SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems

Johan Enmyren Christoph W. Kessler

{x10johen, chrke}@ida.liu.se

Linköping University, Sweden

Outline

- Background
- SkePU
- Evaluation
- Conclusions
- Future Work

Background

- Trend towards multi- and many-core systems
- GPU Computing
- Different parallel programming models for different architectures → portability problem
- Skeleton programming
- SkePU, skeletons for GPU/CPU

SkePU

- C++ template library
- Multiple back ends
- Container
- User functions
- Skeletons
- Multi-GPU
- Various helpers

Container

- Vector-type (modeled after std::vector)
- Handles memory transfers between host and device
- Lazy memory copying

```
skepu::Vector<double> v0(1000,2);
```

User Functions

- Macro-language
- Behind the scenes, a C++ struct

```
BINARY_FUNC(plus, double, a, b,
    return a+b;
)
```

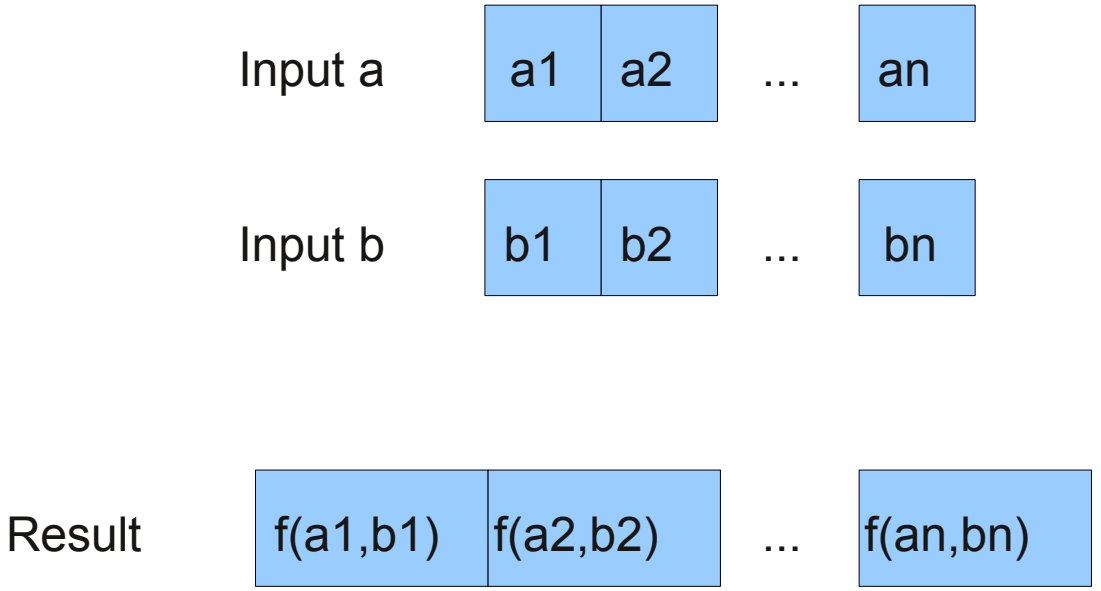
Skeletons

- Objects, overloading operator()

- Map
- Reduce
- MapReduce
- MapOverlap
- MapArray
- (Scan)

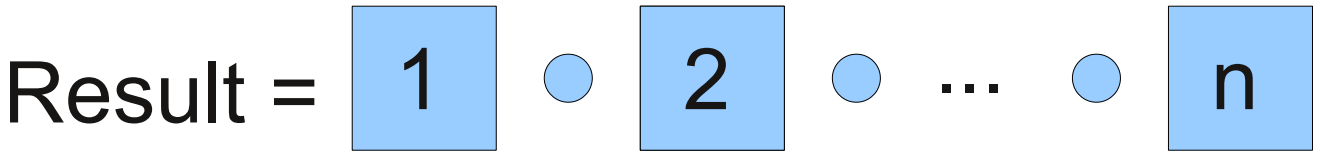
Map

- Each element in the result vector is a function of the corresponding element in one or more input vectors.



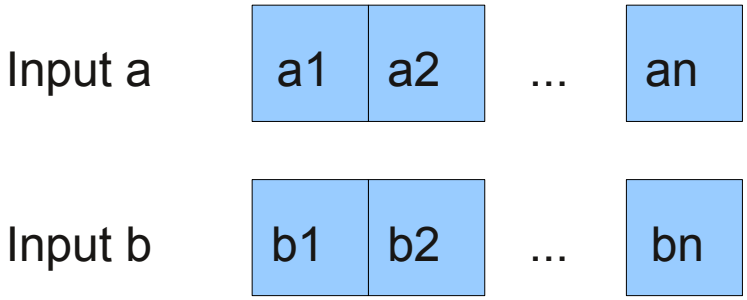
Reduce

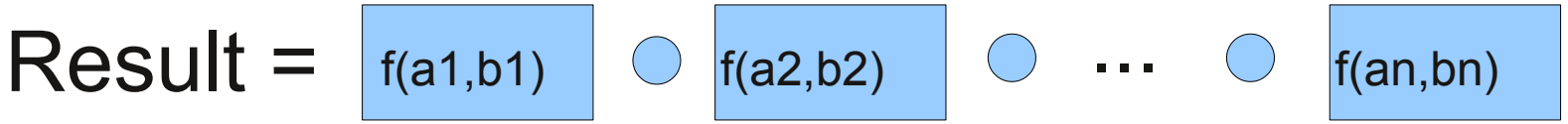
- A scalar result is computed by applying a commutative associative binary operator between each element in the vector.



MapReduce

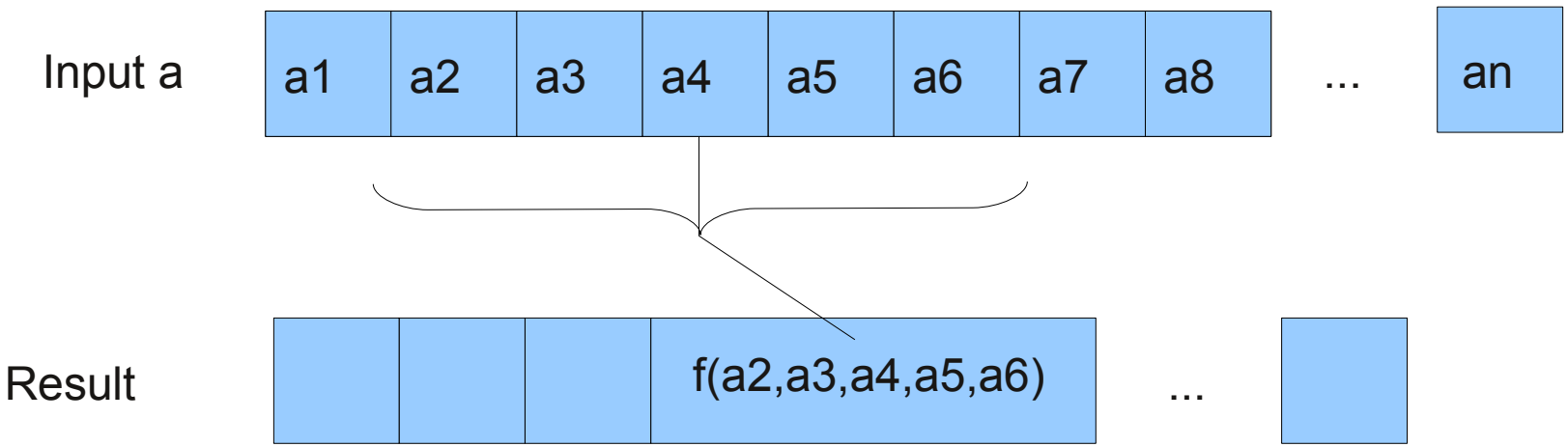
- Combination of Map and Reduce. Produces the same result as if a Map was first performed then a Reduction of the result.





MapOverlap

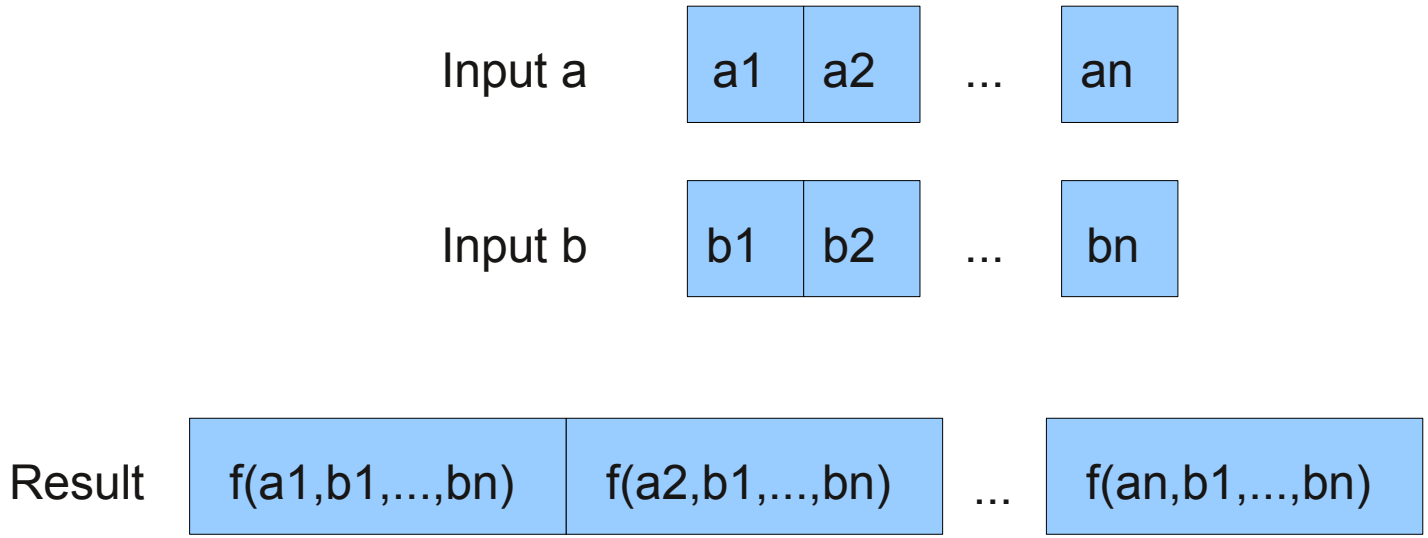
- Similar to a Map, but each element of the result vector is a function of several adjacent elements of one input vector.



Overlap: 2

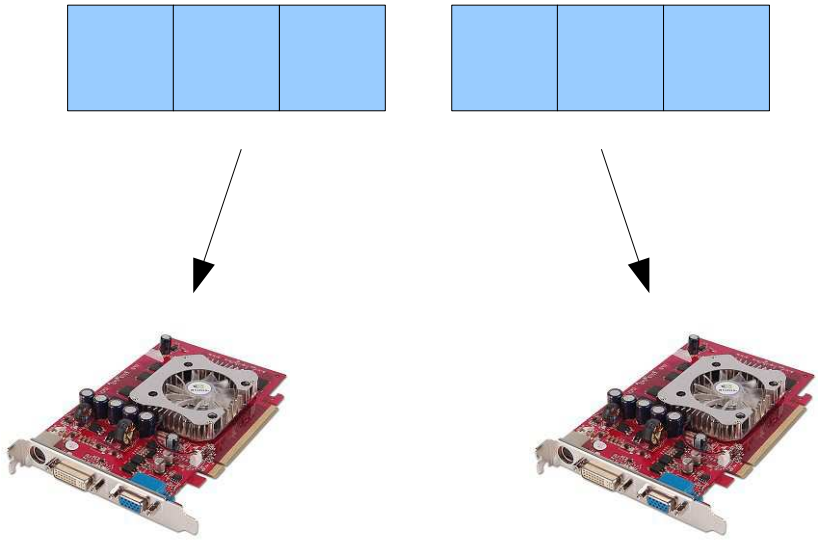
MapArray

- Variant of Map. Each element of the result is a function of the corresponding element of one of the input vectors and any number of elements from the other input vector.



Multi-GPU

- Divides the vector evenly among the GPUs
- Transparent to the user



Helpers

- Device management
- Memory management
- Testing

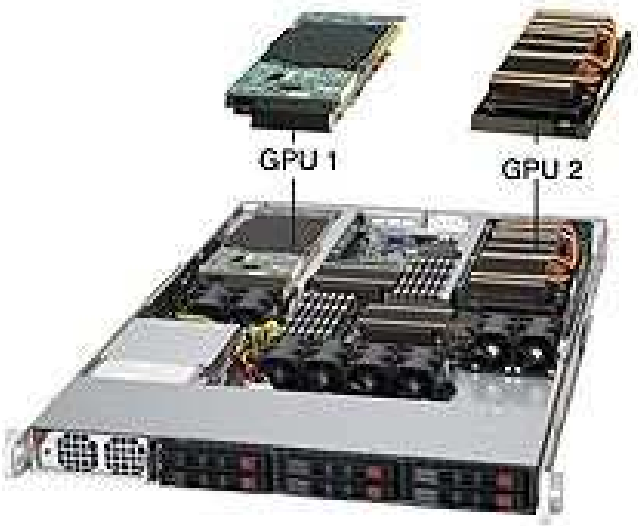
Example: Dot Product

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus, double, a, b,
    return a+b;
)

BINARY_FUNC(mult, double, a, b,
    return a*b;
)
```

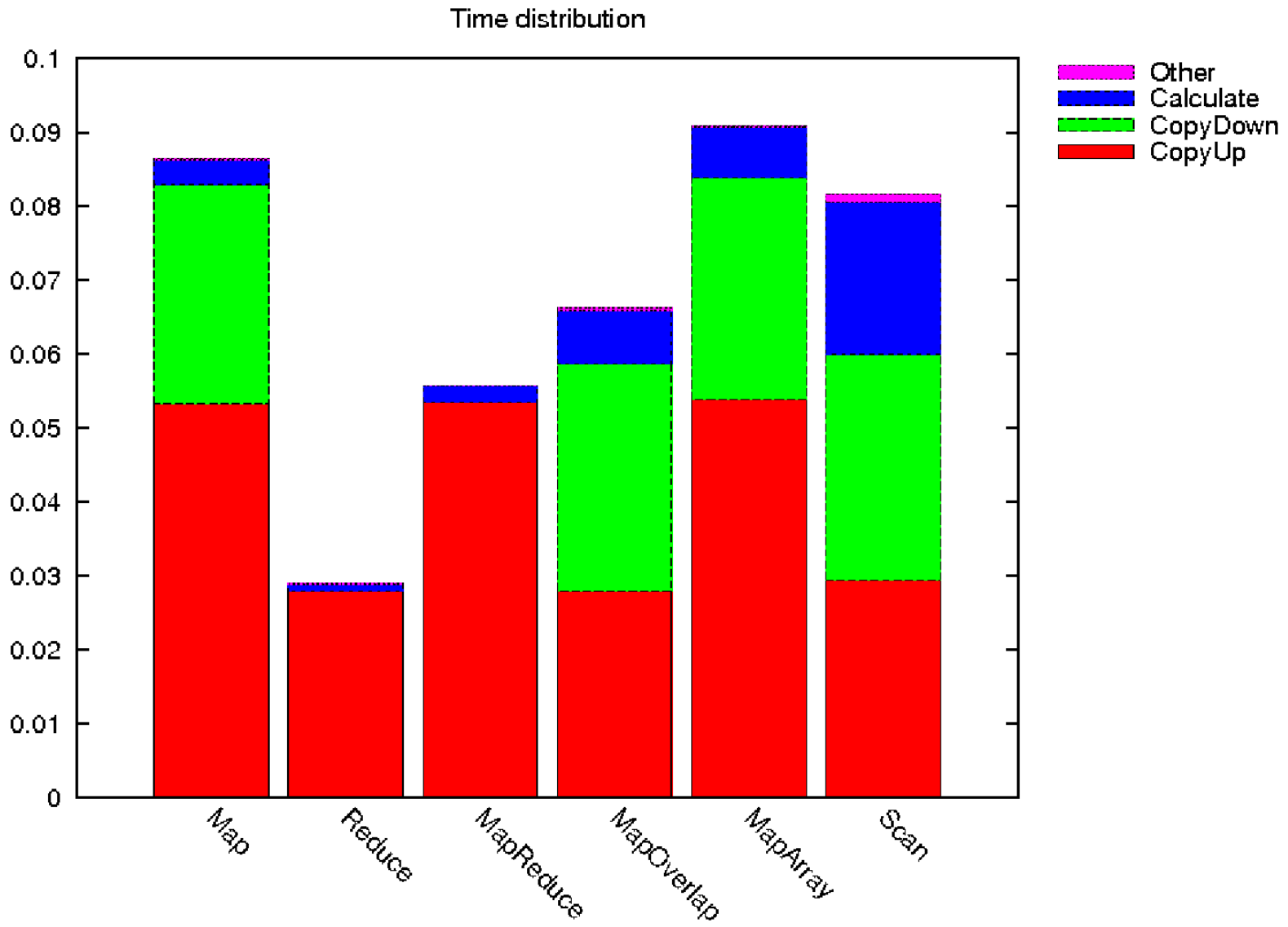
```
int main()
{
    skepu::MapReduce<mult, plus>
    dotProduct(new mult, new plus);

    skepu::Vector<double> v0(1000,2);
    skepu::Vector<double> v1(1000,2);

    double r = dotProduct(v0,v1);

    std::cout<<"Result: " <<r <<"\n";

    return 0;
}
```

Evaluation

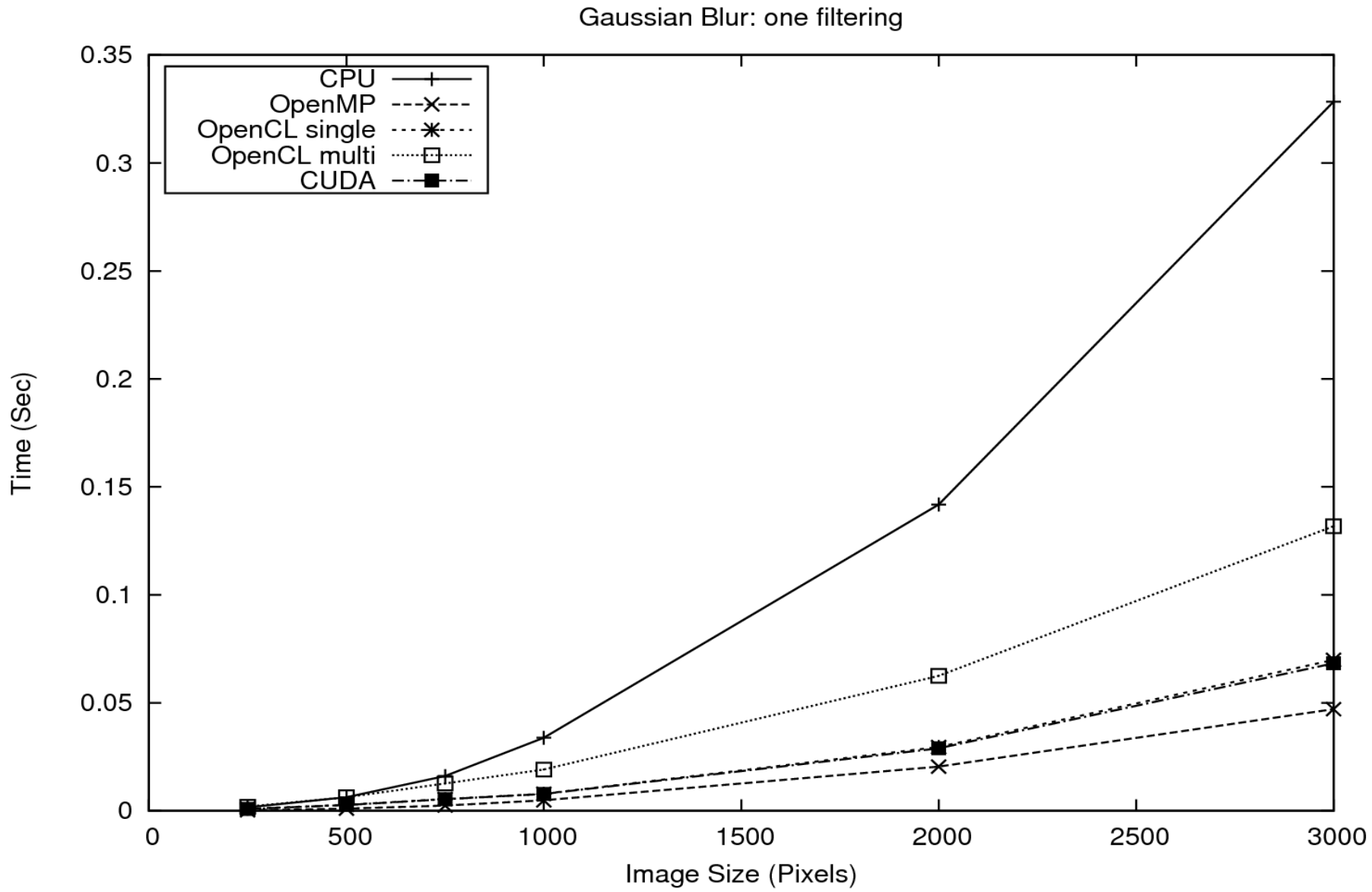
- Time Distribution
- Gaussian Blur
- Dot Product
- Runge-Kutta ODE Solver



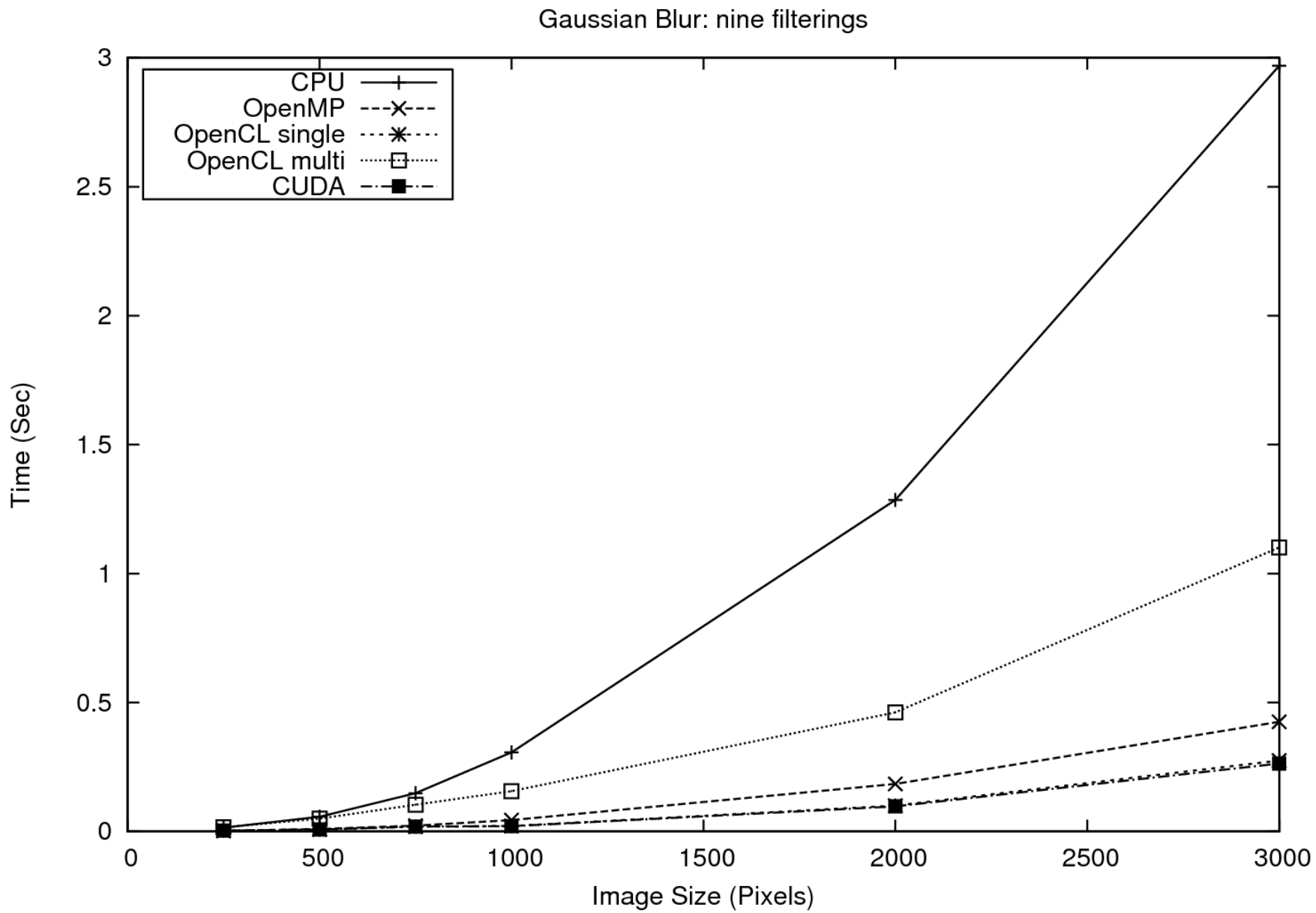
Time Distribution



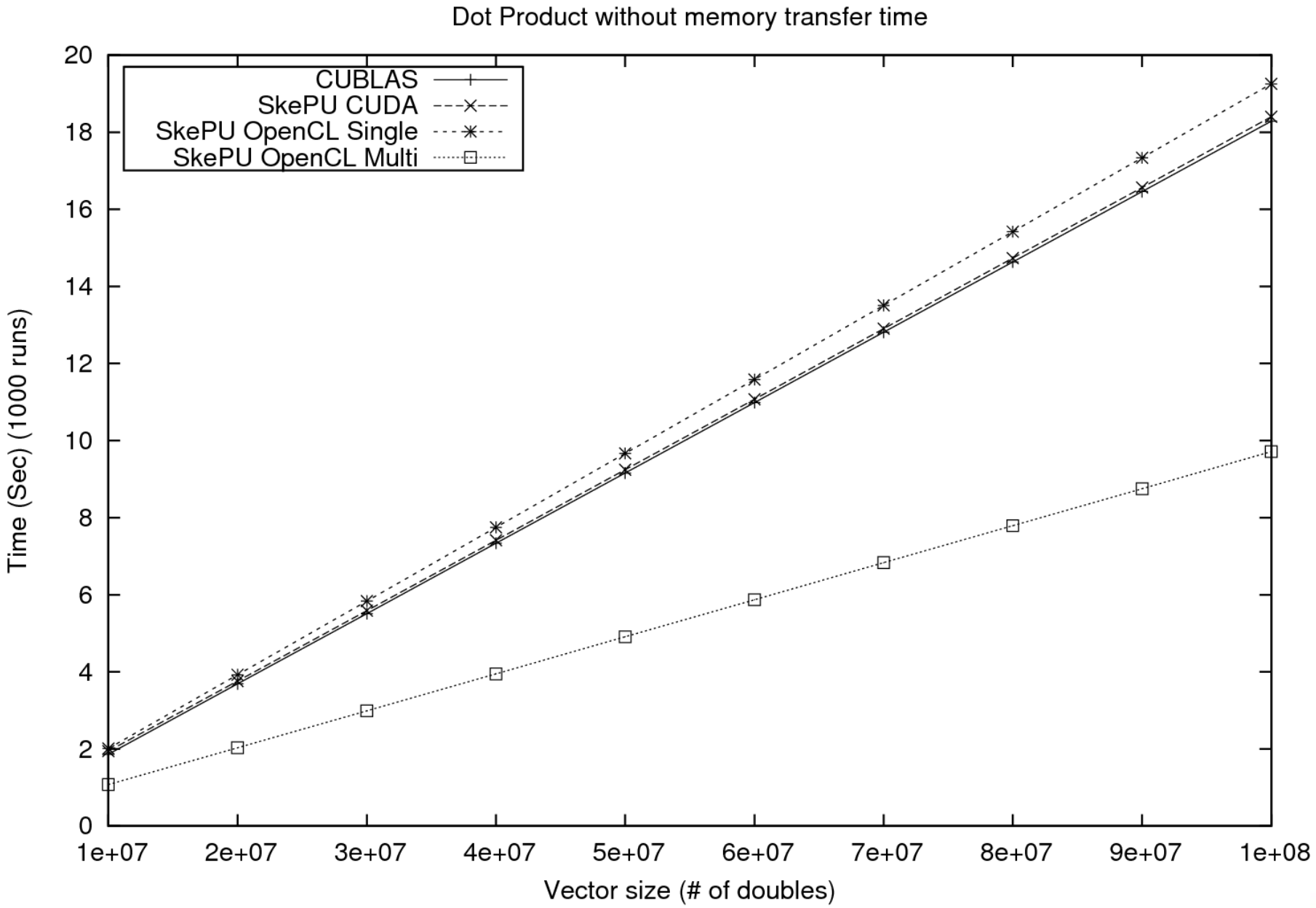
Gaussian Blur



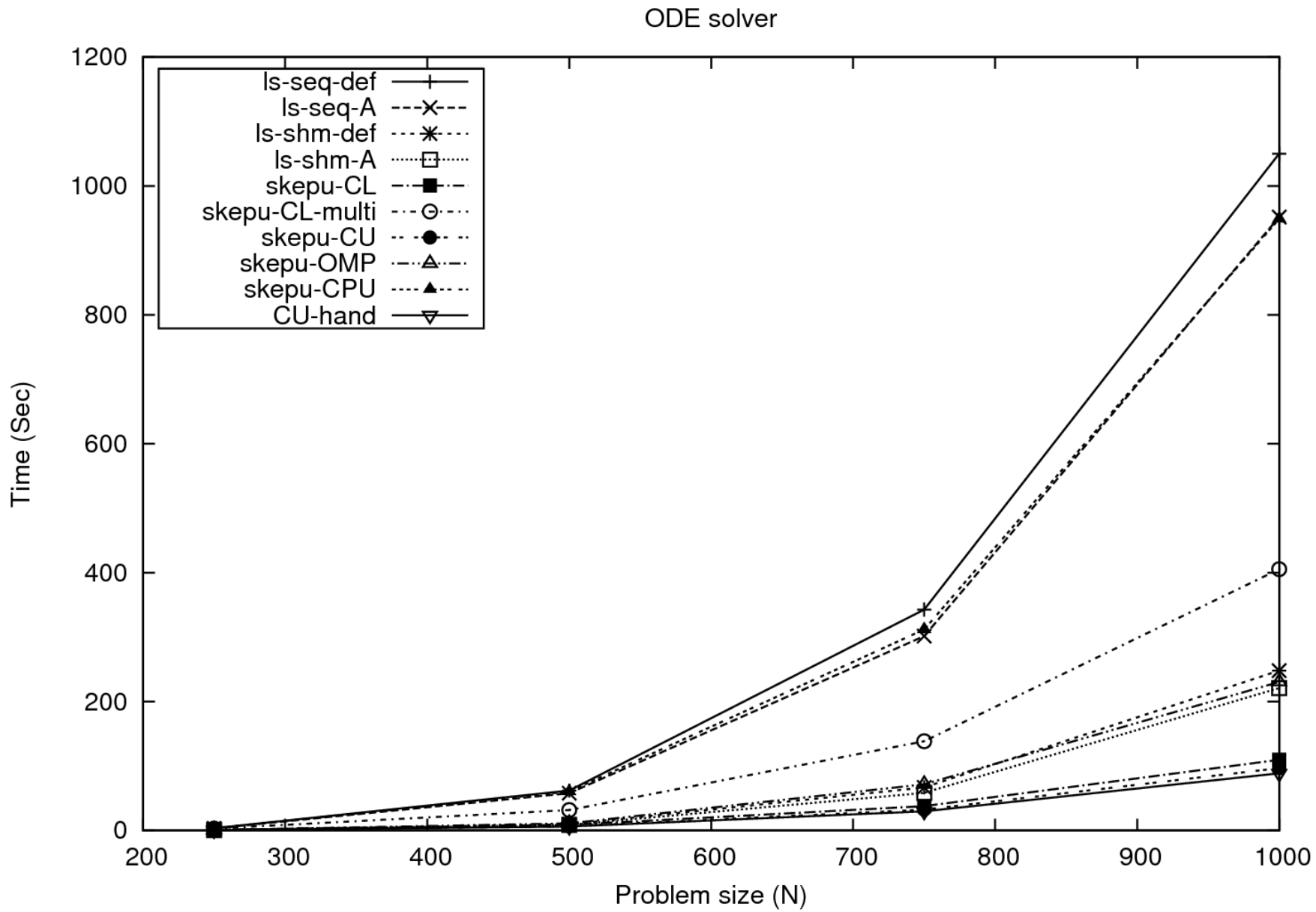
Gaussian Blur



Dot Product


Dot Product without memory transfer time
CUBLAS
SkePU CUDA
SkePU OpenCL Single
SkePU OpenCL Multi
Time (Sec) (1000 runs)
0
2
4
6
8
10
12
14
16
18
20
1e+07
2e+07
3e+07
4e+07
5e+07
6e+07
7e+07
8e+07
9e+07
1e+08
Vector size (# of doubles)

Runge-Kutta ODE Solver



Conclusions

- Skeleton programming is a viable approach to high-level portable GPU computing
- A general interface with multiple back ends provides flexibility to use different architectures
- Memory transfers between host and device can be a bottleneck. Lazy memory copying helps to remedy this

Future Work

- More skeletons, (Scan has been implemented after the article was written). Also task parallel skeletons.
- Two-dimensional data structure
- (Auto)-tuning (Work in progress)

SkePU is available for download at:

http://www.ida.liu.se/~chrke/skepu