Solving the Kuramoto Oscillator Model on Random Graphs

Jeffrey Kelling,
Géza Ódor, Sibylle Gemming

12th July 2019

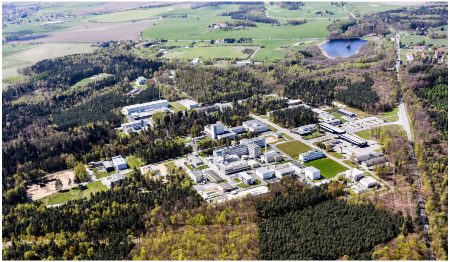
NVIDIA GPU CENTER OF EXCELLENCE

HZDR

HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

Where am I from?

- outside of Dresden, Germany



Jürgen-M. Schulter http://dresden-luftfoto.de

about me:

- member of computational science group
- background in statistical and theoretical solid state physics

Content

Introduction

The Kuramoto Model

- describes a network of coupled oscillators
- system of ordinary differential equations (ODEs)

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \neq j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

$\Rightarrow$ integration to study time-evolution

Member of the Helmholtz Association
Jeffrey Kelling, Géza Ódor, Sibylle Gemming | FWCC | http//www.hzdr.de

Using things that already exist

- `boost::numeric::odeint` odeint.com
  - template library of ODE solvers
  - `boost::numeric` supports various vector backends for accelerators: e.g. Thust (CUDA), VexCL (CUDA/OpenCL)

Using things that already exist

- `boost::numeric::odeint` odeint.com
  - template library of ODE solvers
  - `boost::numeric` supports various vector backends for accelerators: e.g. Thust (CUDA), VexCL (CUDA/OpenCL)
- VexCL
  - library for offloading vector expressions via CUDA or OpenCL
  - direct support for custom kernels

Using things that already exist

- `boost::numeric::odeint` odeint.com
  - template library of ODE solvers
  - `boost::numeric` supports various vector backends for accelerators: e.g. Thust (CUDA), VexCL (CUDA/OpenCL)
- VexCL
  - library for offloading vector expressions via CUDA or OpenCL
  - direct support for custom kernels

- we use 4th order Runge-Kutta form `odeint`

⇒ computing derivates reamins and is the most time-consuming part

VexCL

+ offloading vector expressions, which is what `boost::compute` relies on

```
std::vector<double> host(N, 2);
vex::vector<double> device(context, host);

device *= device;

vex::copy(device, host);
```

VexCL

+ offloading vector expressions, which is what `boost::compute` relies on

```
std::vector<double> host(N, 2);
vex::vector<double> device(context, host);

device *= device;

vex::copy(device, host);
```

– pseudo single-source: kernel compilation at runtime

– no custon function templates

⇒ have to use custom kernel and inject string to get "template"

Shape of the Network I

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \neq j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- parallel implementations depend on network topology

Shape of the Network I

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \neq j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- parallel implementations depend on network topology
- fully connected graph:
  - N^2-problem, vectorizable

Jeffrey Kelling, Géza Ódor, Sibylle Gemming | FWCC | http//www.hzdr.de

Shape of the Network I

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \neq j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- parallel implementations depend on network topology
- fully connected graph:
  - N^2-problem, vectorizable
- regular lattice / band matrix:
  - stencil integration

Shape of the Network II

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- sparse, random graph

Shape of the Network II

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- sparse, random graph
  - requires explicit storage network topology
    i.e. sparse representation, neighbor lists
  - random neighbor sums

Shape of the Network II

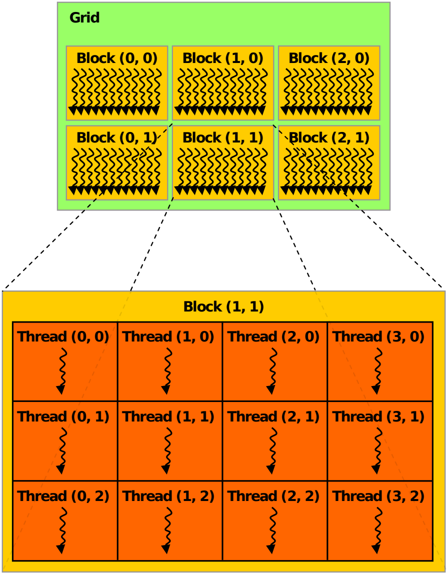
$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- sparse, random graph
  - requires explicit storage network topology
    i.e. sparse representation, neighbor lists
  - random neighbor sums

⇒ **techniques for SIMT vectorization by tuned operation and memory ordering**

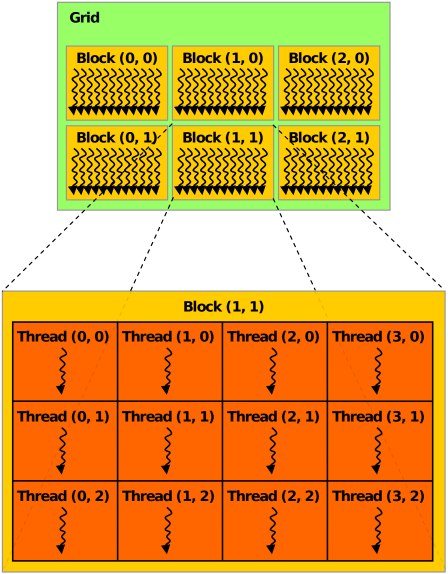
Implementation

Recap: GPU Architecture



- Single-Instruction-Multiple-Thread (SIMT) workers in lock-step
- vector memory transactions (> 64 byte)

Recap: GPU Architecture



- Single-Instruction-Multiple-Thread (SIMT) workers in lock-step
- vector memory transactions (> 64 byte)
- actually, the same goes for CPU (SIMD + Cache-lines) GPUs just have wider vectors and more simultaneous multi threading (SMT)

Vectorization I

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- vectorizing over oscillators j
  - sum over k too short on average ($\lesssim 51$), too little parallelism
  - avoid need for reduction

Member of the Helmholtz Association
Jeffrey Kelling, Géza Ódor, Sibylle Gemming | FWCC | http//www.hzdr.de

Vectorization II: Memory Locality

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- data local to js is continuous

Vectorization II: Memory Locality

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- data local to js is continuous
- data in naive neighbor lists would lead to scattered memory access

Vectorization II: Memory Locality

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- data local to *j*s is continuous
- data in naive neighbor lists would lead to scattered memory access
- data of remote site *k* is at random positions

Vectorization II: Memory Locality

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- data local to js is continuous
- data in naive neighbor lists would lead to scattered memory access
- data of remote site k is at random positions

+ no branches, vectorizable expression

Vectorization II: Memory Locality

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- data local to js is continuous
- data in naive neighbor lists would lead to scattered memory access
- data of remote site k is at random positions

+ no branches, vectorizable expression

– no predictable data reuse within thread block: shared memory of not useful, but caches may be

– low computational density, mostly streaming data

Vectorization II: Memory Locality

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- data local to js is continuous
- data in naive neighbor lists would lead to scattered memory access
- data of remote site k is at random positions

+ no branches, vectorizable expression
− no predictable data reuse within thread block:
  shared memory of not useful, but caches may be
− low computational density, mostly streaming data
⇒ maximize memory locality of reads
⇒ minimize load imbalances

HZDR

Memory Layout

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

Memory Layout

#nodes	… with n^{th} links;	prefix sum
⋮	array of n^{th} links	⋮

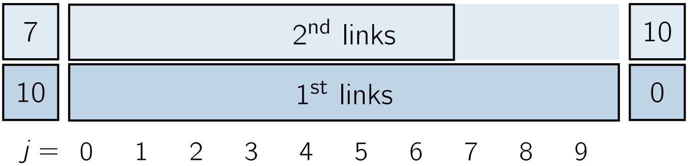
10	1^{st} links	0

$j =$ 0 1 2 3 4 5 6 7 8 9

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

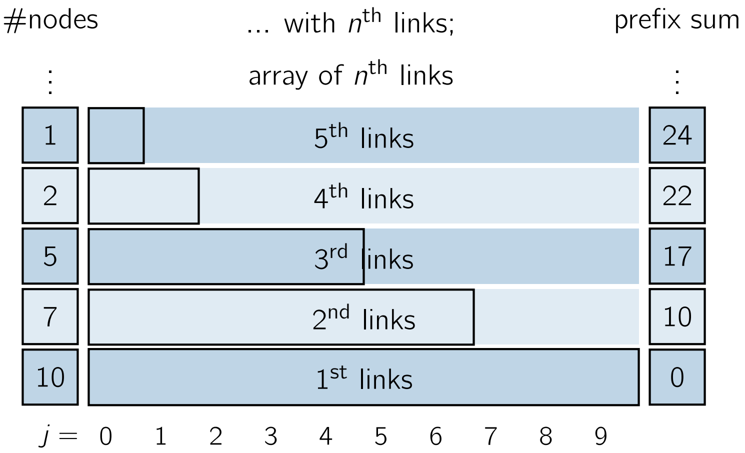
Memory Layout

#nodes ⋮ … with n^{th} links; array of n^{th} links ⋮ prefix sum ⋮



$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

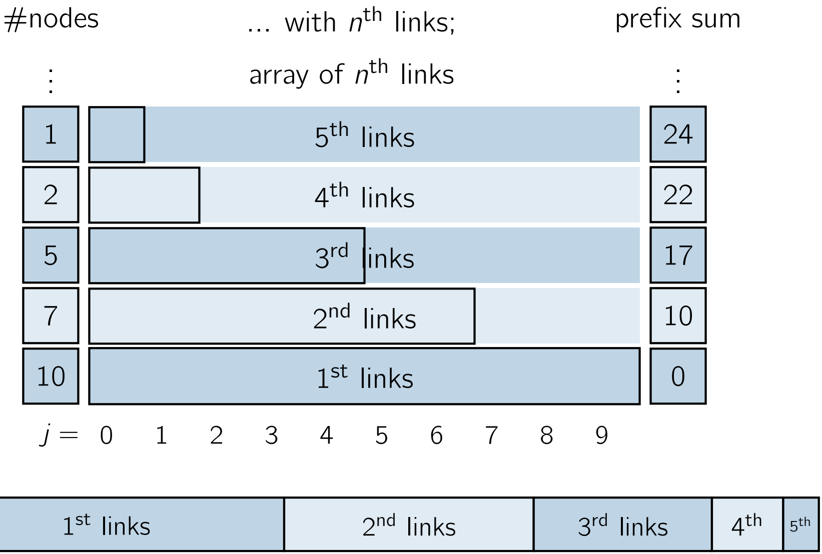
Memory Layout



$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

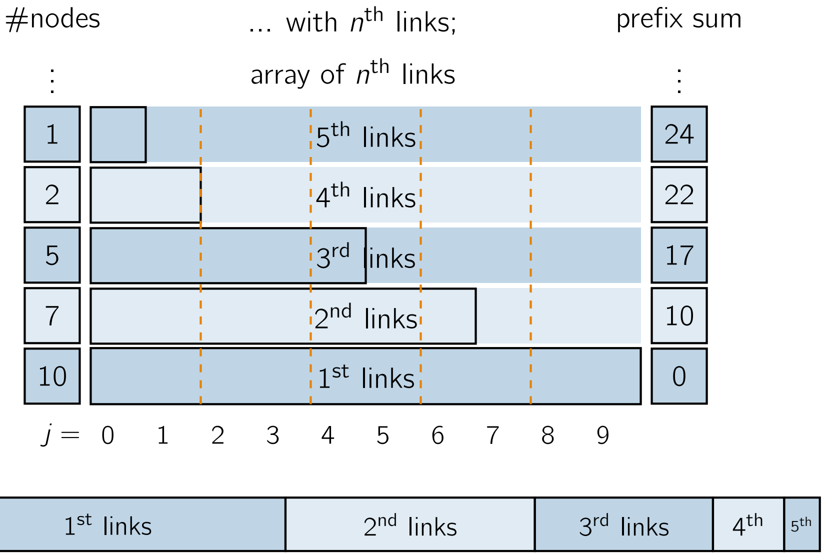
HZDR

Memory Layout



Jeffrey Kelling, Géza Ódor, Sibylle Gemming | FWCC | http//www.hzdr.de

Memory Layout

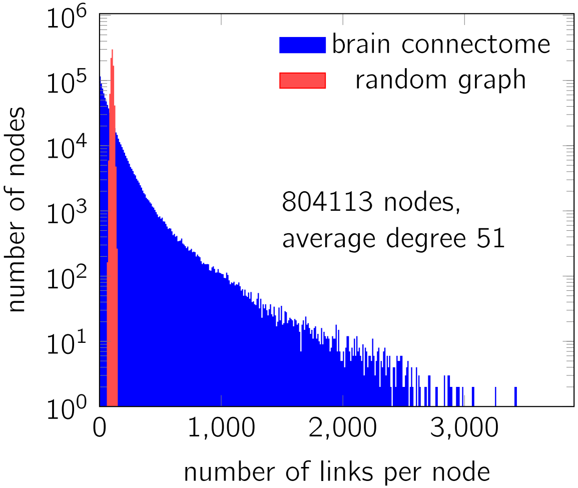


Jeffrey Kelling, Géza Ódor, Sibylle Gemming | FWCC | http//www.hzdr.de

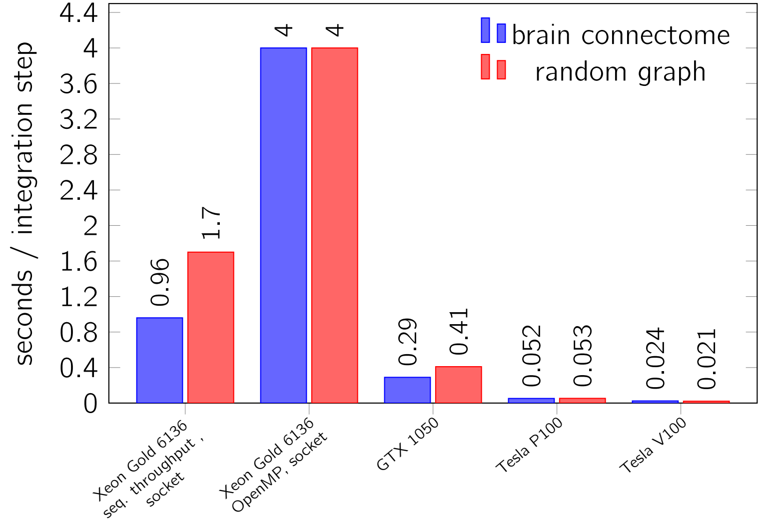
Performance

Networks

long-tailed human brain connectome vs. random graph



Benchmarks


seconds / integration step
4.4
4
3.6
3.2
2.8
2.4
2
1.6
1.2
0.8
0.4
0
brain connectome
random graph
0.96
1.7
4
4
0.29
0.41
0.052
0.053
0.024
0.021
Xeon Gold 6136 seq. throughput , socket
Xeon Gold 6136 OpenMP, socket
GTX 1050
Tesla P100
Tesla V100

Efficiency

$$\frac{\partial \phi_j(t)}{\partial t} = \omega_j + \sum_{k \text{ NN of } j} \lambda_{jk} \cdot \sin\left[\phi_k(t) - \phi_j(t)\right]$$

- profile on tesla P100
  - global load efficiency: $\sim 47\,\%$
    saturating gross load bandwidth to $\sim 70\,\%$
  - data requests dominant stall reason $\sim 50\,\%$

$\Rightarrow$ remains memory-latency bound, due to random accesses to neighbors

Conclusion

Summary

- efficient inmplementation for integration on random graphs $\sim 20\times$ improved throughput over single CPU socket.
- easily adaptable to other models: we use it for 2nd order Kuramoto, too

Summary

- efficient inmplementation for integration on random graphs $\sim 20\times$ improved throughput over single CPU socket.
- easily adaptable to other models: we use it for 2nd order Kuramoto, too

- *handle randomness on GPU by sorting data to maximise the likelyhood of efficient memory acceess and load balance*

Acknowledgments

Thank You.