MONASH University
Information Technology

FIT3142 Distributed Computing

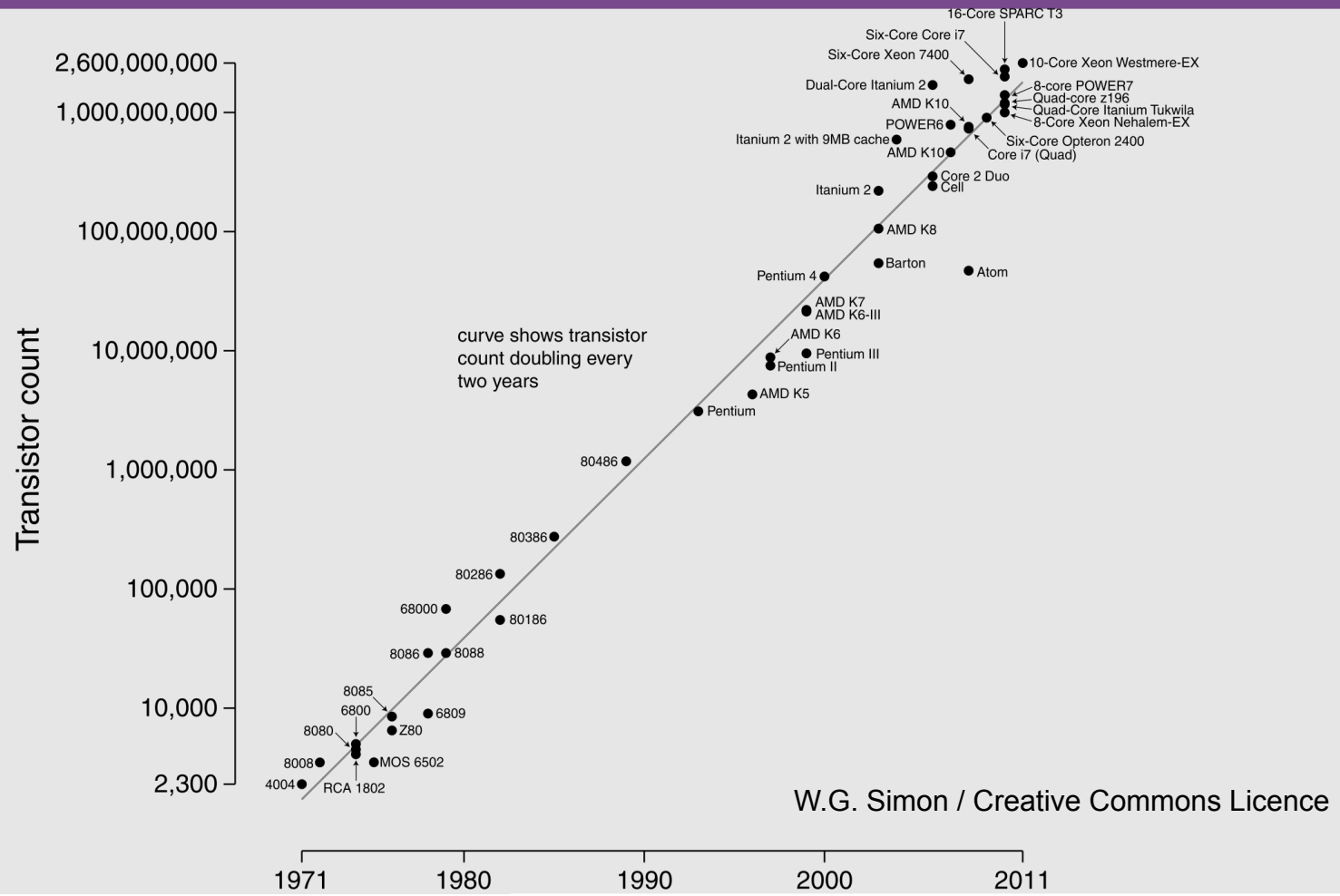
Topic 8 Supplement: GPGPUs vs. Performance

Dr Carlo Kopp
Faculty of Information Technology
Monash University

Why GPGPUs for High Performance Computing?

- GPGPUs are mass produced for consumer applications, mostly gaming but also other demanding uses such as photoediting and multimedia display – therefore *cheap*;
- Moore's Law is a factor in GPGPUs, as it is in CPUs and DRAMs used for main memory in machines;
- Moore's Law describes "exponential growth" in density of integrated circuit chips over time, with a "doubling period" of 18 – 24 months: http://www.csse.monash.edu/~carlo/SYSTEMS/Moores-Law-0700.html; ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf;
- A critical point is that the law applies to chip density – the "corollary" for clock speeds collapsed around 2006 when power density limited further clock speed growth.

Microprocessor Transistor Counts 1971-2011 & Moore's Law



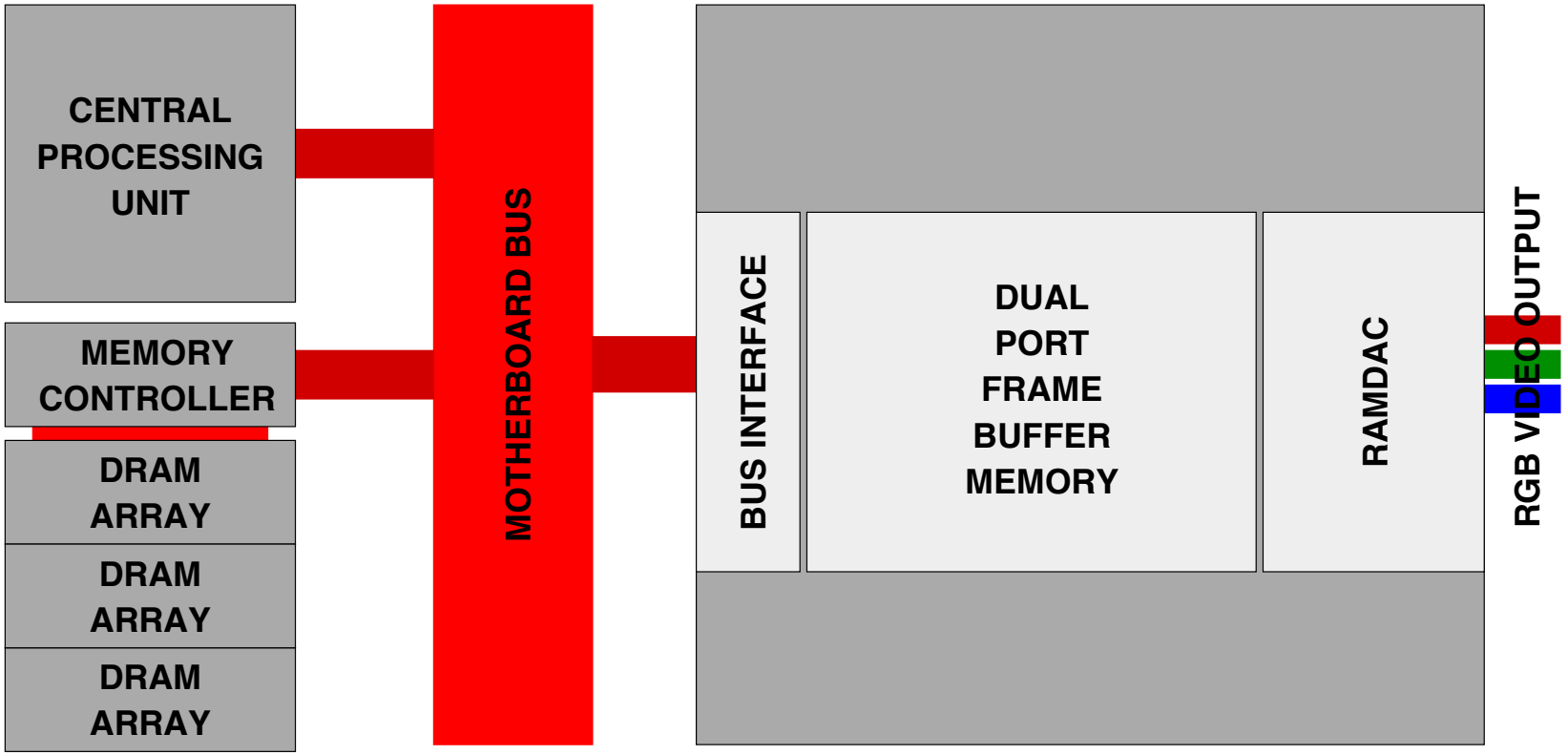
Impacts of Moore's Law

- Transistor counts improve exponentially over time, but CPU clock speeds and main memory speeds do not (any more); increasing numbers of cores in CPUs or GPUs increase the demand for bandwidth to memory for accessing instructions (code), but especially data in many applications;
- In general purpose CPUs, the problem is addressed mainly by increasing the size of the caches which the CPU uses to hold instructions and data and thus minimise memory accesses;
- Graphics applications are data intensive and not always well suited to caching as the results of computations are not flushed periodically to memory, as is the case with cached memory resident operands;
- The result is that GPUs have also employed a different model, which is essentially "data parallel".

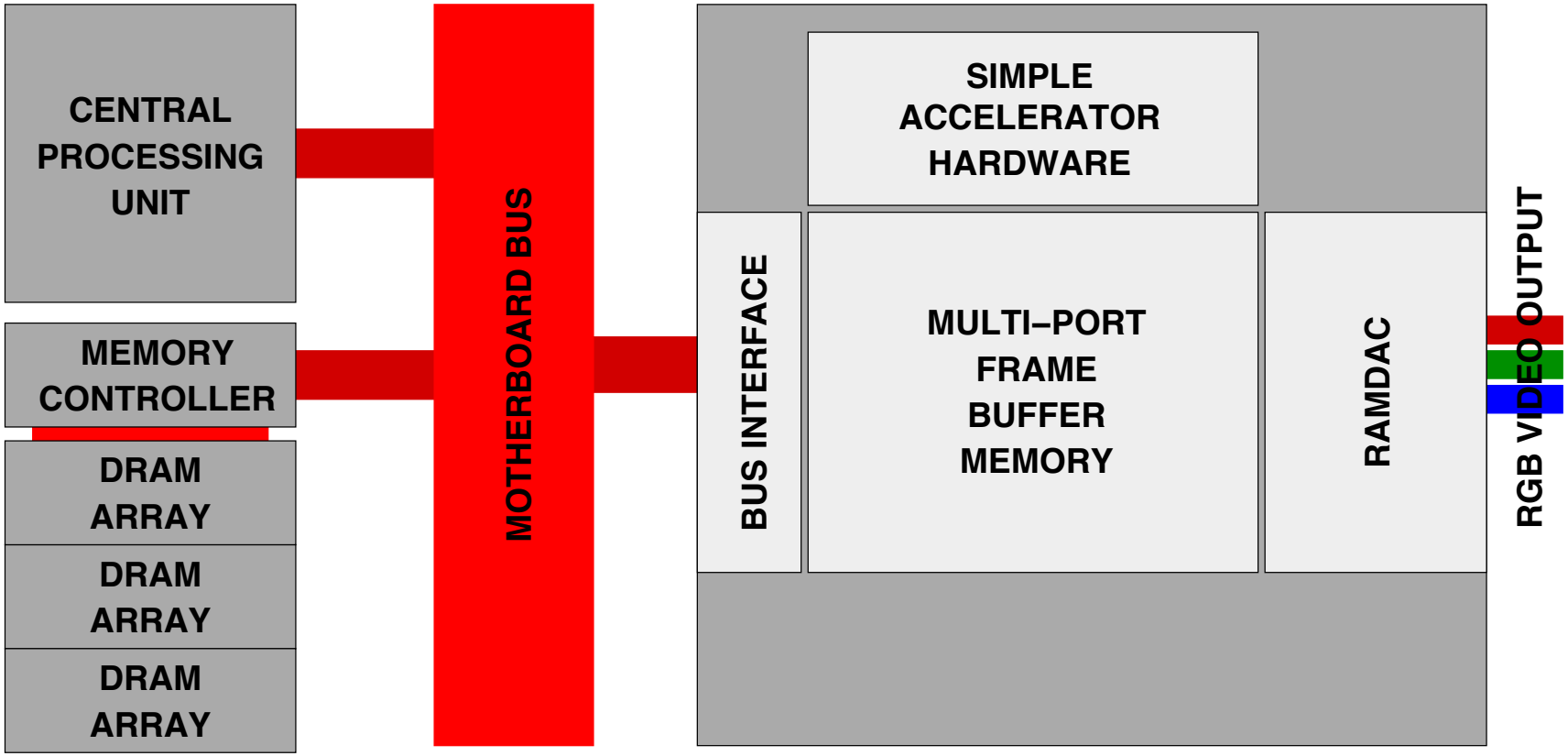
GPU Concepts

- GPGPUs evolved from GPUs, GPUs evolved from “Graphics Accelerators”, which in turn evolved from “Frame Buffers”;
- Frame buffers used a dual port memory which was accessed “upstream” by a CPU which rendered graphical images in the frame buffers, which were pixel-by-pixel read by a RAMDAC chip to generate VGA, SVGA, XGA or proprietary analogue video for display on a Cathode Ray Tube;
- The analogue interface has been replaced by DVI/HDMI and the CRT by LCD, plasma, LCD/LED, LED or OLED displays;
- Early accelerators added dedicated hardware to perform common graphics operations within the frame buffer memory, with the CPU programming the accelerator with specific commands to execute;
- Modern GPGPUs follow the same model, driven by a CPU.

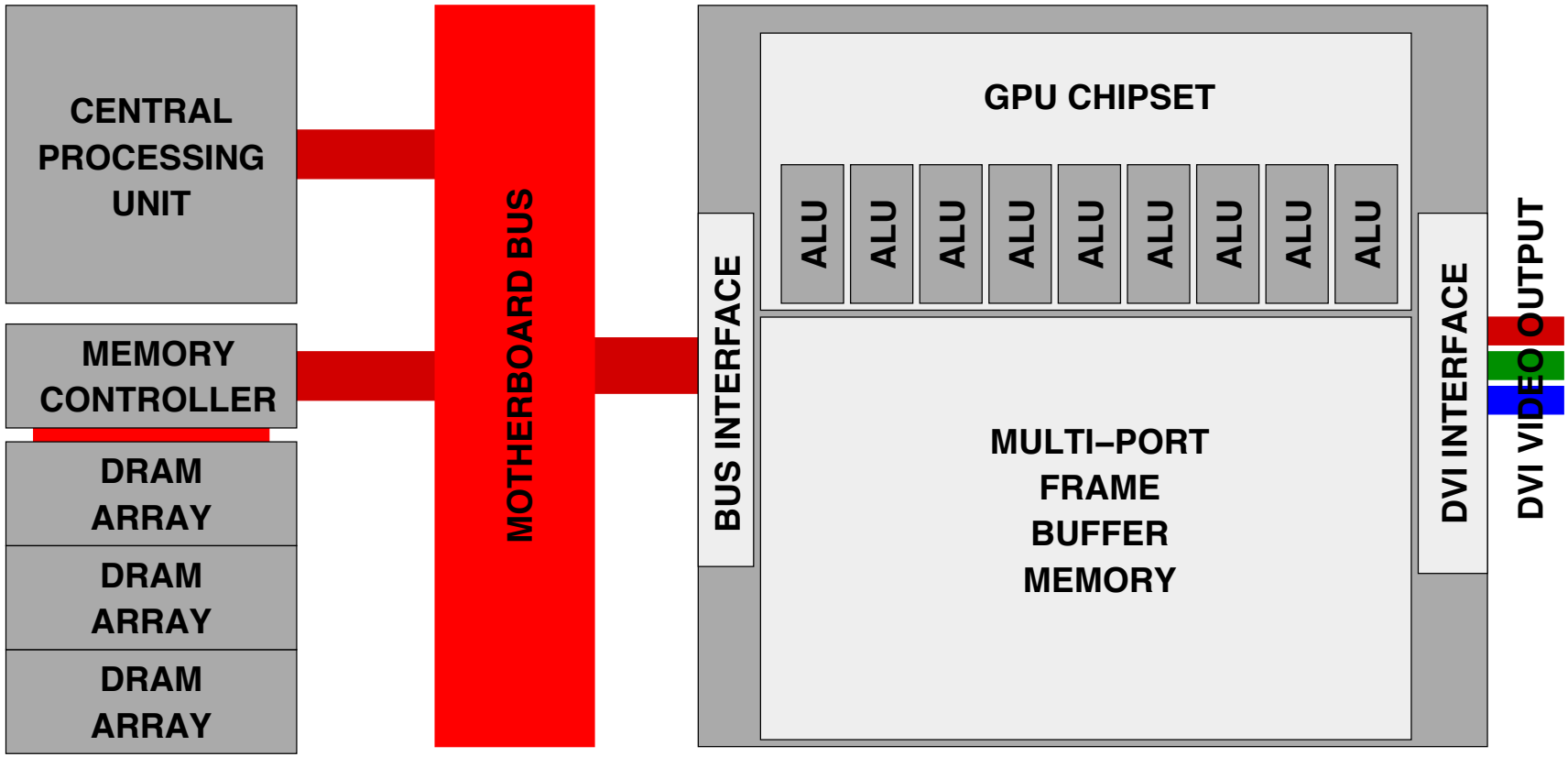
GPGPU Concepts – Frame Buffer



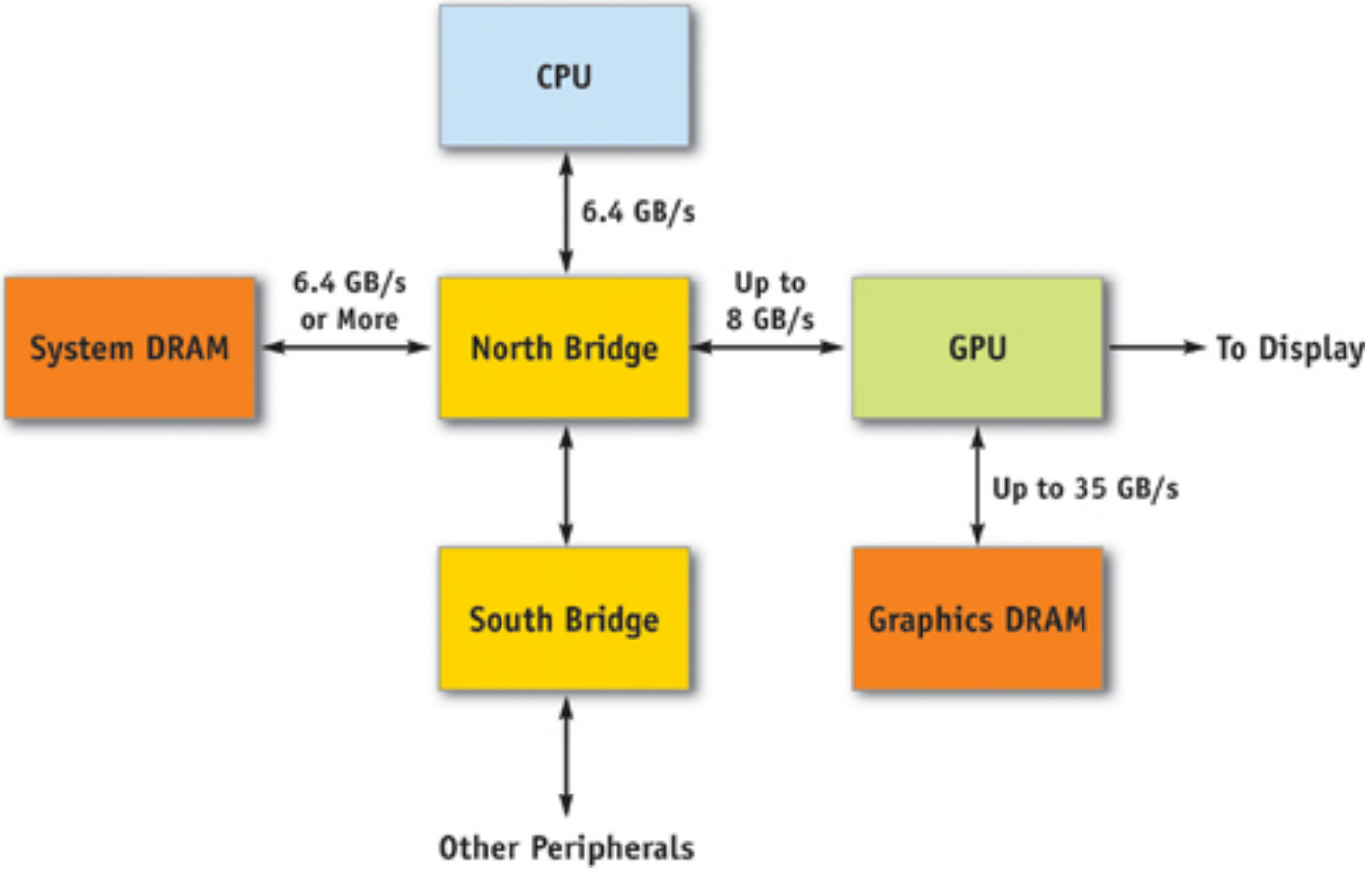
GPGPU Concepts – Accelerated Frame Buffer



GPGPU Concepts – Early GPU



GPGPU Concepts – GPU / CPU Integration (NVIDIA)



GPGPU Concept

- A GPGPU is a direct extension of the "traditional" GPU concept;
- Changes specific to GPGPUs are:
- ✓ programmable pipeline stages;
- ✓ higher precision integer arithmetic units;
- ✓ higher precision floating point arithmetic units;

- A GPGPU is thus able to perform types of computation not unique to graphics processing, such as scientific computations;
- *GPGPUs are still part of a "master-slave" SIMD model, where a general purpose CPU sets up the GPGPU cores to perform their computations.*

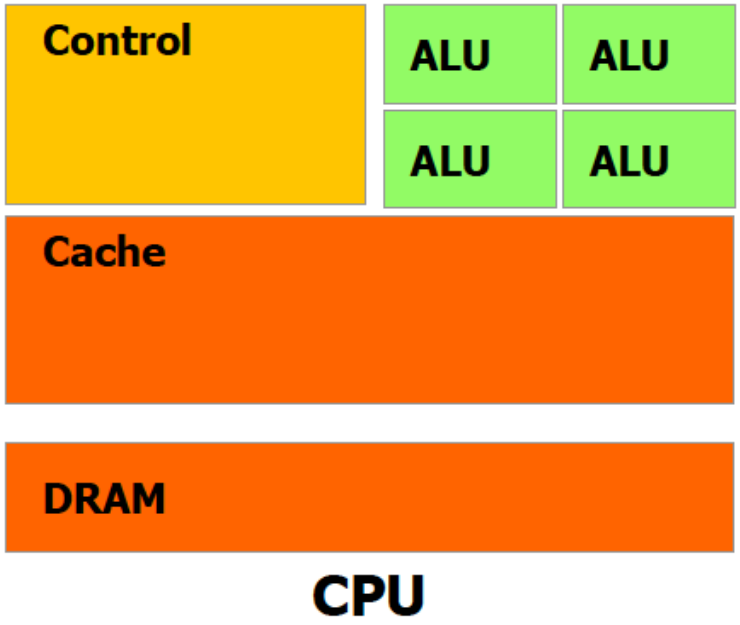
GPGPU Memory Bandwidth

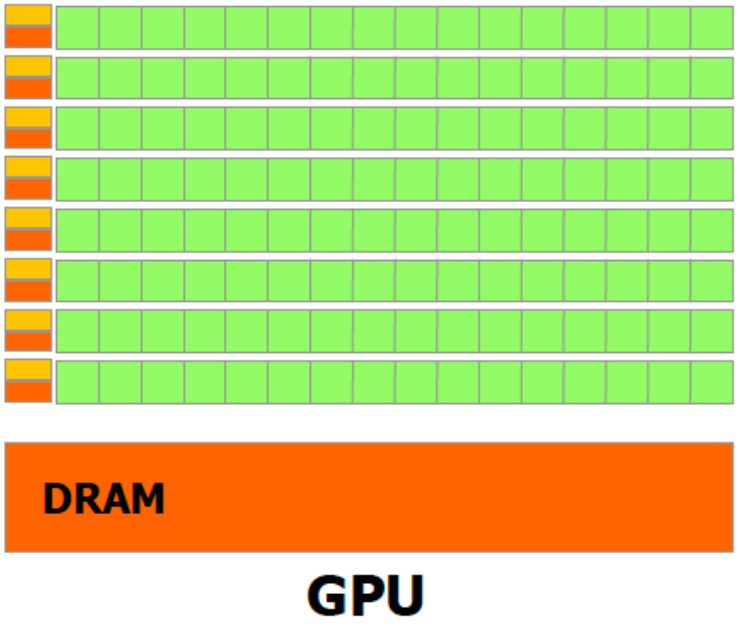
- **A major performance limitation in current processors is the disparity in bandwidth to main memory versus bandwidth to much smaller internal register storage;**
- **In the time taken to read/write main memory, dozens of instructions can be executed – providing the operands are in registers or cache;**
- **GPUs employ dedicated very high speed memory arrays for graphics computations;**
- **A conventional main memory system will have a memory bandwidth typically of Gigabytes/sec up to tens of Gigabytes/ sec (AMD HT/HTX, Intel QPI) – currently ~25 GB/s;**
- ***A GPU/GPGPU internal memory subsystem will have a memory bandwidth of more than 100 Gigabytes/sec (NVIDIA Tesla @ 148 GB/s; AMD Radeon 5870 @ 153.6 GB/sec) – sixfold better.***

GPGPU/GPU versus CPU

Processor	GPGPU/GPU	CPU
Number of Cores	Hundreds	2 - 12
Core complexity	Low	High
Memory bandwidth	~6:1	1:1
Vector operations	Yes	Limited or none
ILP Exploitation	Software scheduling	Control unit
Optimisation	Graphics/Scientific	General purpose
Integration	Master/Slave	Symmetrical MP
Architecture	Data Parallel / Stream	Conventional
Software	CUDA / API	Any

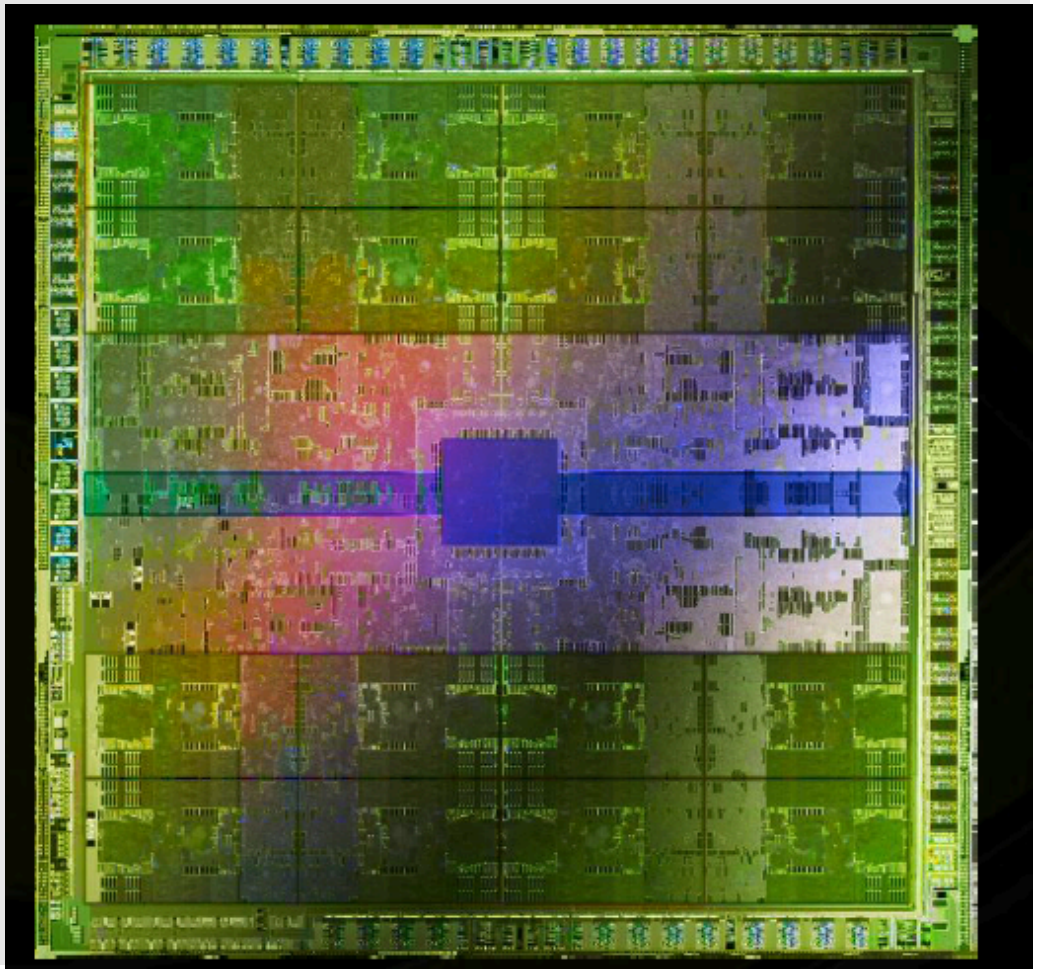
GPGPU/GPU versus CPU (NVIDIA)





Example: NVIDIA Fermi GPGPU Die

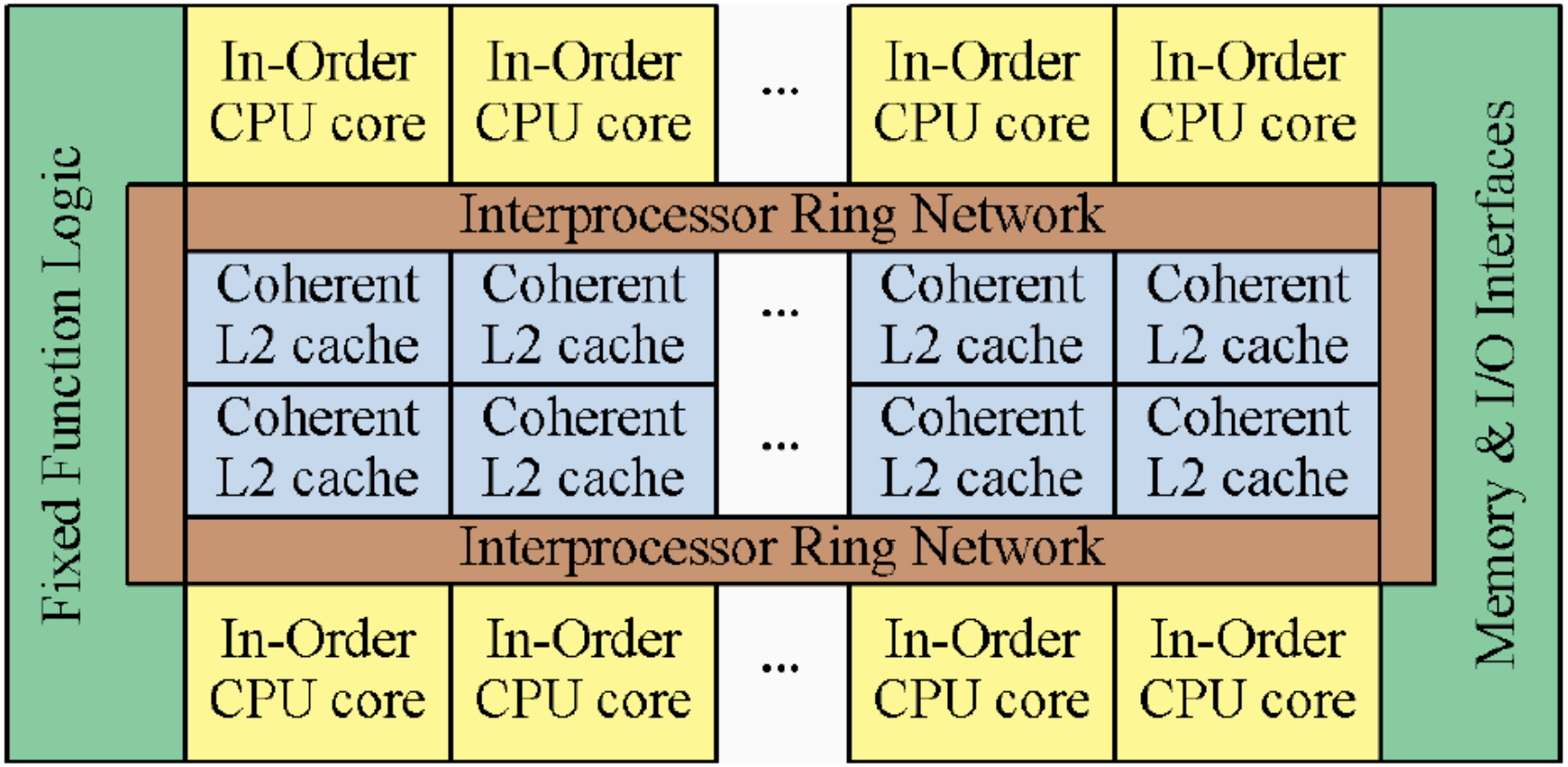
- *Fermi*
- 3 billion transistors
- 512 x CUDA cores
- ~2 x the memory bandwidth cf earlier chip
- L1 and L2 caches
- 8x the peak fp64 performance cf earlier chip
- Error Correcting Code Memory
- C++ API



Example: Intel Larrabee (Seiler 2008)

- The "conventional" GPGPU model is not the only feasible approach for a highly parallel graphics processing architecture;
- In 2007 – 2008 Intel prototyped the Larrabee chip, with eight cores, each based on the x86 general purpose architecture, but with each core extended to perform vector operations;
- In practical terms this is a multicore general purpose processor optimised to perform graphics or multimedia operations;
- The advantage of the Larrabee model is that each core can perform general computation independently and is not part of an asymmetric "master-slave" model used with GPUs and GPGPUs – the disadvantage is performance loss for many problems where the narrowly optimised GPGPU does better.

Example: Intel Larrabee Core



Example: Intel Larrabee Core

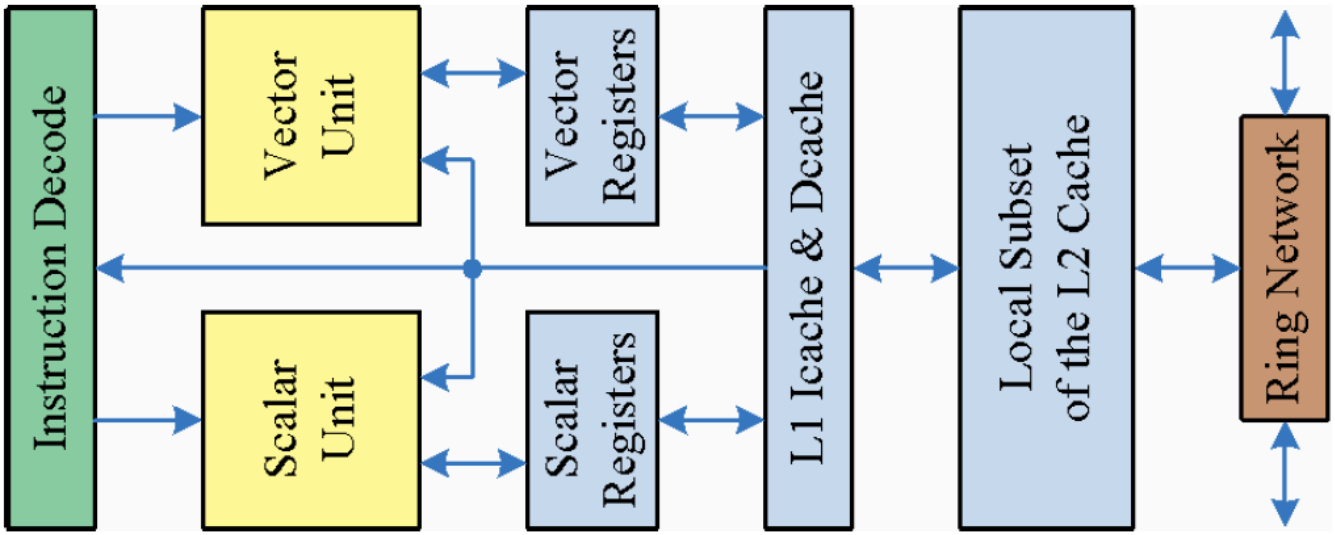


***Figure** 3: Larrabee CPU core and associated system blocks: the CPU is derived from the Pentium processor in-order design, plus 64-bit instructions, multi-threading and a wide VPU. Each core has fast access to its 256KB local subset of a coherent 2nd level cache. L1 cache sizes are 32KB for Icache and 32KB for Dcache. Ring network accesses pass through the L2 cache for coherency.*

Example: Intel Larrabee-like Core Comparison

# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

GPGPU Stream Processing Model

- Simplifies parallel software and hardware by restricting the parallel computation that can be performed;
- A data set constitutes a (data) stream – common in multimedia and graphics applications;
- A “graphics kernel” function is a series of operations;
- Commonly defined as a series of nested loops (with no data specification).

Stream Processing Model

- The kernel function is applied to each element in the stream;
- *Uniform streaming*, where the same kernel function is applied to all elements in the stream, is commonly used;
- Kernel functions are usually *pipelined*, and local on-chip or on card fast memory is reused to minimize external memory bandwidth;
- ***Since the kernel and stream abstractions expose data dependencies, compiler tools must fully automate and optimize on-chip management tasks.***

Data Dependencies and Parallelism

- Two important stream data characteristics (as required by a kernel):
- ✓ Independent
- ✓ Local
- Kernel operations define the basic data unit, both as input and output.
- This allows the hardware to better allocate resources and schedule global I/O.

Data Dependencies and Parallelism

- Definition of the data unit is usually explicit in the kernel, which is expected to have well-defined (e.g. structures) inputs and outputs
- Well defined and independent compute blocks allow scheduling of bulk I/O operations.
- Exploit the underlying hardware performance of cache and specially GPGPU memory bus.
- Kernel locality: Values associated within a single kernel invocation are made local and use the fast local GPGPU memory.

Stream Programming Languages

- **Generally the following type of applications will benefit from stream computing:**

✓ ***Compute Intensive:*** **where the number of arithmetic operations is high for each I/O or global memory access operation;**

✓ ***Data Parallel:*** **where the same kernel function can be applied repeatedly to records in an input stream and a number of records can be processed simultaneously without waiting for results from previous records (i.e. low data dependencies);**

✓ ***High Data Locality:*** **where data is produced once, read once or twice later in the application, and never read again. Intermediate streams passed between kernels as well as intermediate data within kernel functions can capture this locality directly using the stream processing programming model.**

Notable Stream Programming Languages

Open Standards:

- ✓ **OpenCL (Open Computing Language) from Apple, available Mac OS X 10.6 and Linux;**
- ✓ **ACOTES based on OpenMP;**
- ✓ **OpenACC from Cray Computer, NVIDIA,Portland Group, CAPS – language extension via directives;**

- **Vendor Proprietary Schemes:**

- ✓ **BROOK+ from AMD/ATI;**
- ✓ **Compute Unified Device Architecture (CUDA) from Nvidia. Only available for NVIDIA GPUs;**

Open CL

- Open specifications
- ✓ Designed for number crunching
- ✓ Parallel computing tasks
- ✓ Vendors specific implementations, must be complaint to the standard
- Proposed by Apple (include Nvidia and Intel);
- Specifications developed by a number of companies (wider following);
- Specifications maintained by Khronos Group, also provide license fee free contents (KG also maintain the OpenGL specification)

Why Open CL?

- It is portable across implementations;
- Hardware heterogeneity, CPU, GPU support
- OpenCL Programming interface is designed to provide a good framework to utilise underlying (parallel) hardware;
- Applications:

✓ Image, Video, Audio processing

✓ Gaming, Scientific calculations, Simulations

✓ Financial Modelling

✓ Computationally intensive data-parallel applications

OpenACC – Compiler Directive Extensions

- Open parallel programming standard designed to enable the scientific and technical Fortran and C applications to use heterogeneous CPU/GPU computing systems;
- OpenACC allows parallel programmers to provide simple hints, known as “directives” to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself (NVIDIA cite);
- Directives provide a common multi-platform and multi-vendor compatible scheme which allows existing applications to be ported to GPUs with minimal effort;
- The intend behind OpenACC is therefore to provide a very low cost migration path to GPU hardware for existing code written in languages such as Fortran or C++;
- *OpenACC relies on programmers understanding their code.*

OpenACC Example: Calculating Pi (NVIDIA)

Fortran Version

```
program picalc
    implicit none
    integer, parameter ::
n=1000000
    integer :: i
    real(kind=8) :: t, pi
    pi = 0.0
 !$acc parallel loop
    do i=0, n-1
       t = (i+0.5)/n
       pi = pi + 4.0/(1.0 + t*t)
    end do
 !$acc end parallel loop
    print *, 'pi=', pi/n
end program picalc
```

C Version

```
#include <stdio.h>
#define N 1000000

int main(void) {
double pi = 0.0f; long i;
#pragma acc parallel loop
for (i=0; i<N; i++) {
   double t= (double)((i+0.5)/N);
   pi +=4.0/(1.0+t*t);
}
printf("pi=%16.15f\n",pi/N);
return 0;
}
```

NVIDIA CUDA (NVIDIA Website)

- Released by NVIDIA in 2006 - principal author Ian Buck who previously developed the BROOK language;
- NVIDIA Performance Primitives (NPP) library includes over 2200 GPU-accelerated functions for image & signal processing Arithmetic, Logic, Conversions, Filters, Statistics, etc.
- Platforms: MS Windows 32/64; Linux 32/64; Mac OSX 32/64;
- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS Linear Algebra Library
- cuSPARSE – Sparse Matrix Library
- cuRAND – Random Number Generation (RNG) Library
- NPP – Performance Primitives for Image & Video Processing
- Thrust – Templated Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

NVIDIA CUDA (CUDA C Programming Guide)

- **CUDA C extends C by allowing the programmer to define C functions, called *kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.***
- **A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<…>>> *execution configuration syntax.***
- ***Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable;***
- **As an illustration, the following sample code adds two vectors *A and B of size N and stores the result into vector C:***

NVIDIA CUDA (CUDA C Programming Guide)

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
...
// Kernel invocation with N threads
VecAdd<<<1, N>>>(A, B, C);
...
}
```

// each of the N threads that execute VecAdd() performs one pair-wise addition.

NVIDIA CUDA (CUDA C Programming Guide)

- The CUDA programming model assumes that the CUDA threads execute on a physically separate *device that operates as a coprocessor to the host running the C program.*
- *This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.*
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as *host memory and device memory, respectively.*
- *Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime.*
- *This includes device memory allocation and deallocation as well as data transfer between host and device memory.*

Limitations in GPGPU Integration

- Most adaptations for GPGPU programming *expose* the GPGPU hardware to the programmer;
- This presents two problems:
- ✓ Programmers often do not understand hardware performance well and make mistakes, causing bugs or performance loss;
- ✓ Application portability can be compromised, as changing GPGPU hardware vendors and/or APIs requires deep alterations to the code;
- Traditional "vectorising" Fortran compilers would *hide* the hardware from the programmer to minimise the required programmer skills and maximise application portability;
- ***Performance gains from using GPGPUs can be excellent, but rely upon the programmer understanding the GPGPU design and GPGPU API library calls.***

Problems in GPGPU Integration

- Scalability for GPGPU hosted applications is excellent, where the number of cores $N <= N_{GPGPU}$ i.e. the number of cores on the GPGPU chip;
- This is because of the very high memory bandwidth to the private memory attached to the GPGPU chip;
- If the required number of cores is higher, memory bandwidth between the GPGPU modules is then much lower than memory bandwidth to the private memory attached to the GPGPU chip;
- GPGPU solutions are therefore very sensitive to the data parallelism in the problem;
- In systems with multiple GPGPUs, the main bus connecting the global system main memory and GPGPUs becomes a critical system performance bottleneck.

Summary

- Moore's Law
- GPU Concepts / Evolution
- GPGPU Concepts
- Memory bandwidth considerations;
- GPGPUs versus Multicore CPUs;
- Stream Processing in GPGPUs;
- Data dependencies and parallelism;
- Stream programming languages; OpenCL, ACOTES, OpenACC, BROOK+, CUDA;
- Limitations in GPGPU Integration;
- Problems in GPGPU Integration.

References/Reading/Viewing

- Owens, J., *Ch.29 Streaming Architectures and Technology Trends*, GPU Gems 2: Part IV - General-Purpose Computation on GPUS: A Primer, NVIDIA Corp, URI http://developer.nvidia.com/book/export/html/48
- Seiler, L. et al, Larrabee: A Many-Core x86 Architecture for Visual Computing, ACM Transactions on Graphics, Vol. 27, No. 3, Article 18, August 2008.
- NVIDIA CUDA™, NVIDIA CUDA C Programming Guide, 4/16/2012;
- NVIDIA: Tesla Personal Supercomputer; URI: http://www.youtube.com/watch?v=I8FUmS1h-5U
- TYAN B7015 -- 8-GPU Nvidia Tesla Supercomputer Server, URI: http://youtu.be/VML2cXDe32M

END FIT3143