Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

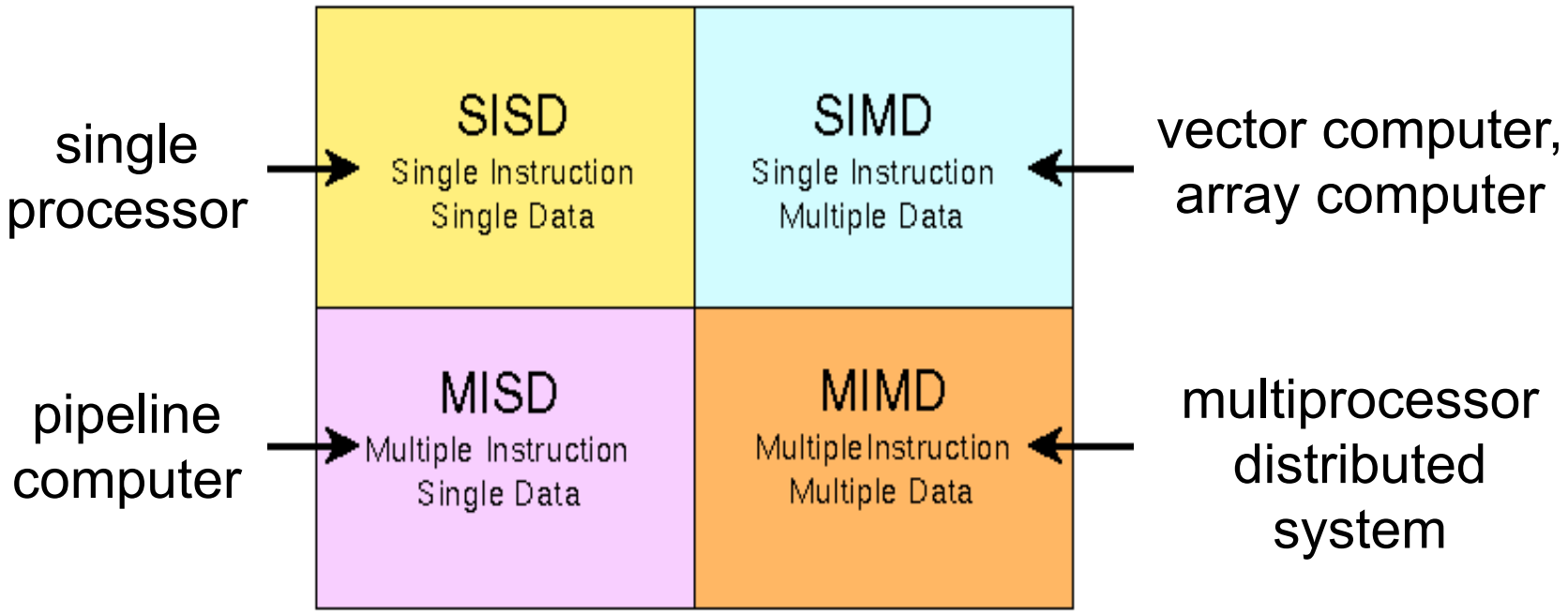
SIMD Systems

Programmierung Paralleler und Verteilter Systeme (PPV)

Sommer 2015

Frank Feinbube, M.Sc., Felix Eberhardt, M.Sc.,
Prof. Dr. Andreas Polze

Computer Classification



M.Flynn, *Very High Speed Computing Systems*, Proceedings of the IEEE, vol 54, 1966, pp. 1901-1909(9)

Programming models - Classification

Hasso Plattner Institut

Creation of parallelism

Explicit

- Coroutines (Modula-2)
- fork & join (cthreads)
- parbegin/parend (Algol 68)
- Processes/Threads (UNIX, Mach, VMS), RPCs
- Futures, OpenCL, OpenMP

vs. Implicit

- Prolog: parallel AND, OR
- Vector expressions (FP, APL)
- Matrix operations (HPF, Intel Ct)

Communication

Message passing

- send/receive primitives
- local (private) variables

vs. Shared address space

- Mutual exclusion primitives
- Similar to sequential programming
- „ease of use“

Specification of parallel execution

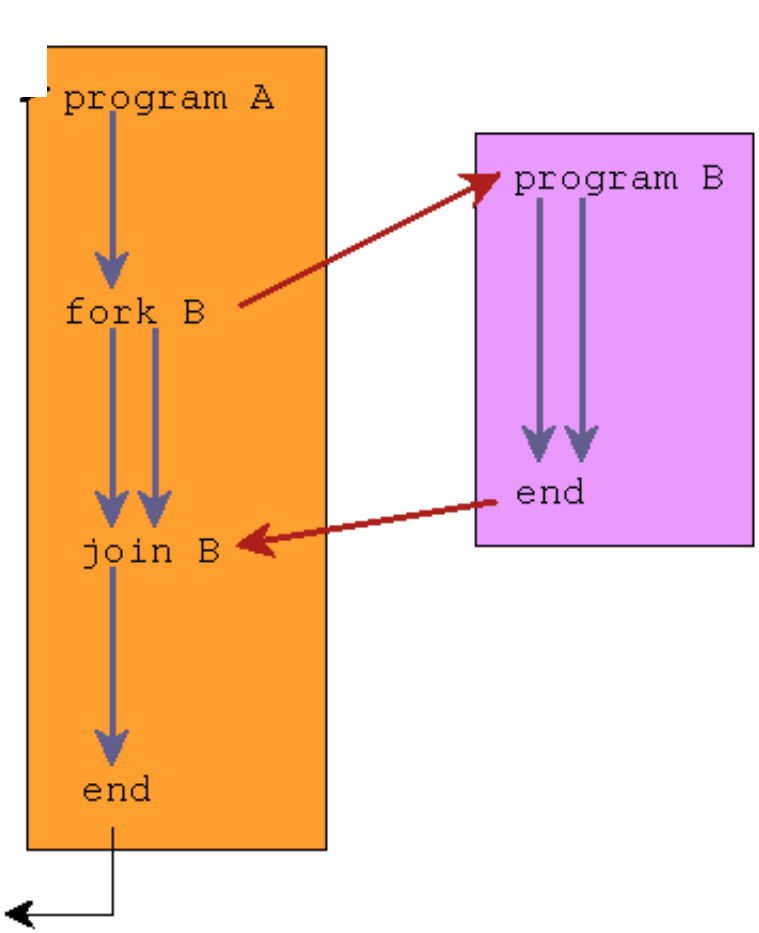
Control parallelism

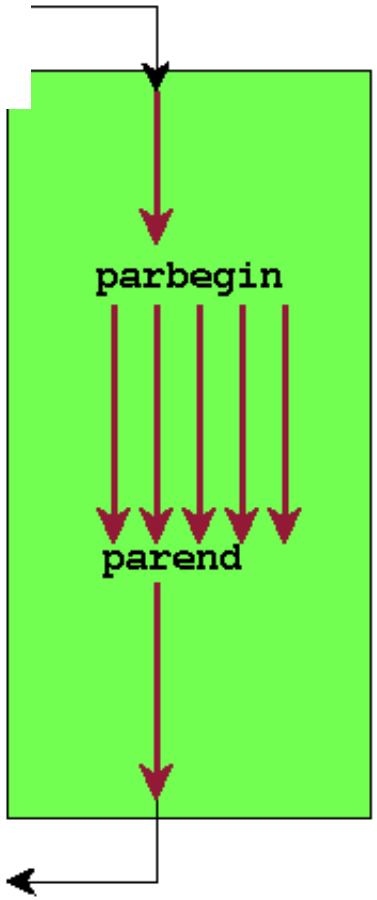
- Simultaneous execution of multiple control flows
- Matches MIMD paradigm
- Difficult to scale

vs. Data parallelism

- Multiple data elements handled simultaneously
- Matches SIMD paradigm
- Single control flow
- Easy to scale

Control Parallelism





Multiprocessor Systems

Hasso Plattner Institut

Symmetric Multiprocessing (SMP)

- Set of equal processors in one system (more SM-MIMD than SIMD)
- Processors share access to main memory over one bus
  - Demands synchronization and operating system support
- Today, every SMP application also works on a uniprocessor machine

Asymmetric multiprocessing (ASMP)

- Specialized processors for I/O, interrupt handling or operating system (DEC VAX 11, OS-360, IBM Cell processor)
- Typically master processor with main memory access and slaves

Large multiprocessor work with NUMA / COMA memory hierarchy

SMP for Scalability and Availability

Hasso Plattner Institut

Advantages

- Performance increase by simple addition of processor card
- Common shared memory programming model
- Easy hardware partitioning, in-built redundancy possible

Disadvantages

- Scale-up is limited by hardware architecture
- Complex tuning of the application needed
- Failover between partitions is solution-dependent

Solves performance and availability problems rather in hardware & operating system than in software

Classification by granularity

$$\text{Granularity} = \frac{t_{\text{basic communication}}}{t_{\text{basic computation}}}$$

Few powerful processor elements:

Coarse grain parallel computers: Cray Y-MP with 8-16 GFlop-Pes

Many relatively weak processor elements:

Fine grain parallel computers: CM-2 (64k 1-bit-processors), MasPar MP-1 (up to 16344 4-bit PEs), C.mmp, KSR-1

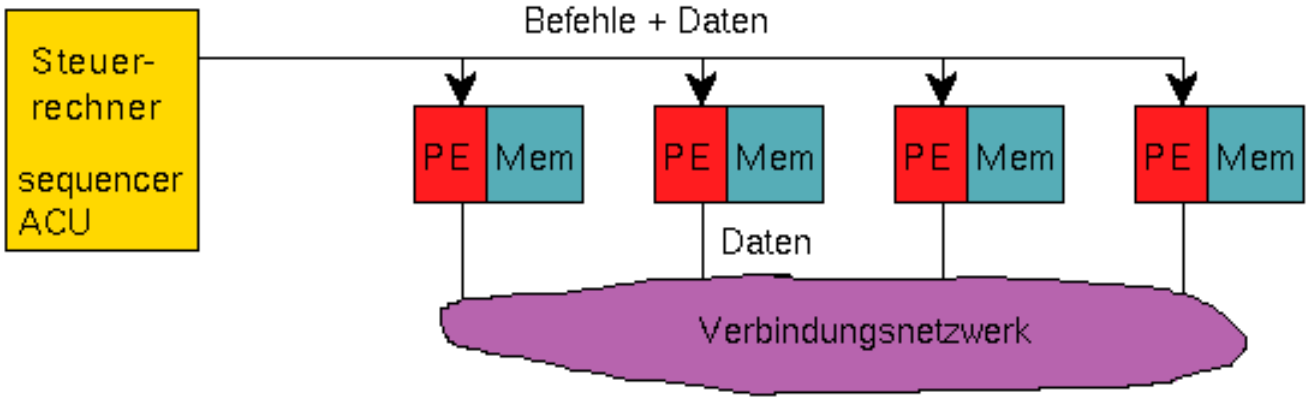
Less than 1000 workstation-class processor elements

Medium grain parallel computers: CM-5, nCUBE2, Paragon XP/S

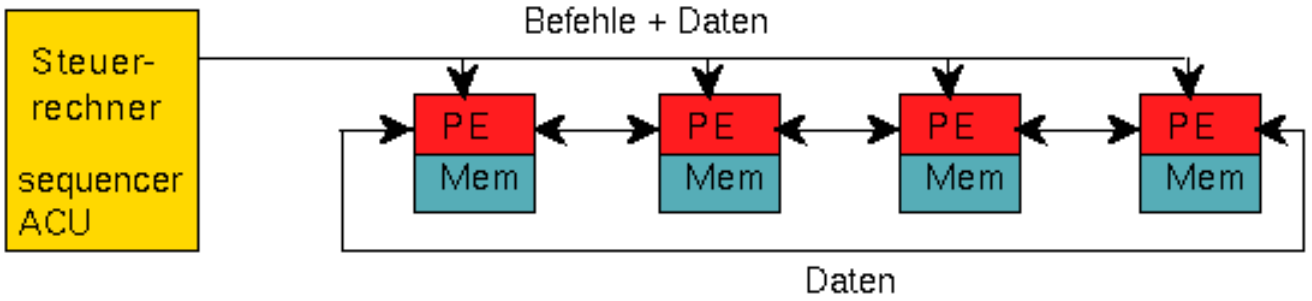
Problem: many algorithms / implementations show limited amount of inherent parallelism

SIMD Computers

Arrayrechner

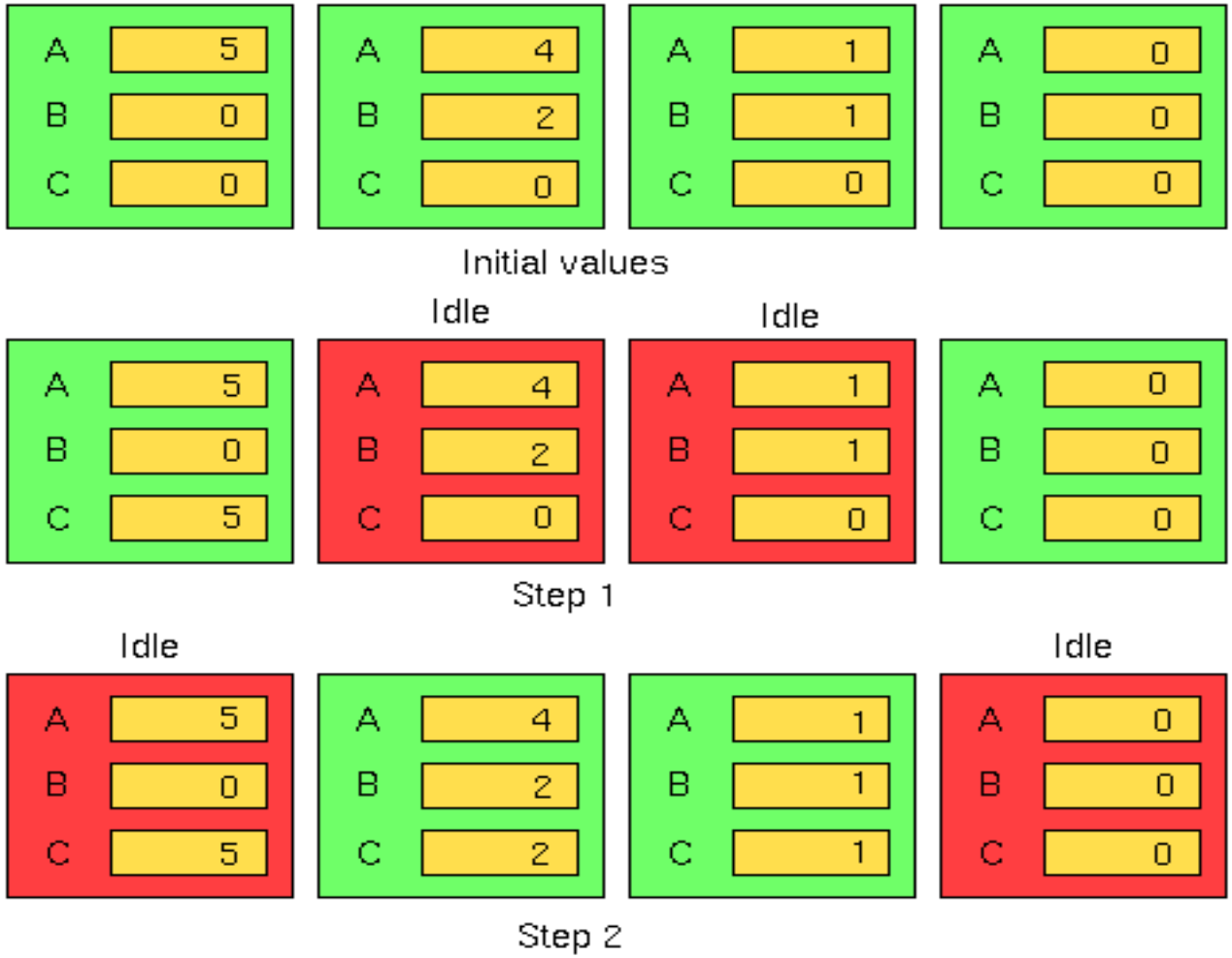


Vektorrechner



SIMD Problems

```
if (B == 0)
        C = A;
else
        C = A/B;
```



SIMD Vector Pipelines

Vector processors have high-level operations for data sets

Became famous with Cray architecture in the 70‘s

Today, vector instructions are part of the standard instruction set

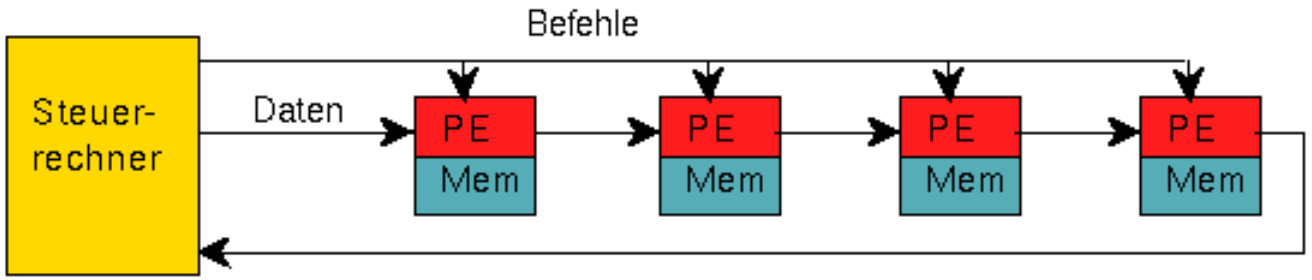
- AltiVec
- Streaming SIMD Extensions (SSE)
  - Example: Vector addition

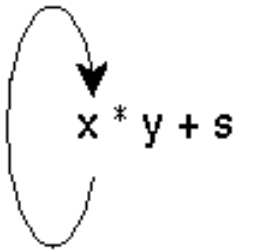
```
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

```
movaps xmm0,address-of-v1
(xmm0=v1.w | v1.z | v1.y | v1.x)

addps xmm0,address-of-v2
(xmm0=v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x)

movaps address-of-vec_res,xmm0
```

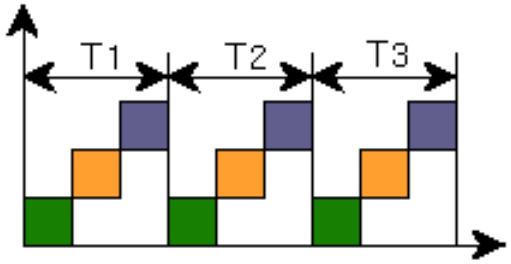
SIMD Pipelining

Pipelinerechner

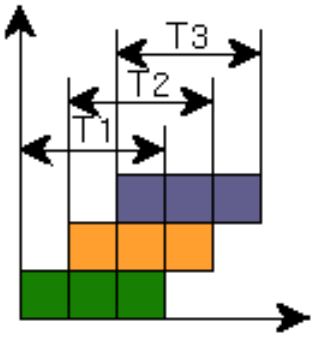




A - load x, y, s
B - multiply x, y
C - add product, s

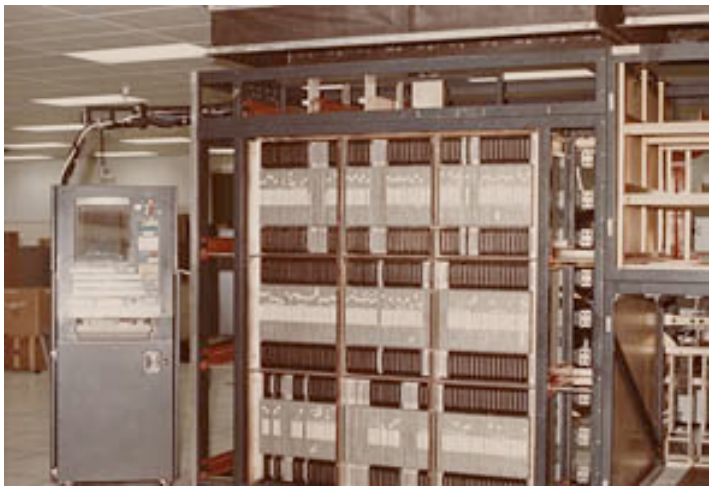


sequentiell



parallel

SIMD Examples



ILLIAC IV (1974)

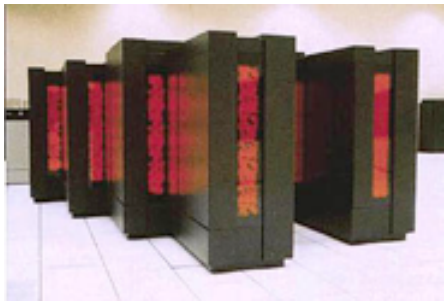
Good for problems with high degree of regularity, such as graphics/image processing

Synchronous (lockstep) and deterministic execution

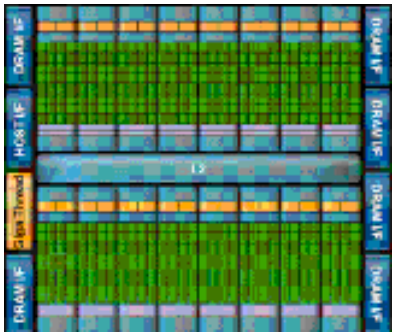
Typically exploit data parallelism

Today: GPGPU Computing, Cell processor, SSE, AltiVec

Cray Y-MP



Thinking Machines CM-2 (1985)



Fermi GPU

Illiac IV

Supercomputer for vector processing from University of Illinois (1966)

One control unit fetches instructions

- Handed over to a set of processing elements (PE‘s)
- Each PE has own memory, accessible by control unit

Intended for 1 GFLOPS, ended up with 100 MFLOPS at the end

Main work on bringing the data to the SIMD machine

- Parallelized versions of FORTRAN language

Credited as fastest machine until 1981

- Computational fluid dynamics (NASA)

(C) Wikipedia

CM2 – Connection Machine

W. Daniel Hillis: The Connection Machine.
1985 (MIT Press Series in Artificial Intelligence)
ISBN 0-262-08157-1

CM2 at Computer Museum, Mountain View, CA

Hersteller:	**Thinking Machines Corporation, Cambridge, Massachusetts**
Prozessoren:	**65.536 PEs (1-Bit Prozessoren)** **Speicher je PE: 128 KB (maximal)** **Peak-Performance: 2.500 MIPS (32-Bit Op.)** **10.000 MFLOPS (Skalar,32Bit)** **5.000 MFLOPS (Skalar,64Bit)**
Verbindungsnetzwerke:	**-globaler Hypercube** **- 4-faches, rekonfigurierbares Nachbarschaftsgitter**
Programmiersprachen:	**-CMLisp (ursprüngliche Variante)** **- *Lisp (Common Lisp Erweiterung)** **-C*(Erweiterung von C)** **-CMFortran (Anlehnung an Fortran 90)** **-C/Paris (C+Assembler Bibliotheksroutinen)**

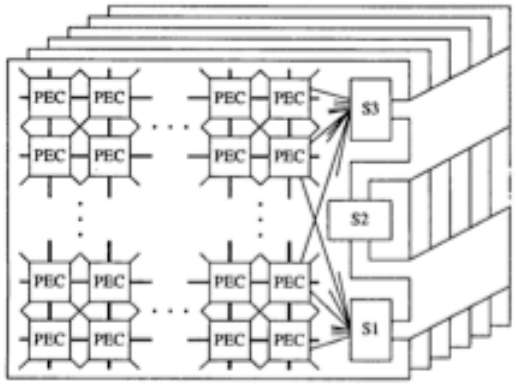
MasPar MP-1

Hersteller:	MasPar Computer Corporation, Sunnyvale, California
Prozessoren:	16.384 PEs (4-Bit Prozessoren) Spei-cher je PE: 64 KB (maximal) Peak-Performance: 30.000 MIPS (32-Bit Op.) 1.500 MFLOPS (32-Bit) 600 MFLOPS (64-Bit)
Verbindungsnetzwerke:	3-stufiger globaler crossbar switch (Router) 8-faches Nachbarschaftsgitter (unabh.)
Programmiersprachen	- MPL (Erweiterung von C) - MPFortran (Anlehnung an Fortran 90

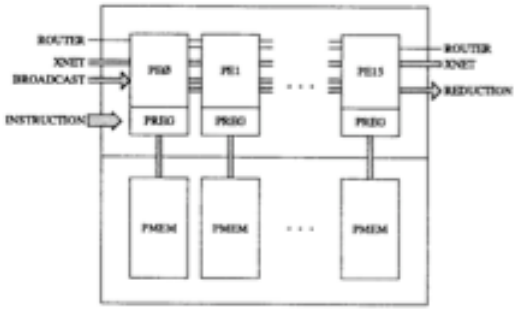
MasPar MP-1 Architecture

Processor Chip contains 32 identical PEs

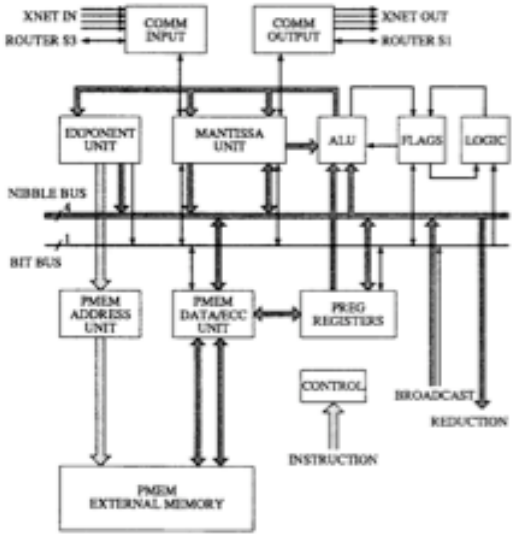
PE is mostly data path logic, no instruction fetch/decode



Interconnection structure



Processor element



Inside a PE

Nickolls, J.R.; MasPar Comput. Corp., Sunnyvale, CA
The design of the MasPar MP-1: a cost effective massively parallel computer
Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Comp. Soc. Intl. Conf..

Distributed Array Processor (DAP 610)

The Distributed Array Processor (DAP) produced by International Computers Limited (ICL) was the world's first commercial massively parallel computer. The original paper study was complete in 1972 and building of the prototype began in 1974.

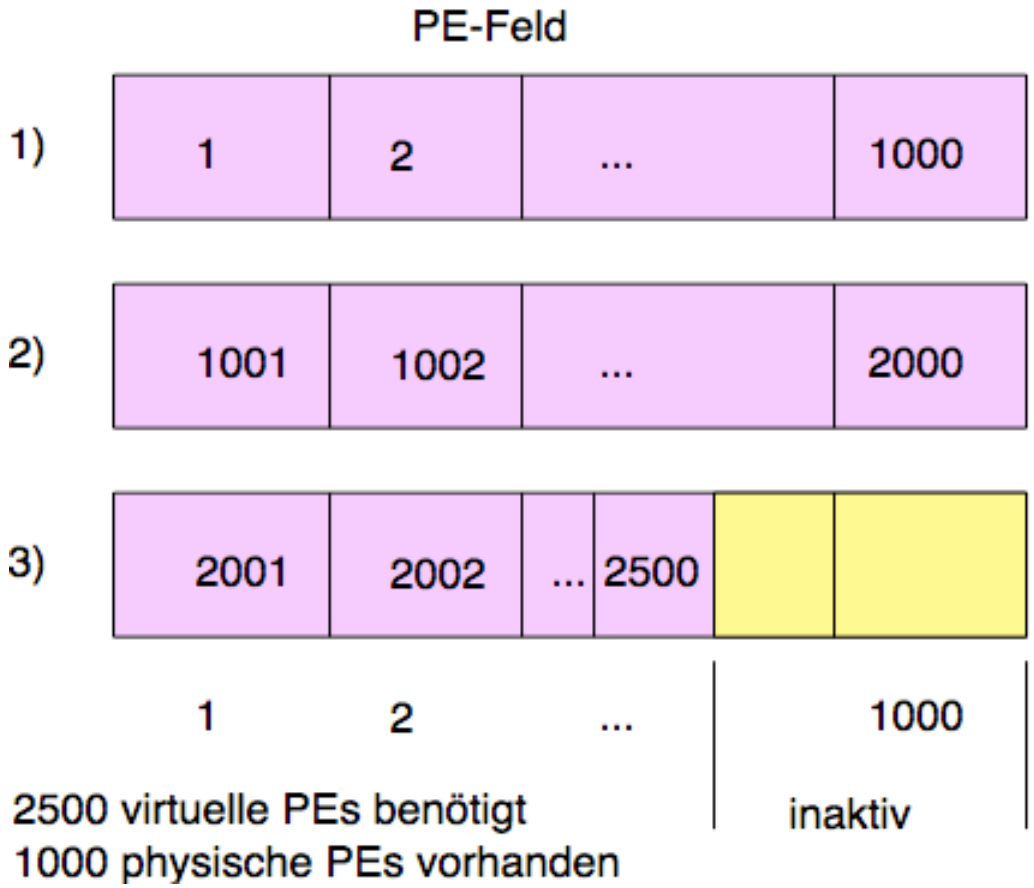
The ICL DAP had 64x64 single bit processing elements (PEs) with 4096 bits of storage per PE. It was attached to an ICL mainframe and could be used as normal memory. (from Wikipedia).

Early mainframe coprocessor...

Hersteller:	**Active Memory Technology (AMT), Reading, England**
Prozessoren:	**4.096 PEs (1-Bit Prozessoren + 8-Bit Koprozessoren)** **Speicher je PE: 32 KB** **Peak-Performance:** **40.000 MIPS (1-Bit Op.)** **20.000 MIPS (8-Bit Op.)** **560 MFLOPS**
Verbindungsnetzwerk:	**- 4-faches Nachbarschaftsgitter** **- (kein globales Netzwerk)**
Programmiersprache:	**- Fortran-Plus (in Anlehnung an Fortran 90)**

Problems with synchronous parallelism: virtual processor elements

Even thousands of PEs may not be sufficient…



SIMD communication – programming is complex

Activation of a group of PEs

Selection of a previously defined connection network

Pair-wise data exchange among active PEs

```
PARALLEL ring[3..8]
    PROPAGATE.rechts(x)
ENDPARALLEL
```

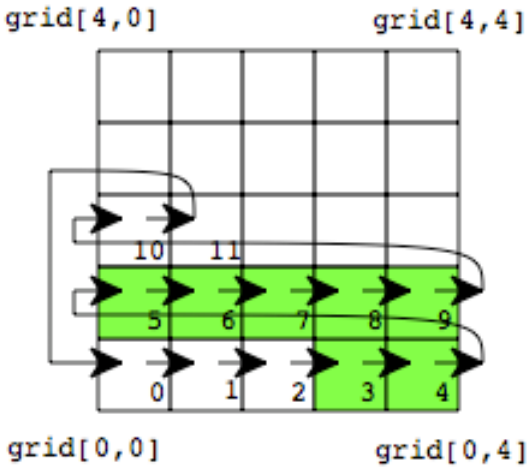
wird abgebildet auf:

a) ein Schritt nach rechts

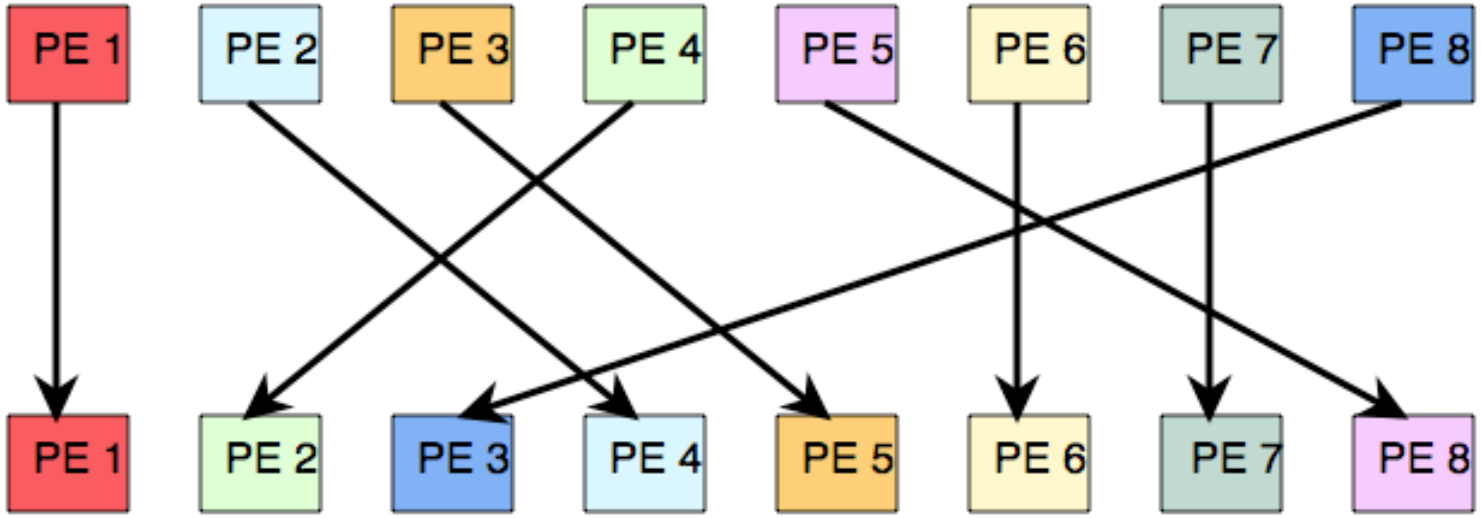
```
PARALLEL
    grid[0..1],[3..4];
    grif[1],   [0..3]
  "grid[i,j] -> grid[i,j+1]"
```

b) eine Zeile höher

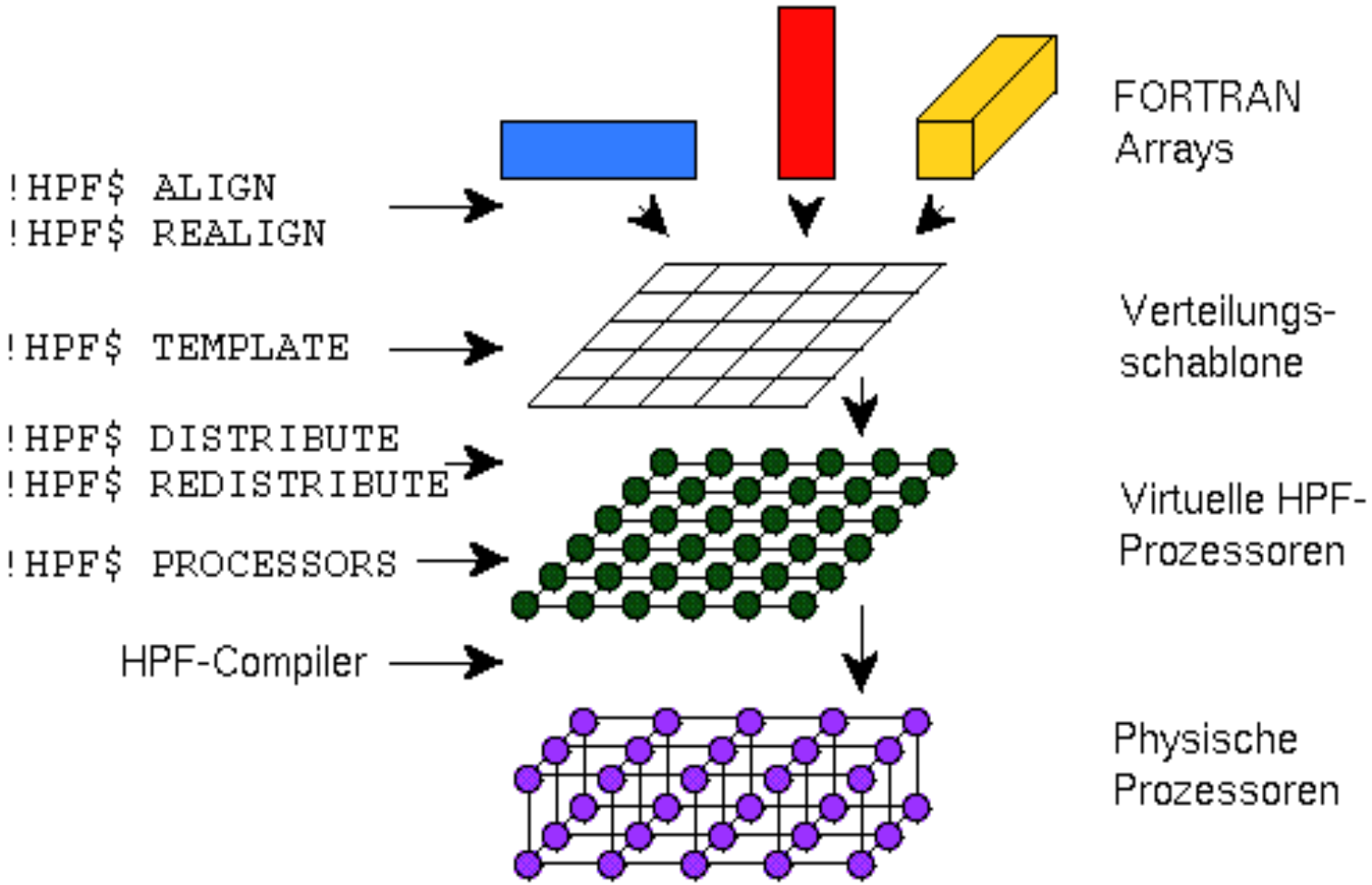
```
PARALLEL
    grid[0],[4]
  "grid[i,j] -> grid[i+1,0]"
```



Permutations – arbitrary data exchange



High Performance Fortran



Data distribution in HPF

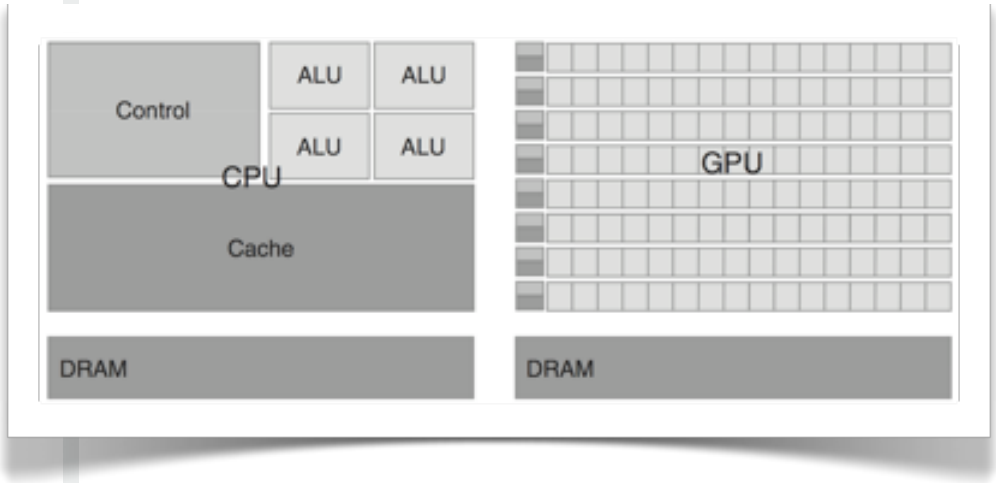
```
!HPF$ PROCESSORS :: prc(5), chess_board(8, 8)
!HPF$ PROCESSORS :: cnfg(-10:10, 5)
!HPF$ PROCESSORS :: mach( NUMBER_OF_PROCESSORS() )
 REAL :: a(1000), b(1000)
 INTEGER :: c(1000, 1000, 1000), d( 1000, 1000, 1000)
!HPF$ DISTRIBUTE (BLOCK) ONTO prc :: a
!HPF$ DISTRIBUTE (CYCLIC) ONTO prc :: b
!HPF$ DISTRIBUTE (BLOCK(100), *, CYCLIC) ONTO cnfg :: c
!HPF$ ALIGN (i,j,k) WITH d(k,j,i) :: c
```

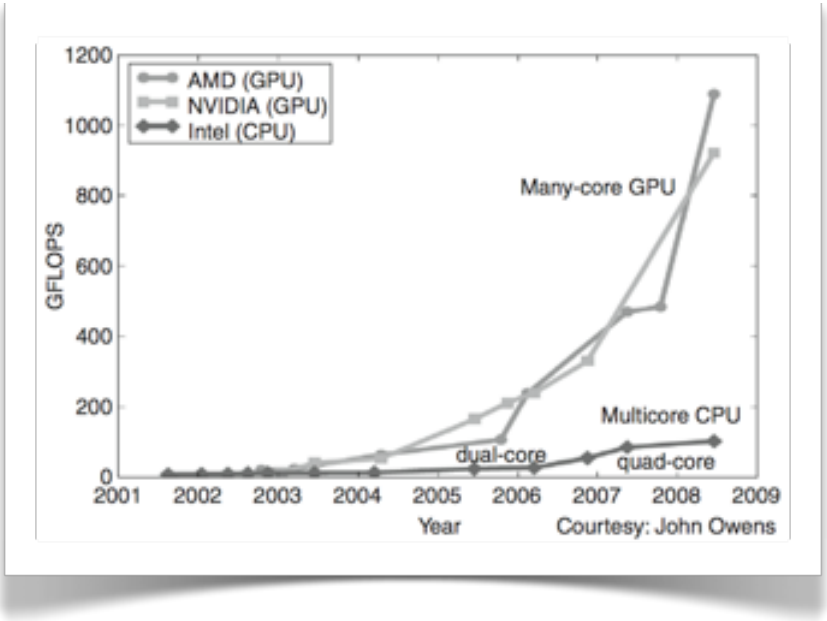
GPGPU Computing – SIMD + multithreading

Pure SIMD approach, different design philosophy

Driven by video / game industry development, recent move towards general purpose computations

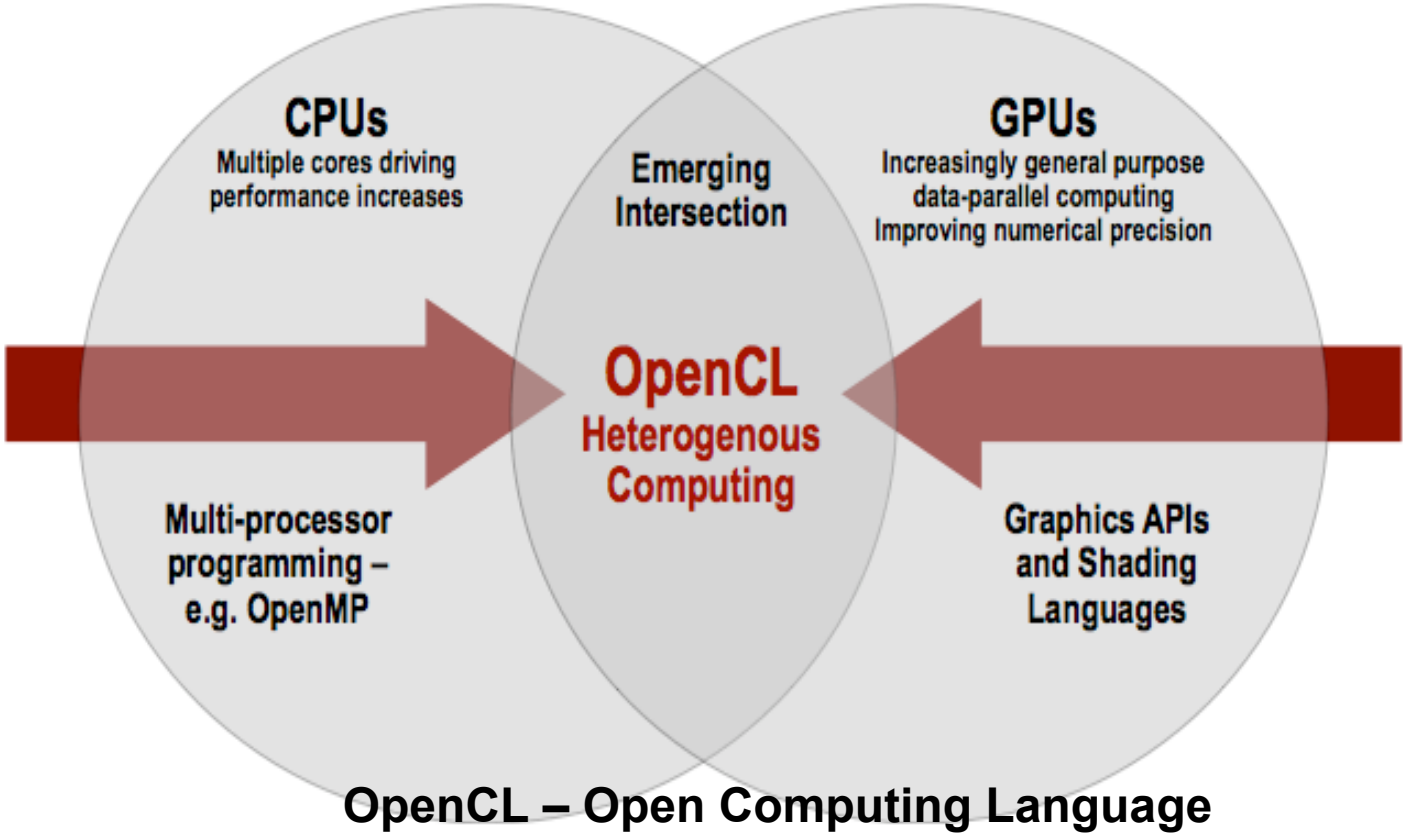
Offloading parallel computation to the GPU is still novel





(C) Kirk & Hwu

Programming Models #1: OpenCL, CUDA



OpenCL – Open Computing Language

CUDA – Compute Unified Device Architecture

Open standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL Design Goals

HPI Hasso Plattner Institut

Use all computational resources in system

- Program GPUs, CPUs, and other processors as peers
- Support both data- and task- parallel compute models

Efficient C-based parallel programming model

- Abstract the specifics of underlying hardware

Abstraction is low-level, high-performance but device-portable

- Approachable – but primarily targeted at expert developers
- Ecosystem foundation – no middleware or “convenience” functions

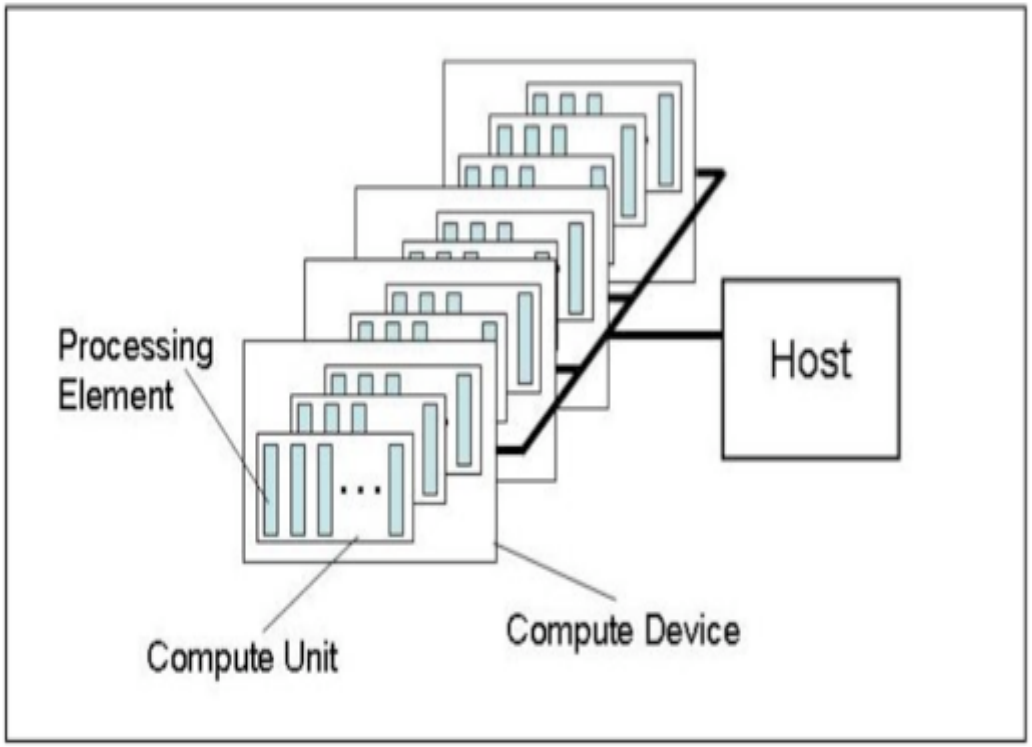
Implementable on a range of embedded, desktop, and server systems

- HPC, desktop, and handheld profiles in one specification

Drive future hardware requirements

- Floating point precision requirements
- Applicable to both consumer and HPC applications

OpenCL Platform Model



One Host + one or more Compute Devices

- Each Compute Device is composed of one or more Compute Units
- Each Compute Unit is further divided into one or more Processing Elements

OpenCL Execution Model

Hasso Plattner Institut

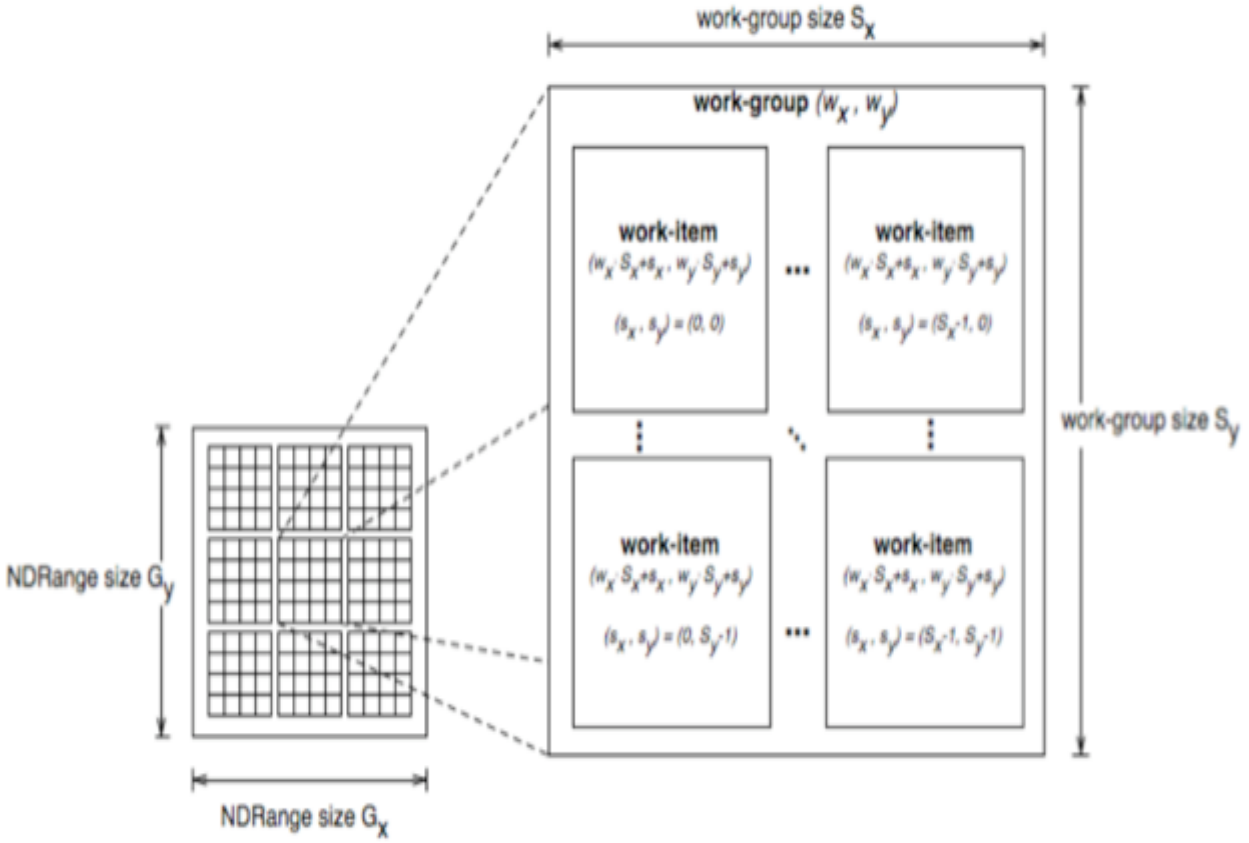
OpenCL Program:

- Kernels
  - Basic unit of executable code — similar to a C function
  - Data-parallel or task-parallel
- Host Program
  - Collection of compute kernels and internal functions
  - Analogous to a dynamic library

Kernel Execution

- The host program invokes a kernel over an index space called an NDRange
  - NDRange = “N-Dimensional Range”
  - NDRange can be a 1, 2, or 3-dimensional space
- A single kernel instance at a point in the index space is called a work-item
  - Work-items have unique global IDs from the index space
- Work-items are further grouped into work-groups
  - Work-groups have a unique work-group ID
  - Work-items have a unique local ID within a work-group

Kernel Execution



Total number of work-items = G_x x G_y

Size of each work-group = S_x x S_y

Global ID can be computed from work-group ID and local ID

Contexts and Queues

Hasso Plattner Institut

Contexts are used to contain and manage the state of the “world”

Kernels are executed in contexts defined and manipulated by the host

- Devices
- Kernels - OpenCL functions
- Program objects - kernel source and executable
- Memory objects

Command-queue - coordinates execution of kernels

- Kernel execution commands
- Memory commands - transfer or mapping of memory object data
- Synchronization commands - constrains the order of commands

Applications queue compute kernel execution instances

- Queued in-order
- Executed in-order or out-of-order
- Events are used to implement appropriate synchronization of execution instances

OpenCL Memory Model

Shared memory model

- Relaxed consistency

Multiple distinct address spaces

- Address spaces can be collapsed depending on the device's memory subsystem

Address spaces

- Private - private to a work-item
- Local - local to a work-group
- Global - accessible by all work-items in all work-groups
- Constant - read only global space

Implementations map this hierarchy

- To available physical memories