
Usman Dastgeer, Johan Enmyren, Christoph W. Kessler

{usmda, x10johen, chrke} @ida.liu.se

Linköping University, Sweden

* Funded by EU FP7 Project PEPPHER
Outline

- Background
- SkePU
- Tuning and Prediction
- Conclusions & Future work
Background

- Trend towards multi/many core computing.
- GPGPU Computing – CUDA, OpenCL
- Different parallel programming models for different architectures
  - portability and performance issue
- Skeleton programming
- SkePU, skeletons for GPU/CPU
SkePU

- C++ template library
- Multiple back-ends
- Smart Containers
- User functions
- Skeletons
- Multi-GPU
- Various helpers
- **Tunable**
Smart Containers

- Vector-type (modeled after std::vector)
- Matrix-type

```cpp
skepu::Vector<double> v0(1000,2);
skepu::Matrix<double> m0(10,20,3);
```

- Handles memory transfers between host and device
- Lazy memory copying
Lazy Memory Copying

- A SkePU container keeps track of where data resides.
- Does not copy data back from device memory until you access it on host (main) memory.
User Functions

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

\[
\text{BINARY\_FUNC(plus, double, a, b,}
\]
\[
\text{\hspace{1cm}return a+b;}
\]
\[
\text{)}
\]
Skeletons

- Objects, overloading operator()
- Map
- Reduce
- MapReduce
- MapOverlap
- MapArray
- Scan (Prefix Sum)
- Farm (Task parallel)

Visit skepu homepage for more information!
Example: Dot Product

```cpp
#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;
}
```
Example: Dot Product

```cpp
#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;
}
```
Backends

- Sequential CPU
- Multiple-CPUs - OpenMP
- GPU CUDA
- GPU OpenCL*
- Multiple-GPUs OpenCL*
- (Multiple-GPUs CUDA)

* NVIDIA OpenCL implementation
Tuning Potential (2 Tesla C1060, 8 CPUs)

Reduce for different vector sizes.
Problem

How to choose which backend configuration to use for a given skeleton call?
Tuning support in SkePU

- Execution Plan
  - Sets optimal back-ends with optimal configuration for different input ranges.
  - Each skeleton uses an execution plan.
  - Performance portability.

```c++
skepu::Reduce<plus> globalSum(new plus);
skepu::ExecPlan plan;
plan.add(1,3500, skepu::CPU_BACKEND);
plan.add(3501,3200000, skepu::OMP_BACKEND, 8);
plan.add(3200001,5400000, skepu::CL_BACKEND, 65536, 128);
plan.add(5400001,MAX_INT, skepu::CLM_BACKEND, 65536, 128);
globalSum.setExecPlan(plan);
```
Tuning Potential (2 Tesla C1060, 8 CPUs)

Reduce for different vector sizes.

Vector Size (# elements)

Time (ms)
Problem

How to choose which backend configuration to use for a given skeleton call?

How to generate execution plan(s) automatically?
Tuning Scope

- How to tune choice between different back-ends for:
  - A single skeleton call
    - With repetitive execution (execution frequency)
    - With varying user functions
  - Multiple skeleton calls
    - Composition problem
Choosing Optimal Backend

1. Determine best configuration for each backend for different problem sizes for a skeleton.
   • A heuristic algorithm based on GP
2. Feed off-line calculated estimates and configuration information for each backend to the prediction framework.
3. Prediction framework will generate an execution plan.
Tuning parameters

- Uses off-line, learning approach to record different parameters for each backend:
  - Copy-Up-Time (host to device)
  - Copy-Down-Time (device to host)
  - Kernel Execution time
  - Total time
  - Overhead time
- Build a model
  - Fixed Exec time (F) + Repetitive Exec time (R)
Fixed & Repetitive time

```cpp
int main()
{
    skepu::MapOverlap<overlap> skel(new overlap);
    skepu::Vector<double> v0(1000,2);

    for(int i=0; i<5; i++)
        skel(v0);

    std::cout<<"Result: " <<v0 <<"\n";
    return 0;
}
```
Prediction Framework

- Use pre-calculated estimates to predict execution time for:
  1. A single skeleton call with a given execution frequency \((\geq 1)\).
  2. A single skeleton call with different user functions than the estimates.
Estimation - execution frequency

A single skeleton call with a given execution frequency (N, \( \geq 1 \)).

\[
T(N) = F + (R \times N)
\]

Estimates for different back-ends are used to decide which backend to use for a given skeleton call.
Estimation - execution frequency
(2 Tesla C1060, 8 CPU cores)

Predictions and actual executions for repetitive executions - Reduce skeleton
Changing User function

To predict for different user functions:
- Record execution time of new user function through benchmarks
  - Fixed type of user functions
- The prediction framework interpolate by either:
  - Taking overall average ratio for different problem sizes.
  - Taking intermediate ratios for different problem sizes.
Changing User function
(2 Tesla C1060, 8 CPU cores)
Choosing Optimal Backend
(2 Tesla C1060, 8 CPU cores)

Comparing Tuned MapReduce (DotProduct) with other CL, CLM - 10 Executions

<table>
<thead>
<tr>
<th>CLM</th>
<th>4096, 32</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLM</td>
<td>65536, 512</td>
<td>Green</td>
</tr>
<tr>
<td>CLM</td>
<td>8192, 128</td>
<td>Blue</td>
</tr>
<tr>
<td>CL</td>
<td>65536, 512</td>
<td>Pink</td>
</tr>
<tr>
<td>CL</td>
<td>8192, 128</td>
<td>Cyan</td>
</tr>
<tr>
<td>CL</td>
<td>4096, 32</td>
<td>Yellow</td>
</tr>
<tr>
<td>TUNE</td>
<td></td>
<td>Black</td>
</tr>
</tbody>
</table>

Time (ms)

Vector Size (# elements)
Choosing Optimal Backend
(2 Tesla C1060, 8 CPU cores)

Comparing Tuned MapReduce (DotProduct) with CL, CLM, OMP - Single Execution

- CLM 8192, 128
- CLM 65536, 512
- CL 8192, 128
- OMP 16T
- OMP 8T
- OMP 4T
- TUNE

Time (ms)

Vector Size (# elements)
Tuning Composition of skeletons

- Tuning application consisting of multiple skeleton calls.
- An ODE Solver ported to SkePU

```plaintext
... s1(v1, v2, out1); // MapArray skeleton call
while (...) {
    s2(out1, v1); // Map call 1
    for (...) {
        s3(v1, v3); // Map call 2
    }
    res = s4(v2) // Reduce call
}
    s5(out1, v5, out2); // Map call 3
} res2 = s6(out1, out2); // MapReduce call
...```
Tuning Composition of skeletons

- Integration with StarPU runtime system.
  - Usage of multiple resources (CPUs, GPUs)
  - Dynamic scheduling support
Conclusion and Future work

- Skeleton programming is helpful for programmability, portability and performance.
- Auto-tuning SkePU is viable:
  - Skeletons model regular patterns.

Future Work
- More skeletons (pipe, divide and conquer)
- Using parametric performance models while using StarPU.
SkePU is available for download at:

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

*** Thank you ***
Calculating parameters

- For GPU back-ends:
  - Fixed time
    - Copying time: effective bandwidth (# bytes) except CUDA Multi GPU systems
  - Repetitive time
    - Kernel Execution time
      - Depends upon skeleton + user function
      - Currently measured but will use analytical model (work in progress)
  - Overhead Time
Calculating parameters Cont.

- For OpenMP back-end:
  - Through execution (No explicit copying information)
  - Fixed time
    - Overhead (second – first execution), to hide caching effects
  - Repeat time
    - Total - Overhead time.
Evaluation

Two architectures for experiments

1. Dual-quadcore Intel(R) Xeon with 2 NVIDIA GT200 (Tesla C1060) GPUs.
2. Intel Core 2 Duo E6600 with one GeForce GTS250 GPU
```cpp
struct plus
{
    skepu::FuncType funcType;
    std::string func_CL;
    std::string funcName_CL;
    std::string datatype_CL;
    plus()
    {
        funcType = skepu::BINARY;
        funcName_CL.append("plus");
        datatype_CL.append("double");
        func_CL.append("double plus(double a, double b)\n        {{\n        " return a+b;\n        }}\n        ");
    }
}

double CPU(double a, double b)
{
    return a+b;
}
__device__ double CU(double a, double b)
{
    return a+b;
}
```
Map

Input a  \[ a_1 \quad a_2 \quad \ldots \quad a_n \]

Input b  \[ b_1 \quad b_2 \quad \ldots \quad b_n \]

Result  \[ f(a_1, b_1) \quad f(a_2, b_2) \quad \ldots \quad f(a_n, b_n) \]
Reduce

Result = 1 \cdot 2 \cdot \ldots \cdot n
MapReduce

Input a  \[ a_1, a_2, \ldots, a_n \]

Input b  \[ b_1, b_2, \ldots, b_n \]

Result  \[ f(a_1,b_1), f(a_2,b_2), \ldots, f(a_n,b_n) \]
MapOverlap

Input a:

\[ a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, \ldots, a_n \]

Result:

\[ f(a_2, a_3, a_4, a_5, a_6) \]

Overlap: 2
MapArray

Input a
- a1
- a2
- ...
- an

Input b
- b1
- b2
- ...
- bn

Result
- f(a1,b1,...,bn)
- f(a2,b1,...,bn)
- ...
- f(an,b1,...,bn)
Scan (Prefix Sum)

Input: \( a_1, a_2, \ldots, a_n \)

Result: \( r_1, r_2, \ldots, r_n \)

\[
\begin{align*}
\text{Result} &= \\
&= \begin{array}{ccc}
& a_1 & a_2 & \ldots & a_n \\
\hline
r_1 & a_1 & & & \\
r_2 & a_1 \cdot a_2 & & & \\
& \ldots & & & \\
r_n & a_1 \cdot a_2 \ldots \cdot a_\text{n} & & & \\
\end{array}
\end{align*}
\]