achieved on average a 4x acceleration, and between benchmark case 1 – 3 a 12x acceleration.
- On the best cases we were able to achieve a 25x acceleration between cases 1 - 2 and 80x acceleration between cases 1 - 3.



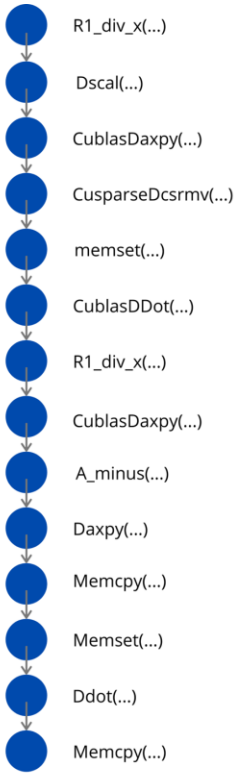
Expected Speedup

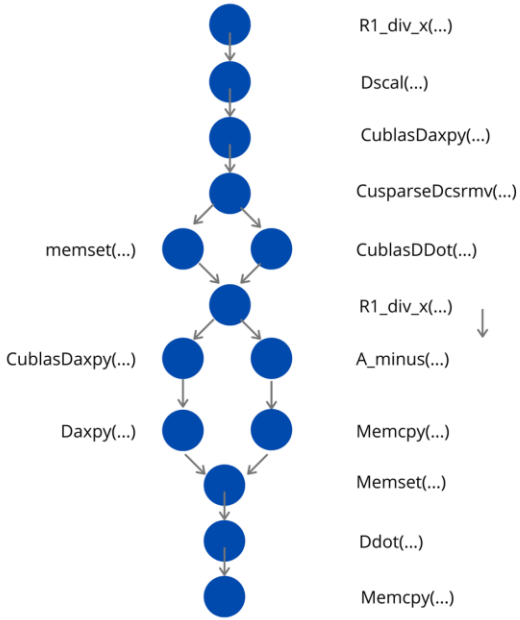
- The results of the theoretical maximum is represented in Red
- In Blue is the performance of a determined kernel using 32 tiles

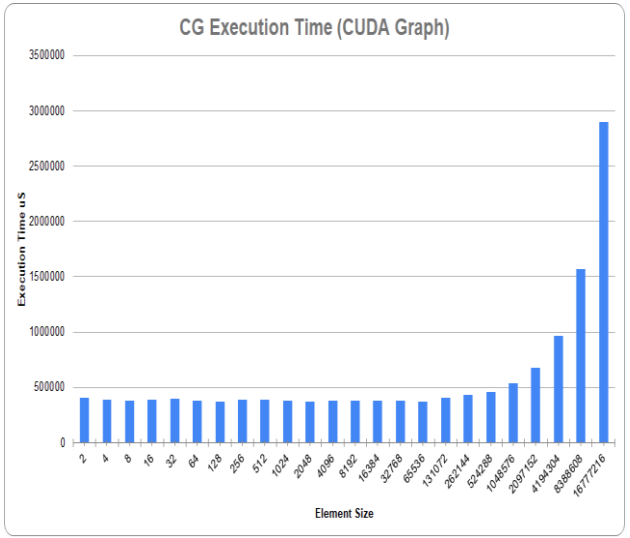


Accelerating CG Using CUDA Graph

Basic Algorithm







Programming Model Integration

CUDA GRAPH + OPENACC

OpenACC Kernel Definition

```
#include <openacc.h>
#include <cuda_runtime.h>

//OPENACC CODE
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
#pragma acc kernels deviceptr(x,y) async(0)
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

Main

```
int main (int argc, char *argv[]){

cudaStream_t stream1, streamForGraph;
cudaGraph_t graph;

cudaStreamCreate(&stream1);
cudaStreamCreate(&streamForGraph);

void* stream = acc_get_cuda_stream(acc_async_sync);

acc_set_cuda_stream(0,stream1);

cudaStreamBeginCapture(stream1 , cudaStreamCaptureModeGlobal);

saxpy(SIZE,2.0,x,y);
saxpy(SIZE,2.0,x,y);
saxpy(SIZE,2.0,x,y);
saxpy(SIZE,2.0,x,y);

cudaStreamEndCapture(stream1 , &graph);
```

Launch

```
cudaGraphExec_t graphExec;
        cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);

    for (int i = 0; i < ITERATIONS; i++) {
            cudaGraphLaunch(graphExec, streamForGraph);
    }

    cudaStreamSynchronize(streamForGraph);
```

EPEEC PRODUCTIVITY AT EXASCALE

Application (Particle Swarm Optimization)

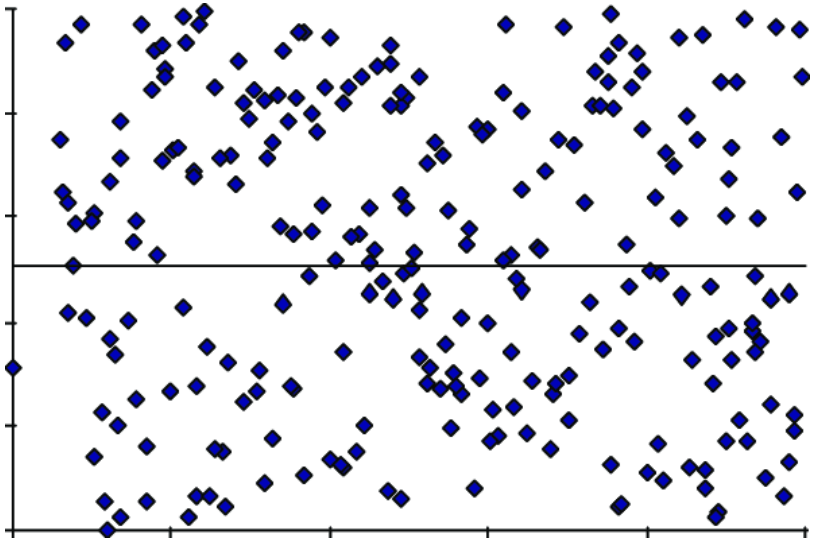
PSO Definition

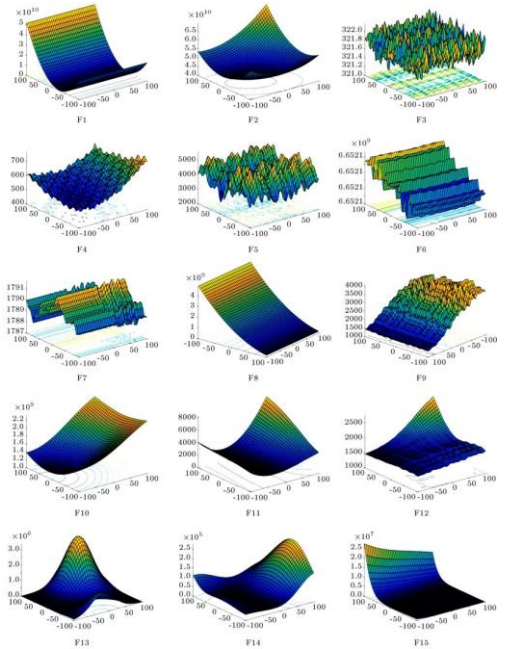
- For each particle we define a set of variables where we evaluate any given function.

- The result of the evaluation determines the fitness of the particle.

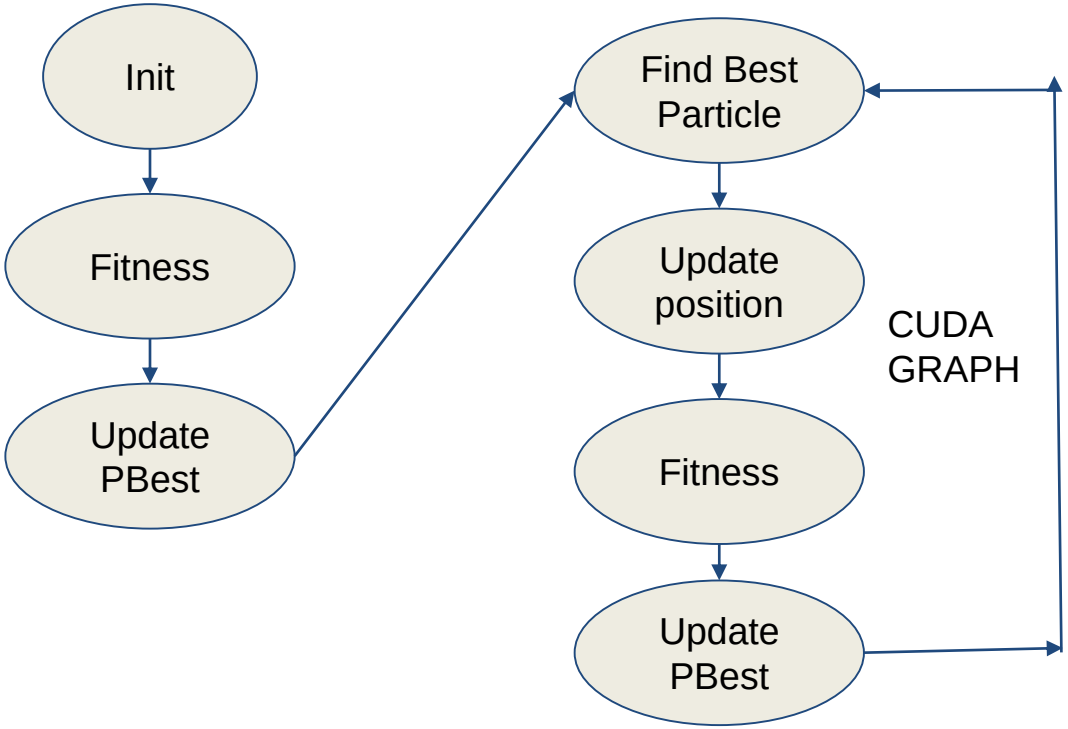
```
typedef struct particle{

    double posX;
    double posY;
    double posZ;

    double speedX;
    double speedY;
    double speedZ;

    double bestX;
    double bestY;
    double bestZ;

    double fitness;
    double bestValue;
    double currentValue;

} Particle;
```

Search Space





Workflow


Init
Fitness
Update PBest
Find Best Particle
Update position
Fitness
Update PBest
CUDA GRAPH

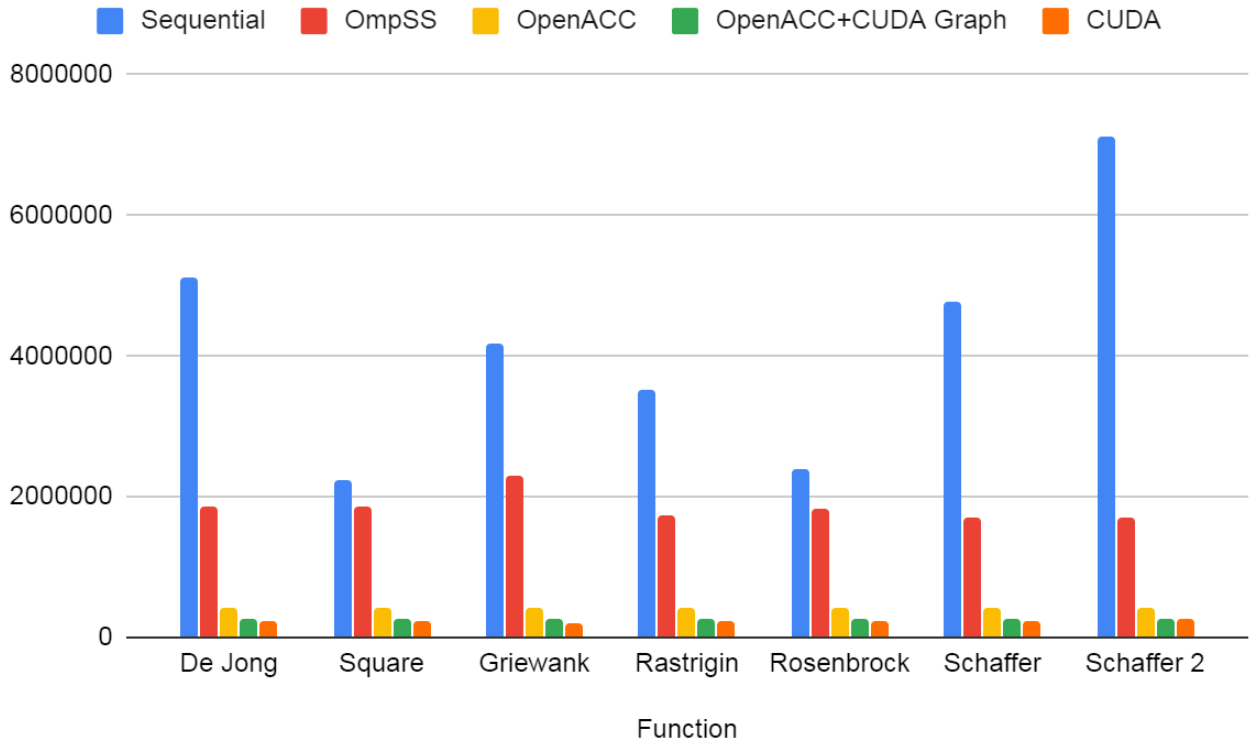
Algorithm implementation

```
initTask();
fitnessTask();
updatePBestTask();
for (int i=0; i<ITERATIONS; i++){
    findBestParticleTask();
    updateSpeedVectorTask();
    fitnessTask();
    updatePBestTask();
}
```

```
void fitnessTask(){
    for(int i=0; i<POPULATION; i++){
        calculateFitness(particleVector[i],OBJECTIVE);
    }
}

void updatePBestTask(){
    for(int i=0; i<POPULATION; i++){
        updatePBest(particleVector[i],OBJECTIVE);
    }
}

void findBestParticleTask(){
    findBestParticle(particleVector,OBJECTIVE,GlobalBestFitness
}
```

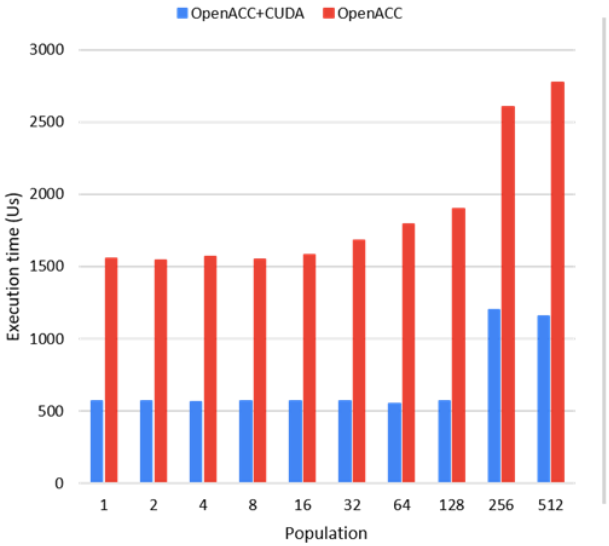
EPEEC
PRODUCTIVITY AT EXASCALE

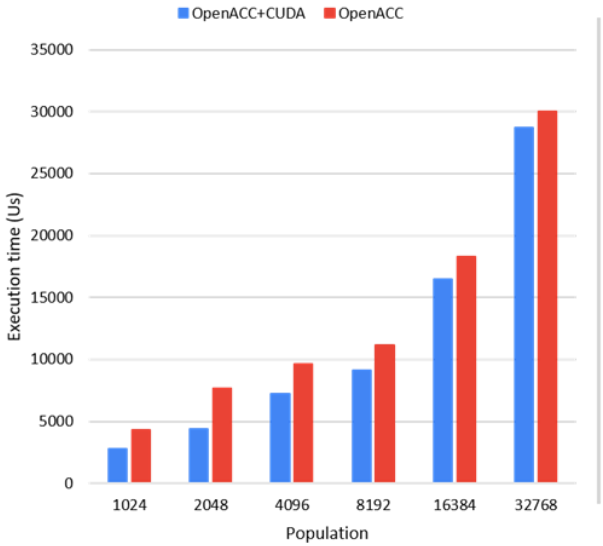
Results

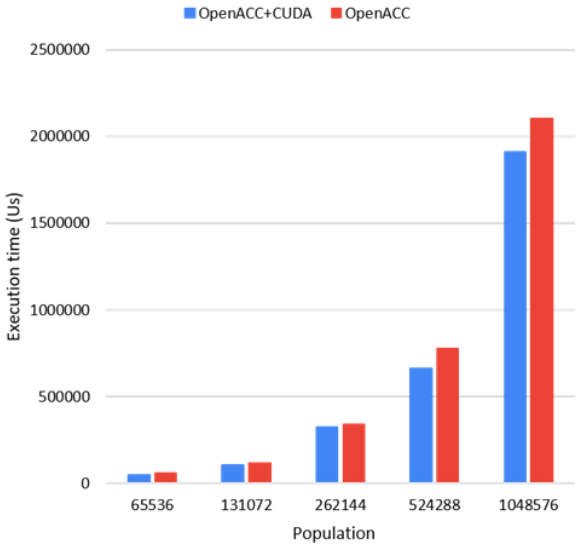


EPEEC
PRODUCTIVITY AT EXASCALE

Population Performance

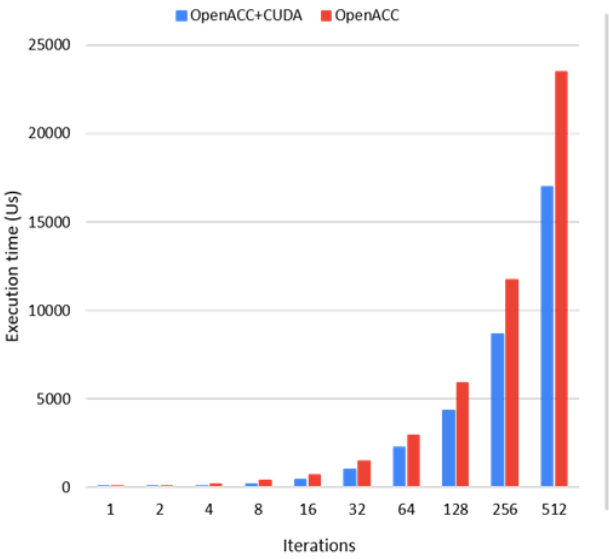


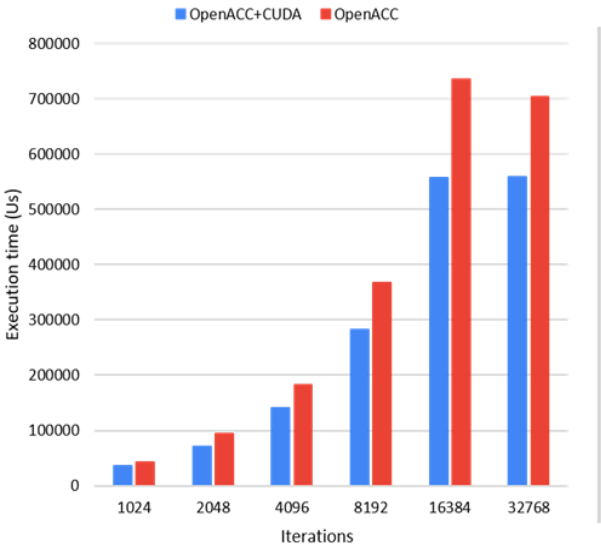


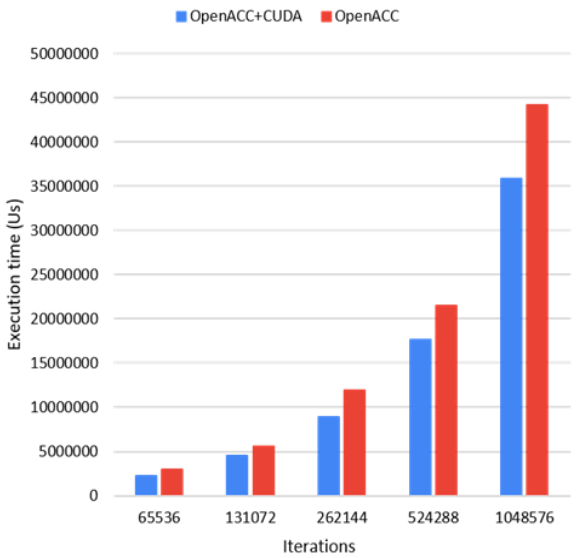


EPEEC
PRODUCTIVITY AT EXASCALE

Iteration Performance







EPEEC
PRODUCTIVITY AT EXASCALE

Thank You!

EPEEC
PRODUCTIVITY AT EXASCALE

www.epeec-project.eu

The EPEEC project has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement № 801051