



Click to edit Master subtitle style

New Languages for High Performance Computing

Iris Christadler, Leibniz Supercomputing Centre, Germany
August 2010, PDC/KTH Summer School



Outline

1. **The free lunch is over**
Multicore CPUs are ubiquitous
2. **Hardware accelerators**
New languages enter the HPC world
3. **The quest for a parallel language**
Examples of emerging languages



The free lunch is over

“But if you want your application to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written concurrent application. And that’s easier said than done, because not all problems are inherently parallelizable and because concurrent programming is hard.”

The Free Lunch Is Over

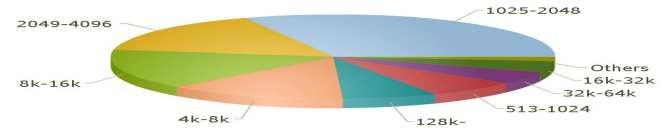
A Fundamental Turn Toward Concurrency in Software

By Herb Sutter

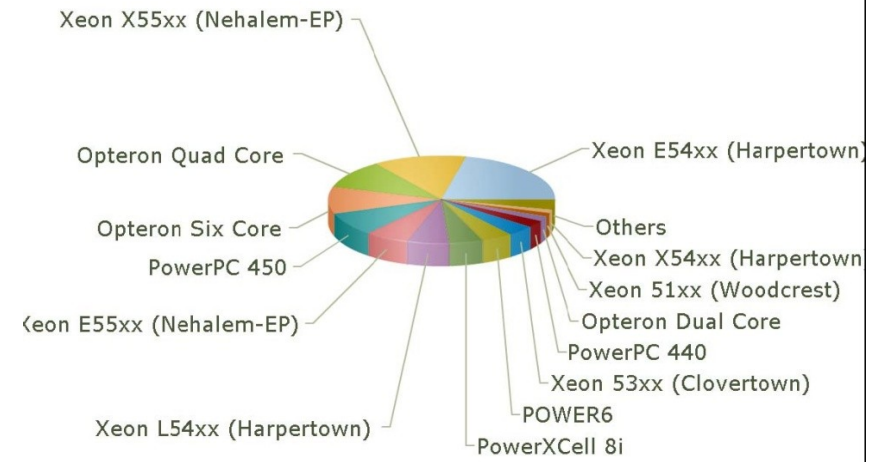
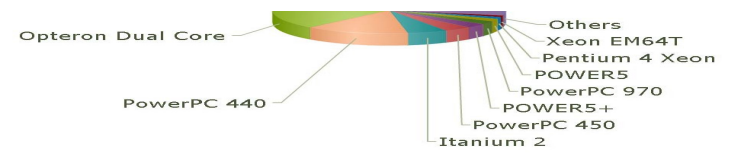
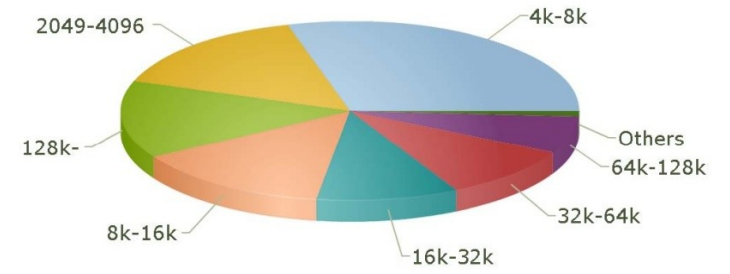
[<http://www.gotw.ca/publications/concurrency-ddj.htm>]



Number of Processors / Performance
November 2007



Number of Processors / Performance
November 2009



Images: www.top500.org

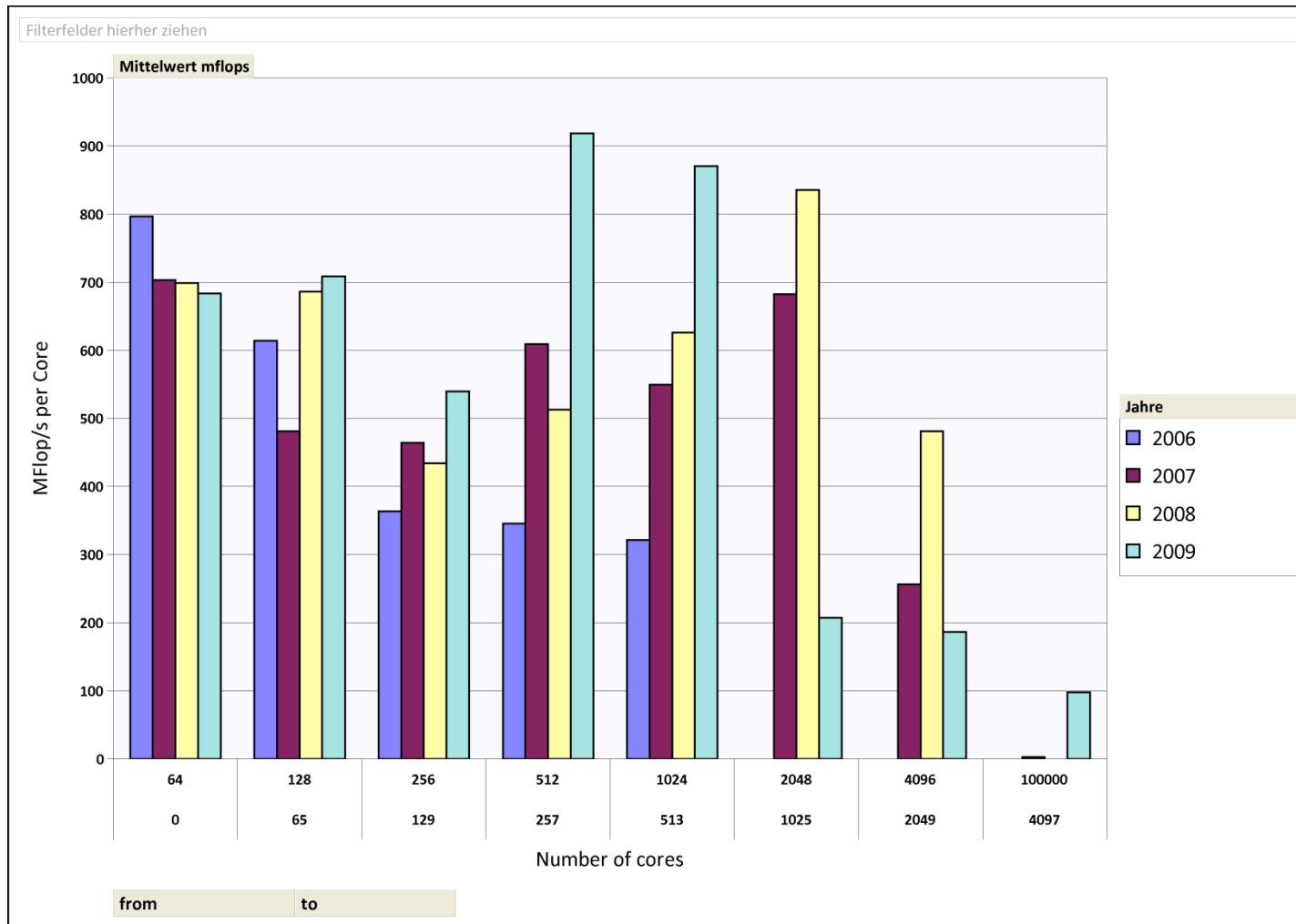
New Languages for High Performance Computing, Iris Christadler, LRZ

August 2010



LRZ's job mix


Performance per core versus number of cores






My favorite "Dongarra" slides

[<http://www.netlib.org/utk/people/JackDongarra/SLIDES/dongarra-isc2004.pdf>]
[<http://www.netlib.org/utk/people/JackDongarra/SLIDES/sc09-exascale-panel.pdf>]

 **Real Crisis With HPC Is With The Software**

- ◆ **Programming is stuck**
 - Arguably hasn't changed since the 70's
- ◆ **It's time for a change**
 - Complexity is rising dramatically
 - highly parallel and distributed systems
 - From 10 to 100 to 1000 to 10000 to 100000 of processors!!
 - multidisciplinary applications
- ◆ **A supercomputer application and software are usually much more long-lived than a hardware**
 - Hardware life typically five years at most.
 - Fortran and C are the main programming models
- ◆ **Software is a major cost component of modern technologies.**
 - The tradition in HPC system procurement is to assume that the software is free.

29

 **Some Current Unmet Needs**

- ◆ **Performance / Portability**
- ◆ **Fault tolerance**
- ◆ **Better programming models**
 - Global shared address space
 - Visible locality
- ◆ **Maybe coming soon (since incremental, yet offering real benefits):**
 - Global Address Space (GAS) languages: UPC, Co-Array Fortran, Titanium)
 - "Minor" extensions to existing languages
 - More convenient than MPI
 - Have performance transparency via explicit remote memory references
- ◆ **The critical cycle of prototyping, assessment, and commercialization must be a long-term, sustaining investment, not a one time, crash program.**

30

IESP: The Need



- The largest scale systems are becoming more complex, with designs supported by consortium
 - ▣ The software community has responded slowly
- Significant architectural changes evolving
 - ▣ Software must dramatically change
- Our ad hoc community coordinates poorly, both with other software components and with the vendors
 - ▣ Computational science could achieve more with improved development and coordination

A Call to Action



- Hardware has changed dramatically while software ecosystem has remained stagnant
- Previous approaches have not looked at co-design of multiple levels in the system software stack (OS, runtime, compiler, libraries, application frameworks)
- Need to exploit new hardware trends (e.g., manycore, heterogeneity) that cannot be handled by existing software stack, memory per socket trends
- Emerging software technologies exist, but have not been fully integrated with system software, e.g., UPC, Cilk, CUDA, HPCS
- Community codes unprepared for sea change in architectures
- No global evaluation of key missing components

www.exascale.org

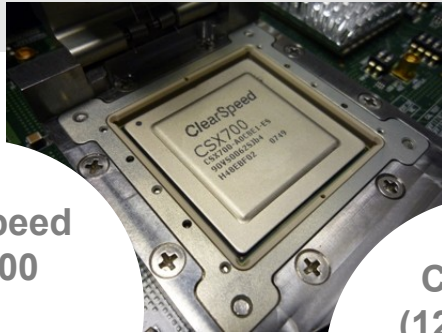


- New languages enter the HPC world

Hardware Accelerators



Hardware in 2008/9



ClearSpeed
CSX700

96 GF dp

CATS 700
(12 CSX700)

1.152 TF dp



PowerXCell8i
(8 SPU)

205 GF sp
102 GF dp

QS22 blade
(16 SPU)

410 GF sp
205 GF dp

2.53GHz
Nehalem-EP

20 GF sp
10 GF dp

8 N-EP
cores

162 GF sp
81 GF dp



C1060 GPU

~1 TF sp
78 GF dp



S1070
"Tesla"

~4 TF sp
312 GF dp



Hardware 2010

http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf

Intel MIC architecture "Knights Ferry"

32 cores @ 1.2 GHz
4 threads/core
8 MB shared coherent cache
1-2 GB GDDR5

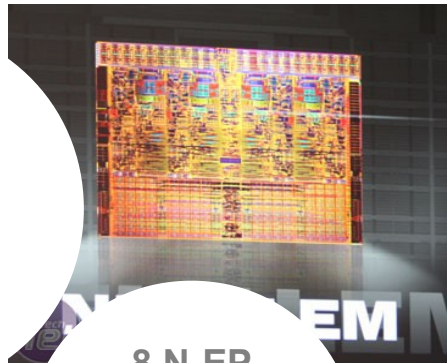
"Knights corner"
1st MIC product

22nm process
>50 cores



2.53GHz
Nehalem-EP

20 GF sp
10 GF dp



8 N-EP
cores

162 GF sp
81 GF dp

C2070
GPU

1.0 TF sp
0.5 TF dp

S2070
"Fermi"

4 TF sp
2 TF dp



NVIDIA Tesla S2070 (Quelle: NVIDIA)



Pros and Cons for HWA

Pros:

- HWA can help to tackle research problems
(in many cases they are simply less expensive than traditional solutions)
- HWA help to shrink the physical footprint of systems
- HWA can help to reduce the power consumption
both of the machine and the cooling system

Cons:

- You need to make use of them,
otherwise they simply waste energy
- HWA will probably increase the error-rate
- HWA are no solution for scalability problems
- HWA are difficult to program (?)

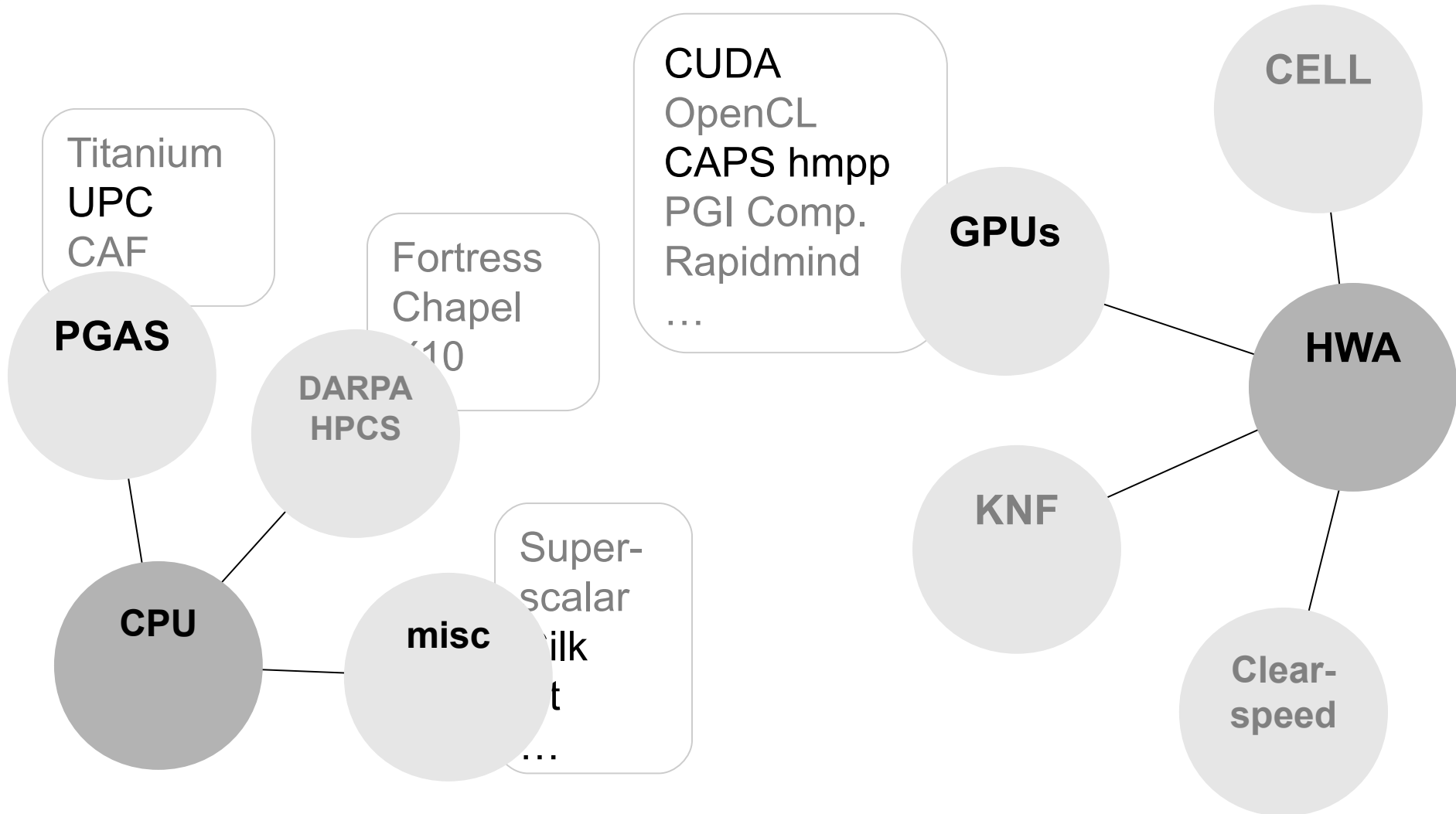


- Examples of emerging languages

The quest for a parallel language



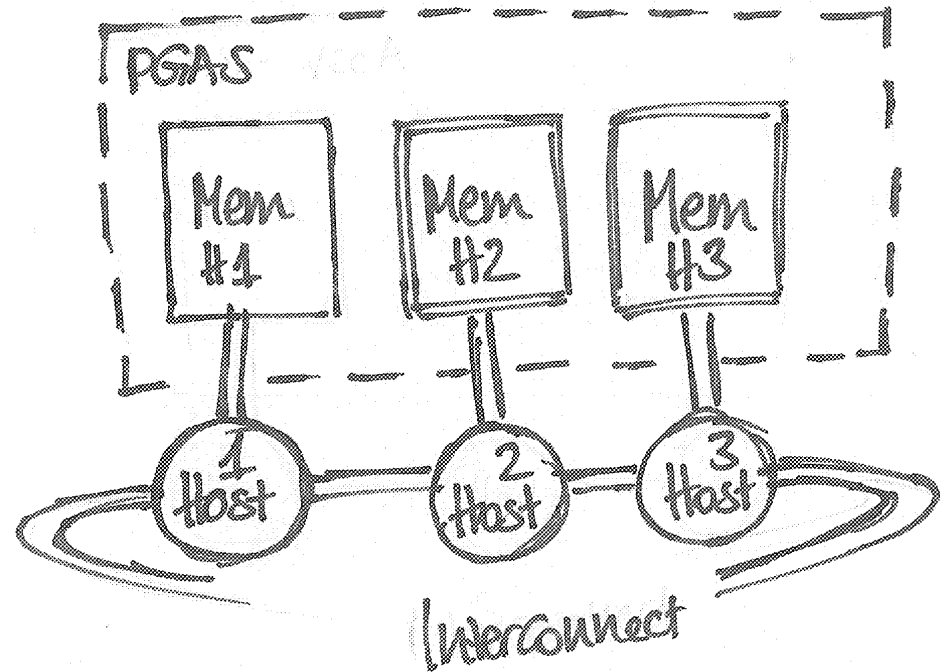
Overview



PGAS

Partitioned Global Address Space language

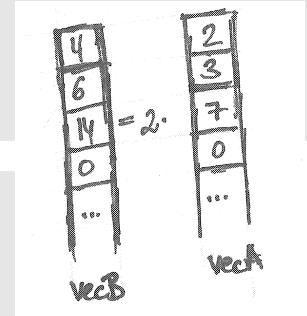
- The concept and it's different implementations
- UPC vs. CAF: some fundamental differences
- Performance improvements?
- Ease-of-use?
- Planned hardware (interconnect) support





UPC

Unified Parallel C – a PGAS example



```
# include <upc.h>
# define CHUNK (int) (N/THREADS)

shared [CHUNK] double vecA[N];
shared [CHUNK] double vecB[N];
...

int main () {
    ...
    if (MYTHREAD == 0) {
        printf("Main thread\n");
    }
    ...
    upc_barrier;
    upc_forall (j=0; j<N; j++; &a[j]) {
        vecB[j] = 2*vecA[j];
    }
    upc_barrier;
    ...
}
```

CUDA

The Compute Unified Device Architecture for Nvidia GPUs

host code:

```
// allocate memory on gpu
cudaMalloc (ptr, size);
cudaMemcpy (dst, src, size, dir:host2dev);

// launch kernel
kernel<<<gridSize, blockSize>>> (funcParams);

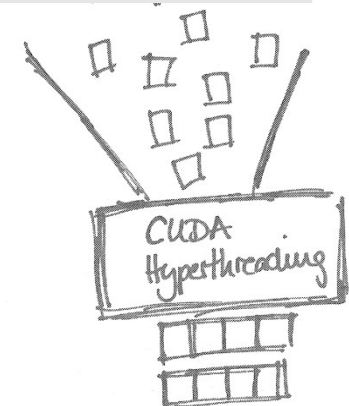
// copy results back
cudaMemcpy (... , ..., dir:dev2host);
cudaFree;
```

device code:

```
__global__ void kernel(float* vecA, float* vecB, int height, int width){

    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        vecB[y*width+x] = 2*vecA[y*width+x];
    }
}
```

- GPGPU programming de-facto standard
- Small lightweight kernels
- Parallelization through hyper-threading
- Hiding cache misses and latency through threading





CAPS hmpp

GPGPU programming using simple directives

```
#pragma hmpp kernel codelet, target=CUDA,  
        args[vecA, height, width].io= in, args[vecB].io=inout  
void kernel (float *vecA, float *vecB, int height, int width)  
for (x= 0; x<width; x++)  
    for (y= 0; y<height; y++)  
        vecB[y*width+x]= 2*vecA[y*width+x];  
}  
  
int main() {  
    ...  
    #pragma hmpp kernel allocate  
    ...  
    #pragma hmpp kernel advancedload, args[vecA,vecB,  
    ...  
    #pragma hmpp kernel callsite &  
    #pragma hmpp kernel args[...].advancedload=true  
    kernel (vecA, vecB, height, width);  
    ...  
    #pragma hmpp kernel release  
}
```

- What is CAPS hmpp?
- Similar to the PGI acc. compiler approach
- Similarities with OpenMP
- Possibilities & Limitations

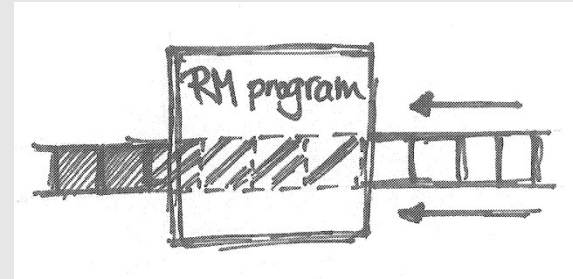


RapidMind

as a placeholder for Intel Ct (not yet publicly released)

```
#include <rapidmind/platform.hpp>
using namespace RapidMind;

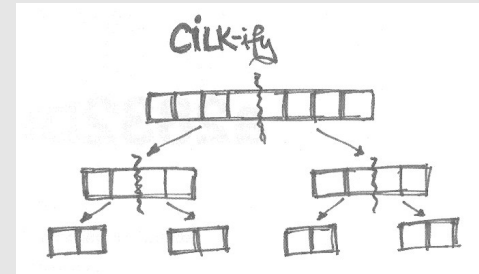
int main () {
    ...
    // declaration
    Array<2, Value4f> vecA;
    Array<2, Value4f> vecB;
    ...
    Program kernel= BEGIN {
        // program definition
        In<Value4f> a;
        Out<Value4f> b;
        b= 2*a;
    } END;
    ...
    // program call
    vecB = kernel(vecA);
    ...
}
```



- „Array computing“/“data stream computing”
- Intel acquired RapidMind
- RapidMind technology will be integrated in Ct
- RapidMind offered backends for: Cell, Cuda, GLSL, x86

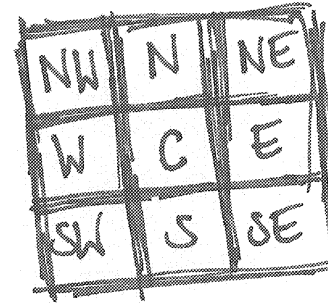
```
cilk double kernel(float *vecA, float *vecB, int i0, int i1){
  if ((i1-i0) == 1) {
    vecB[i0]= 2*vecA[i0];
    return;
  }
  else {
    int im= (i0+i1) / 2;
    double vecL, vecR;
    vecL= spawn kernel(vecA, vecB, i0, im);
    vecR= spawn kernel(vecA, vecB, im, i1);
    sync;
    return;
  }
}
```

```
Cilk int main() {
  ...
  spawn kernel(...);
  sync;
  ...
}
```



- Automatic parallelization and load balancing (!) through recursion
- Divide-and-conquer style programming
- Needs an adaptation of many algorithms

- Now do this exercise for a more



$$C = 4 \cdot C + 2 \cdot (N + W + S + E) + 1 \cdot (NW + \dots)$$

A 2D Stencil Computation



Contact details: Iris Christadler (christadler@lrz.de), LRZ, Germany

Thank you for your attention!