Unified memory

GPGPU 2015: High Performance Computing with CUDA

University of Cape Town (South Africa), April, 20th-24th, 2015

Manuel Ujaldón

Associate Professor @ Univ. of Malaga (Spain)

Conjoint Senior Lecturer @ Univ. of Newcastle (Australia)

CUDA Fellow @ Nvidia

NVIDIA

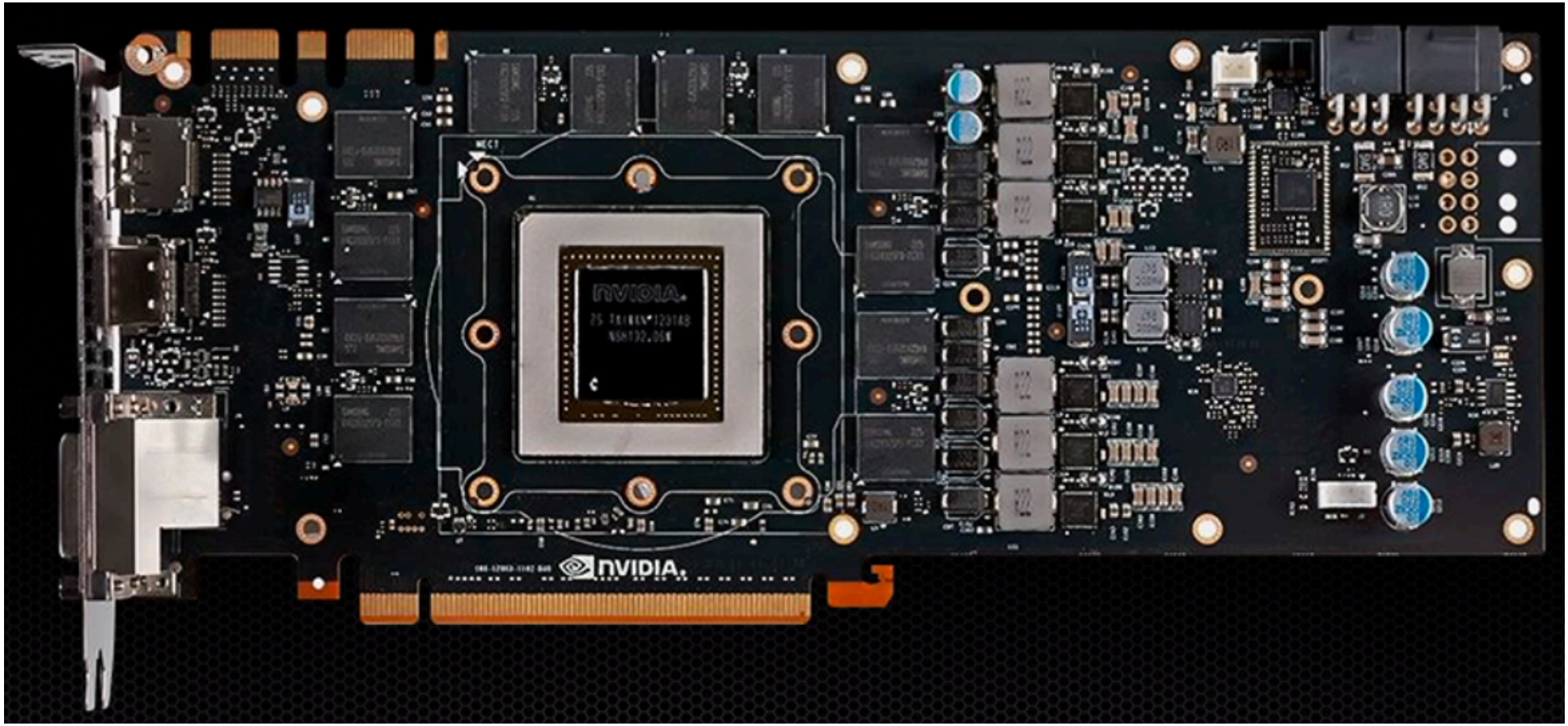
Talk outline [28 slides]

1. State of art of technology [12]
2. Programming with unified memory [4]
3. Examples [8]
4. Final remarks [4]

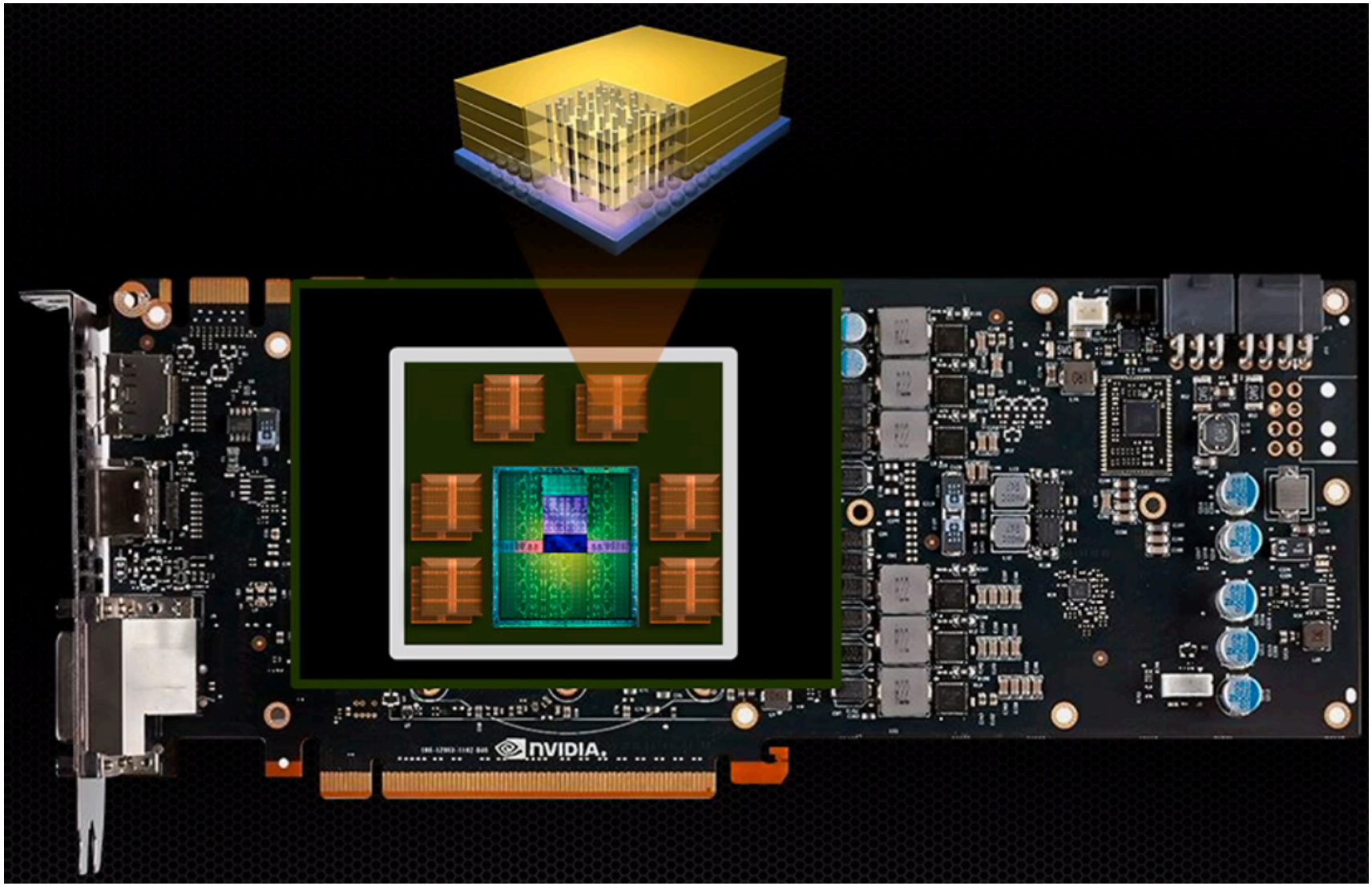
I. State of art of technology

NVIDIA

A 2015 graphics card:
Kepler/Maxwell GPU with GDDR5 memory

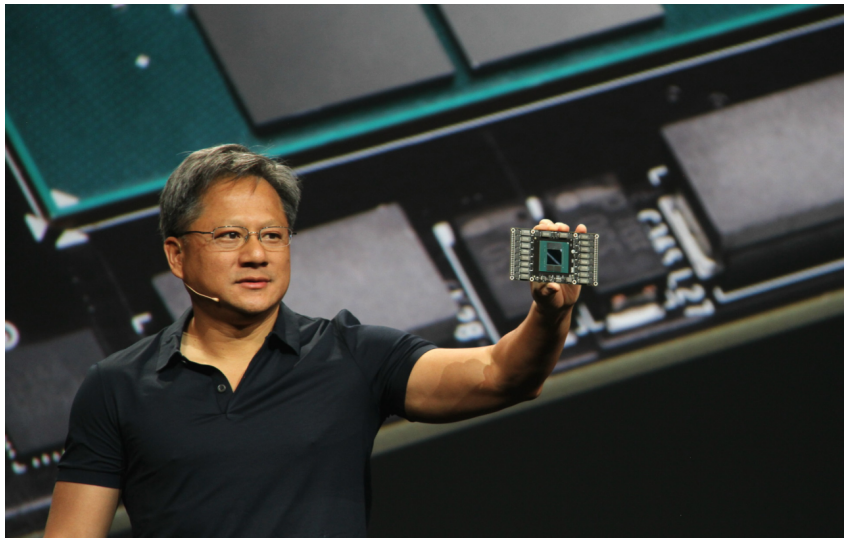


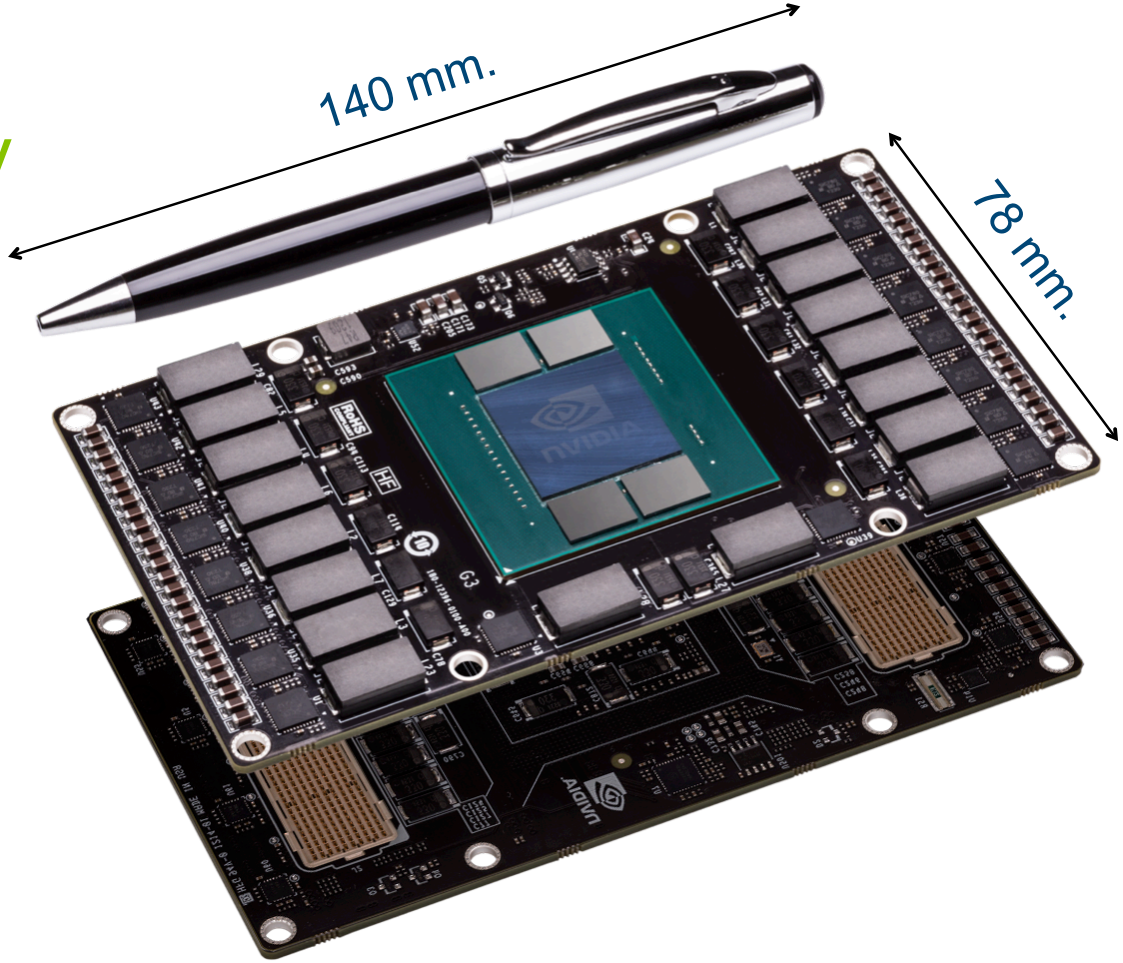
A 2017 graphics card: Pascal GPU with 3D memory (stacked DRAM)



The Pascal GPU prototype: SXM2.0 Form Factor

SMX2.0*:
3x Performance Density

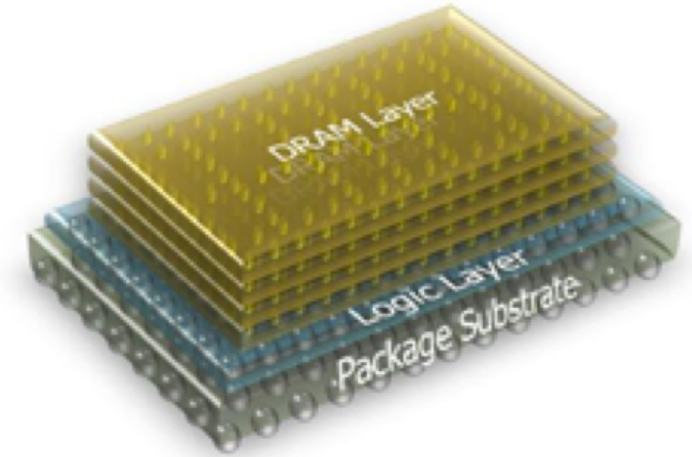


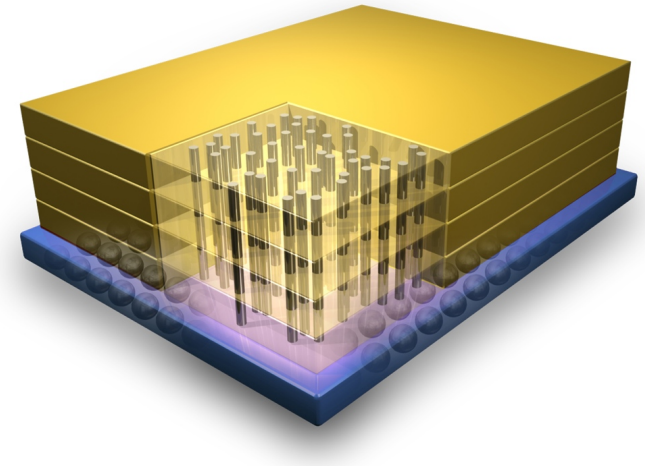


(* Marketing Code Name. Name is not final).

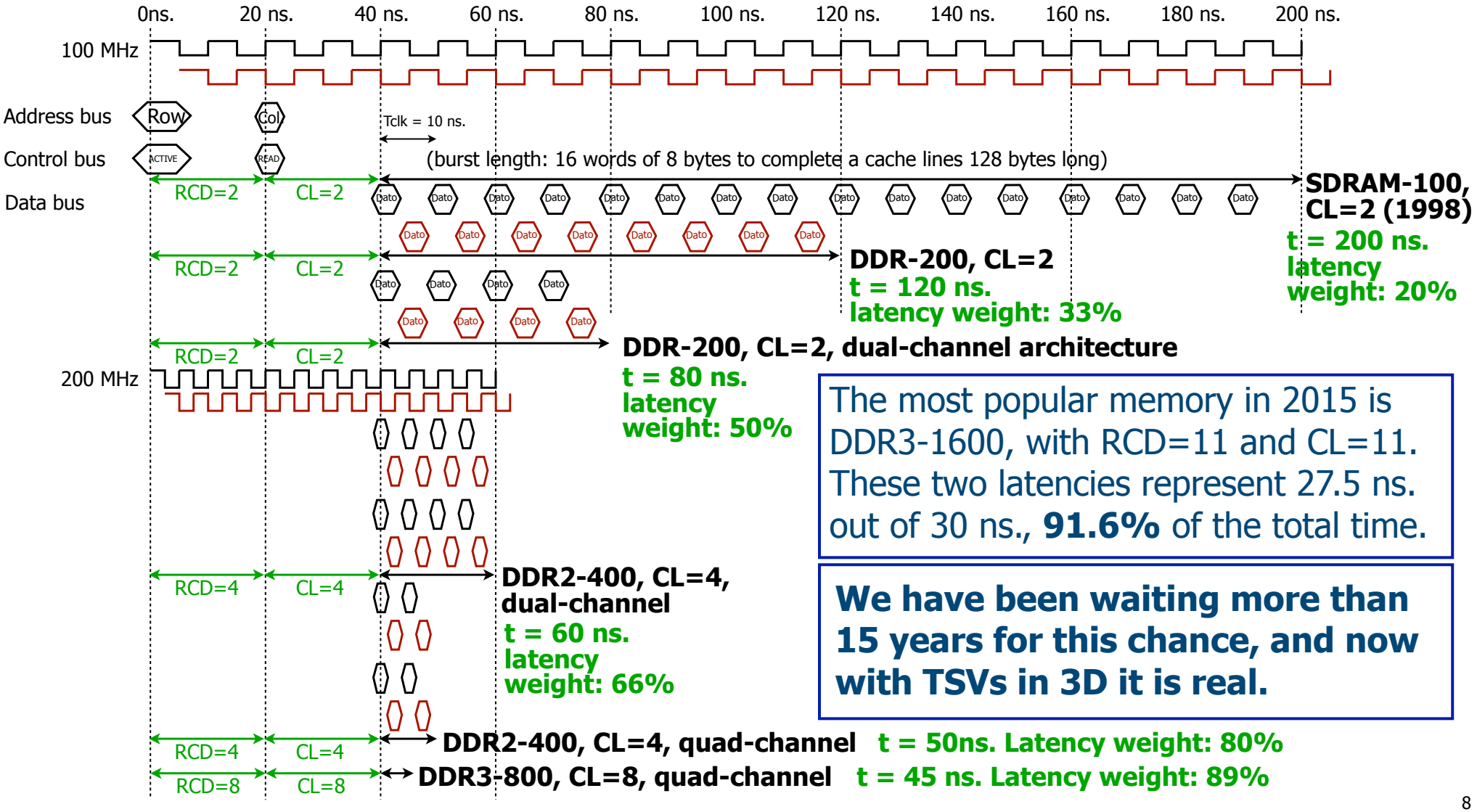
Details on silicon integration

- DRAM cells are organized in **vaults**, which take borrowed the interleaved memory arrays from already existing DRAM chips.
- A logic controller is placed at the base of the DRAM **layers**, with data matrices on top.
- The assembly is connected with through-silicon vias, **TSVs**, which traverse vertically the stack using pitches between 4 and 50 um. with a vertical latency of 12 picosecs. for a Stacked DRAM endowed with 20 layers.





Time to fill a typical cache line (128 bytes)



The most popular memory in 2015 is DDR3-1600, with RCD=11 and CL=11. These two latencies represent 27.5 ns. out of 30 ns., **91.6%** of the total time.

We have been waiting more than 15 years for this chance, and now with TSVs in 3D it is real.

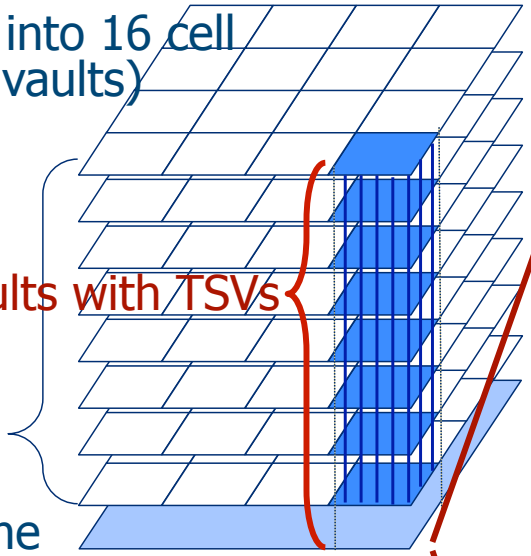
3D integration, side by side with the processor

Step 1: Partition into 16 cell matrices (future vaults)

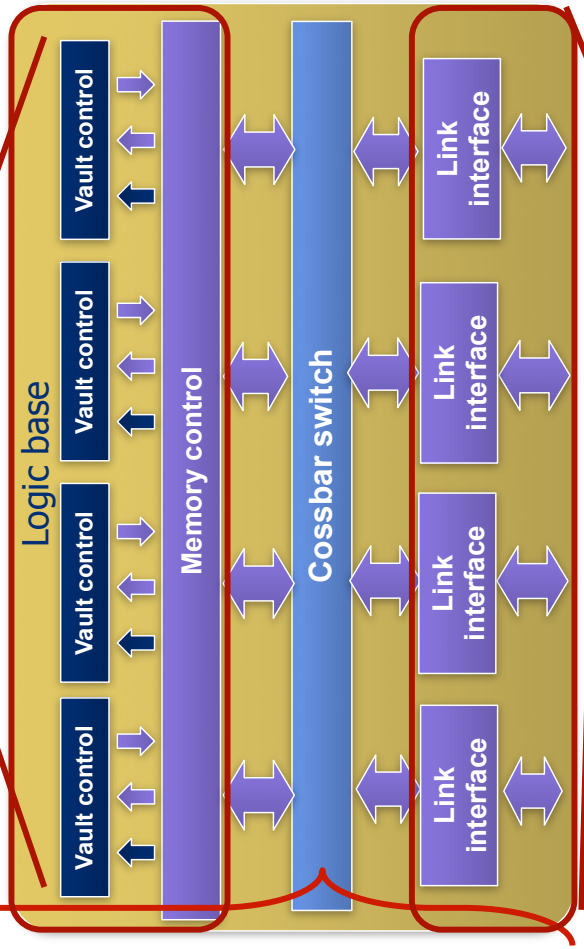
Step 4: Build vaults with TSVs

Step 3: Pile-up DRAM layers.

Step 2: Gather the common logic underneath.



Step 5: Buses connecting 3D memory chips and the processor are incorporated.

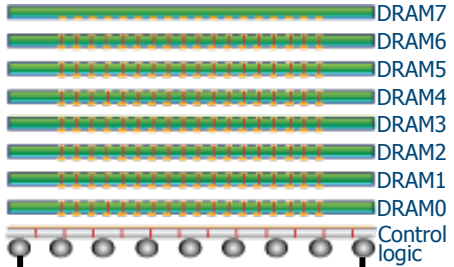


Links to processor(s), which can be another 3D chip, but more heterogeneous:
- Base: CPU y GPU.
- Layers: Cache (SRAM).

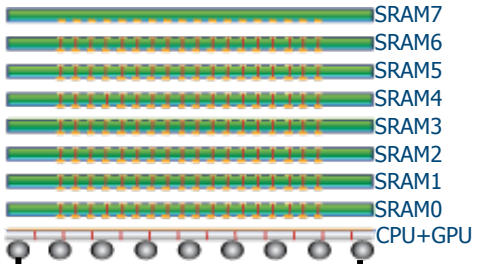
A typical multi-core die uses >50% for SRAM. And those transistors switch slower on lower voltage, so the cache will rely on interleaving over piled-up matrices, just the way DRAM does.

3D technology for DRAM memory

Typical DRAM chips use 74% of the silicon area for the cell matrices.



3D technology for processor(s)



Using 3D chips to build a Haswell-like CPU

- We have CPU, GPU and SRAM in different proportions within silicon die, depending on 8 available models:

- And, in addition, we want to include some DRAM layers.

Intel already authored a research showing the best choices (*)

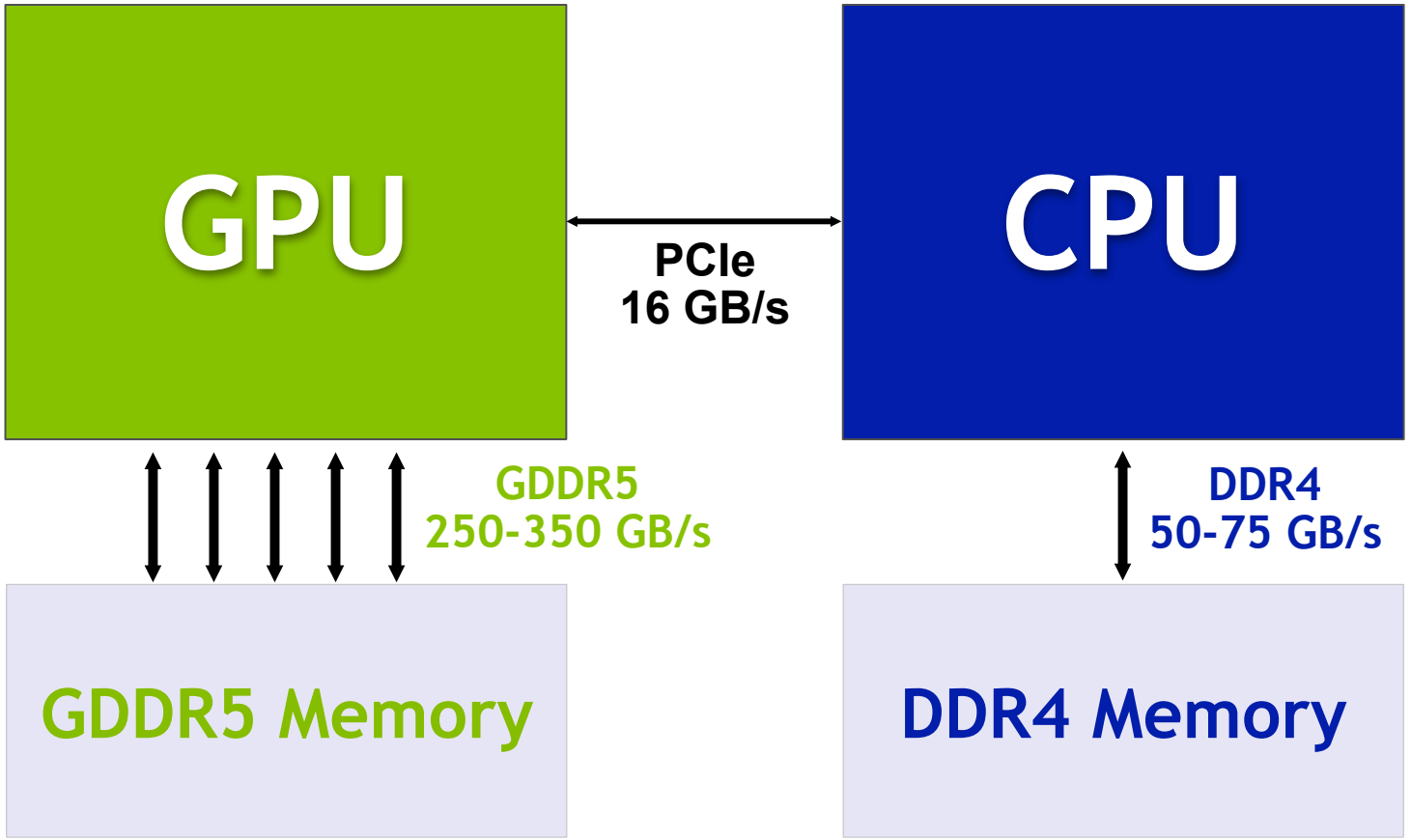
- Axiom: DRAM is 8 times more dense than a SRAM.
- Hypothesis: A core uses similar die area than 2 MB L3 (Ivy Bridge @ 22nm. fulfills this today if we left L2 aside).
- Evaluation: 2 layers, with the following alternatives (all reached similar temperatures):

Layer #1	Layer #2	Area	Latency	Bandwidth	Power cons.
2 cores + 4 MB L3	Empty	1+0 = 1	High	High	92 W.
2 cores + 4 MB L3	8 MB L3	1+1 = 2	Medium	Medium	106 W.
2 cores	32 MB. DRAM	1/2+1/2=1	Low	Low	88 W.
2 cores + 4 MB L3	64 MB. DRAM	1+1 = 2	Very low	Very low	98 W.

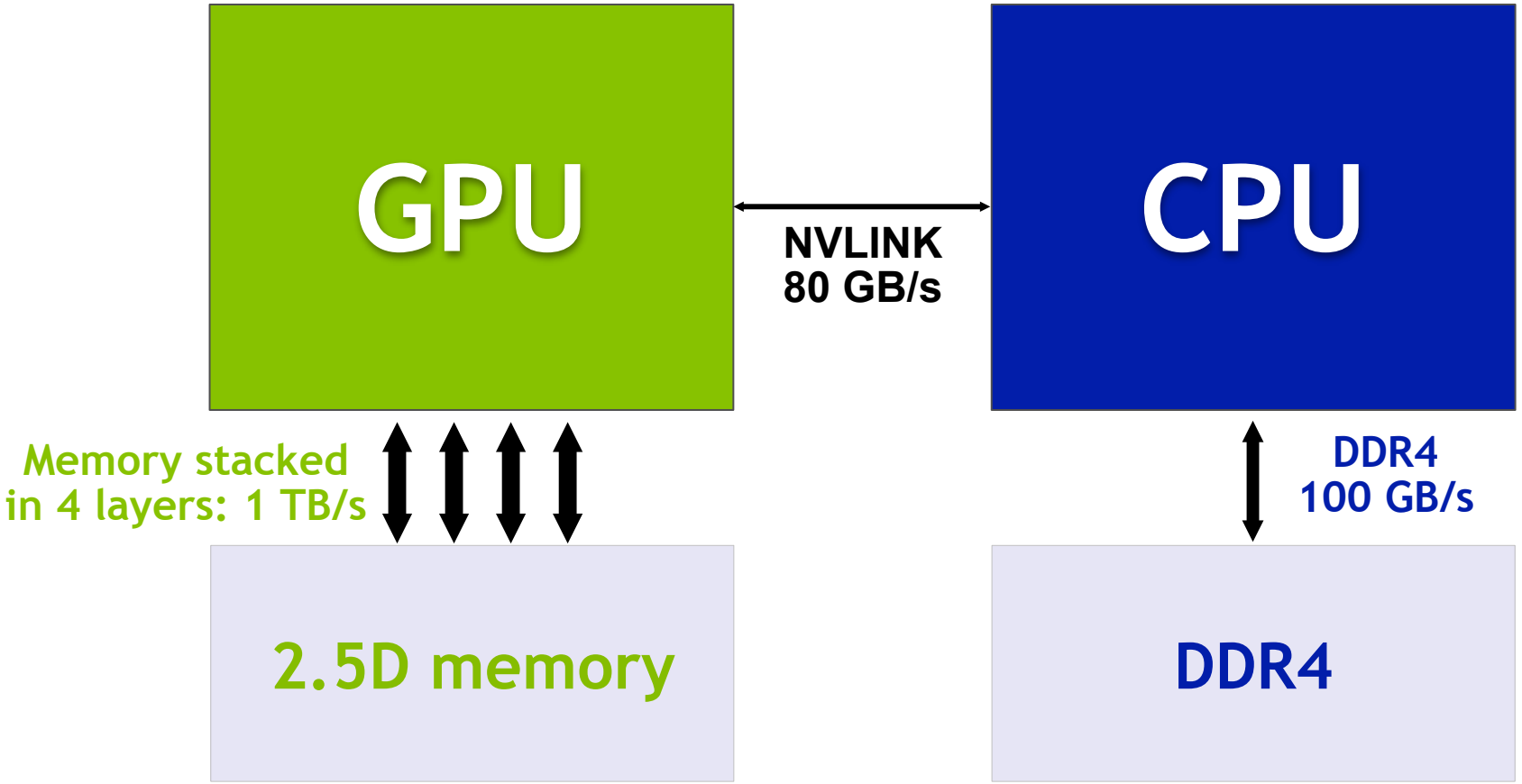
- Given the higher role played by latency, the last row is the winner: DRAM is the greatest beneficiary of 3D integration.

(*) B. Black et al. "Die Stacking (3D) Microarchitecture", published in MICRO'06.

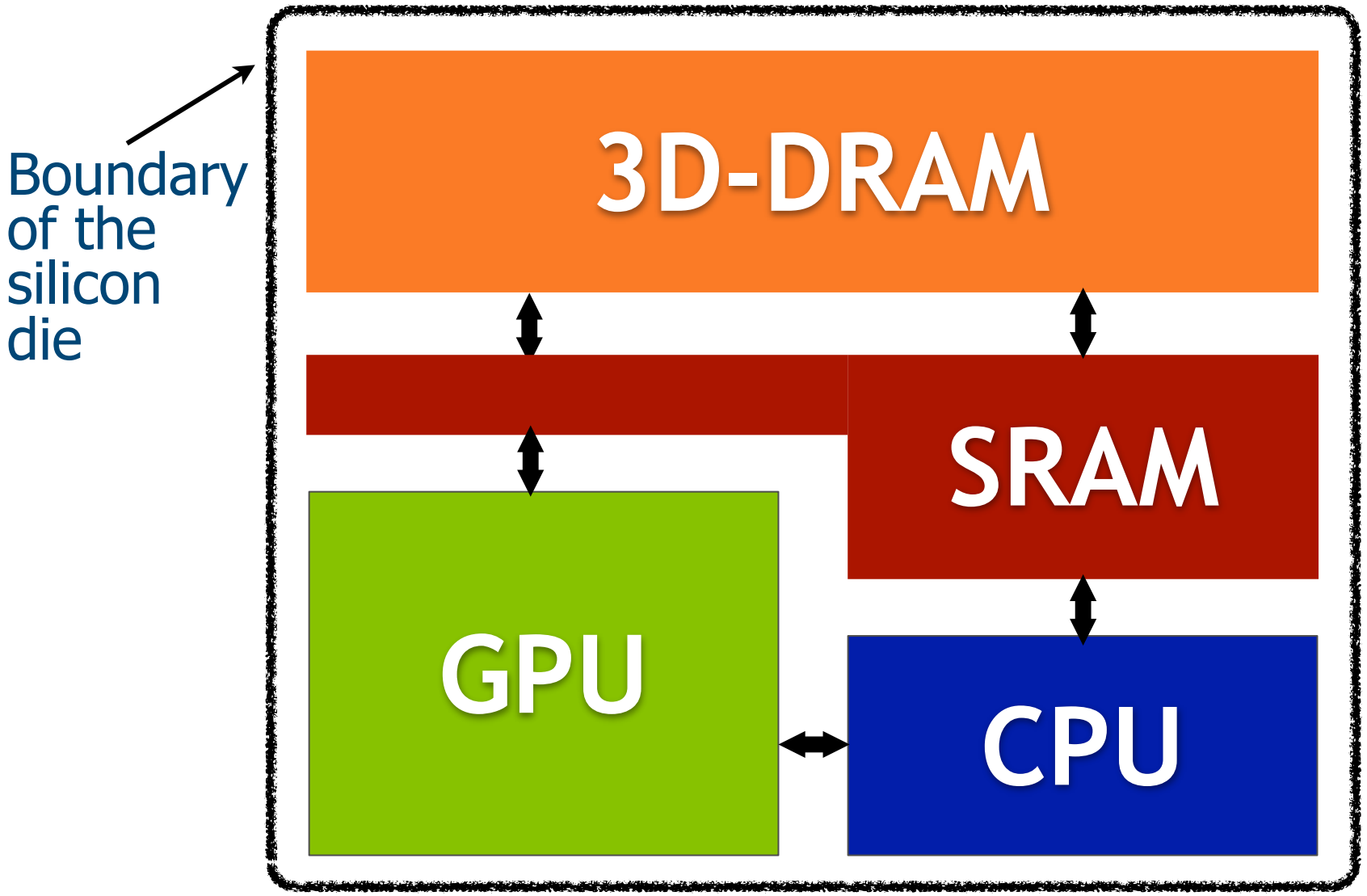
Today


GPU
CPU
PCIe
16 GB/s
GDDR5
250-350 GB/s
DDR4
50-75 GB/s
GDDR5 Memory
DDR4 Memory

In two years

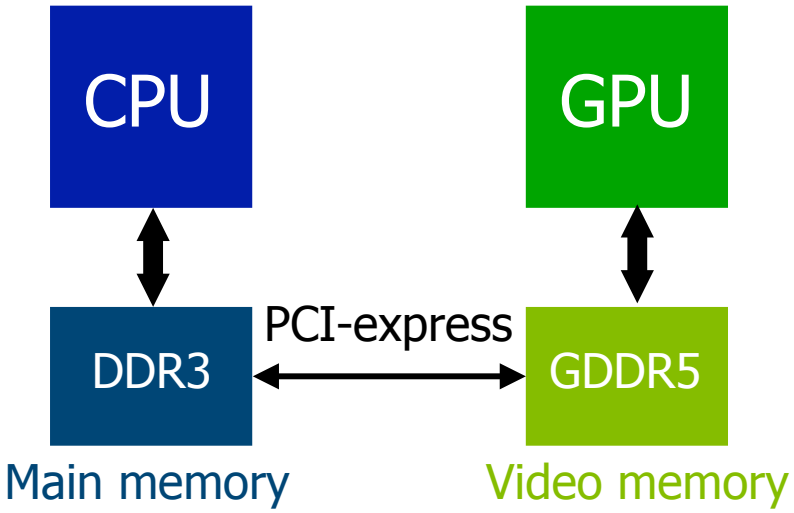

GPU
CPU
NVLINK
80 GB/s
Memory stacked
in 4 layers: 1 TB/s
DDR4
100 GB/s
2.5D memory
DDR4

In four years:
All communications internal to the 3D chip



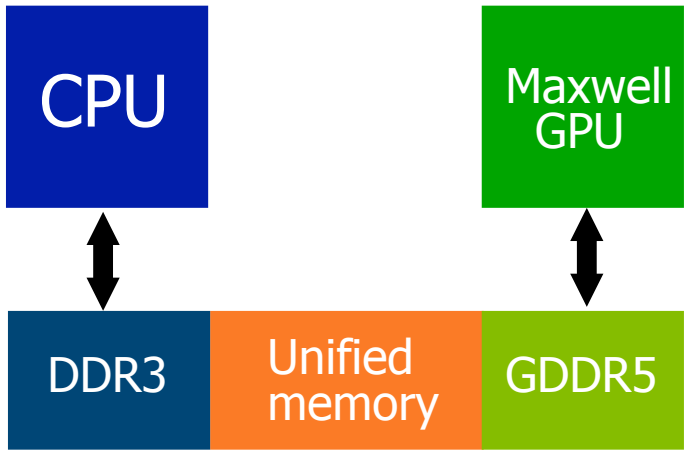
The idea: Accustom the programmer to see the memory that way

CUDA 2007-2014



The old hardware and software model: Different memories, performances and address spaces.

CUDA 2015 on



The new API: Same memory, a single global address space.

Performance sensitive to data proximity.

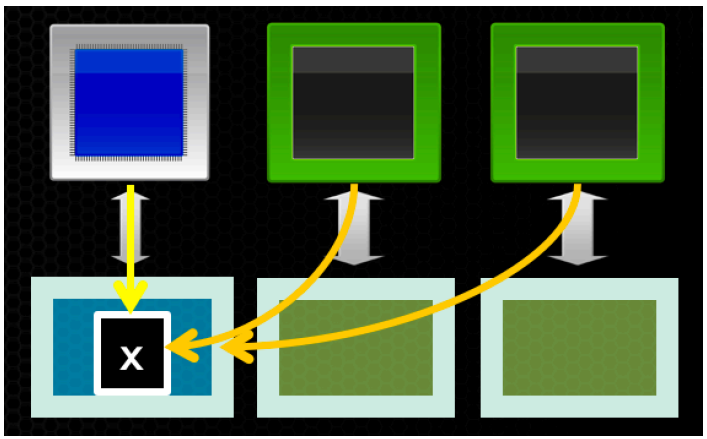
II. Programming with unified memory

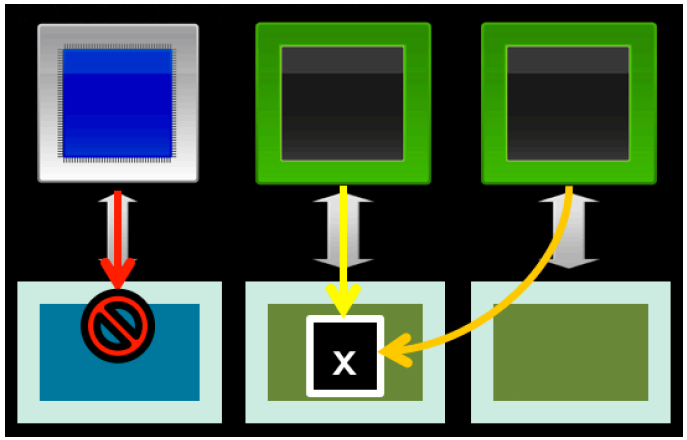
Unified memory contributions

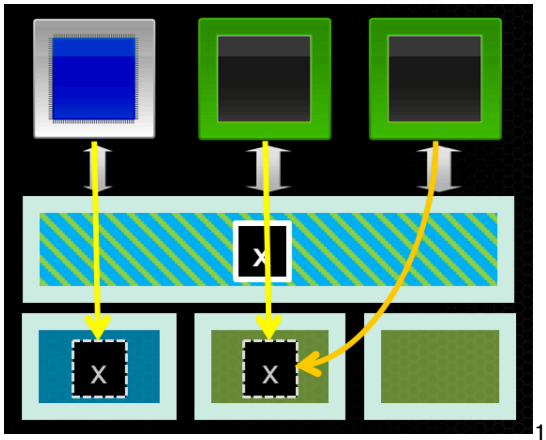
- Simpler programming and memory model:
  - Single pointer to data, accessible anywhere.
  - Eliminate need for `cudaMemcpy()`.
  - Greatly simplifies code porting.
- Performance through data locality:
  - Migrate data to accessing processor.
  - Guarantee global coherency.
  - Still allows `cudaMemcpyAsync()` hand tuning.

CUDA memory types

	Zero-Copy (pinned memory)	Unified Virtual Addressing	Unified Memory
CUDA call	cudaMallocHost(&A, 4);	cudaMalloc(&A, 4);	cudaMallocManaged(&A, 4);
Allocation fixed in	Main memory (DDR3)	Video memory (GDDR5)	Both
Local access for	CPU	Home GPU	CPU and home GPU
PCI-e access for	All GPUs	Other GPUs	Other GPUs
Other features	Avoid swapping to disk	No CPU access	On access CPU/GPU migration
Coherency	At all times	Between GPUs	Only at launch & sync.
Full support in	CUDA 2.2	CUDA 1.0	CUDA 6.0







Additions to the CUDA API

- New call: **`cudaMallocManaged(pointer,size,flag)`**
  - Drop-in replacement for `cudaMalloc(pointer,size)`.
  - The flag indicates who shares the pointer with the device:
    - `cudaMemAttachHost`: Only the CPU.
    - `cudaMemAttachGlobal`: Any other GPU too.
  - All operations valid on device mem. are also ok on managed mem.
- New keyword: **`__managed__`**
  - Global variable annotation combines with **`__device__`**.
  - Declares global-scope migratable device variable.
  - Symbol accessible from both GPU and CPU code.
- New call: **`cudaStreamAttachMemAsync()`**
  - Manages concurrently in multi-threaded CPU applications.

Unified memory: Technical details

- The maximum amount of unified memory that can be allocated is the **smallest** of the memories available on GPUs.
- Memory pages from unified allocations touched by CPU are required to **migrate back** to GPU before any kernel launch.
- The CPU cannot access any unified memory as long as GPU is executing, that is, a `cudaDeviceSynchronize()` call is required for the CPU to be allowed to access unified memory.
- The GPU has **exclusive** access to unified memory when any kernel is executed on the GPU, and this holds even if the kernel does not touch the unified memory (see an example on next slide).

III. Examples

NVIDIA

First example: Access constraints

```
__device__ __managed__ int x, y = 2;              // Unified memory

__global__ void mykernel()                        // GPU territory
{
  x = 10;
}

int main()                                        // CPU territory
{
  mykernel <<<1,1>>> ();


  y = 20;  // ERROR: CPU access concurrent with GPU
  return 0;
}
```

First example: Access constraints

```
__device__ __managed__ int x, y = 2;              // Unified memory

__global__ void mykernel()                        // GPU territory
{
  x = 10;
}

int main()                                        // CPU territory
{
  mykernel <<<1,1>>> ();
  cudaDeviceSynchronize();                        // Problem fixed!
  // Now the GPU is idle, so access to "y" is OK
  y = 20;
  return 0;
}
```

Second example: Sorting elements from a file

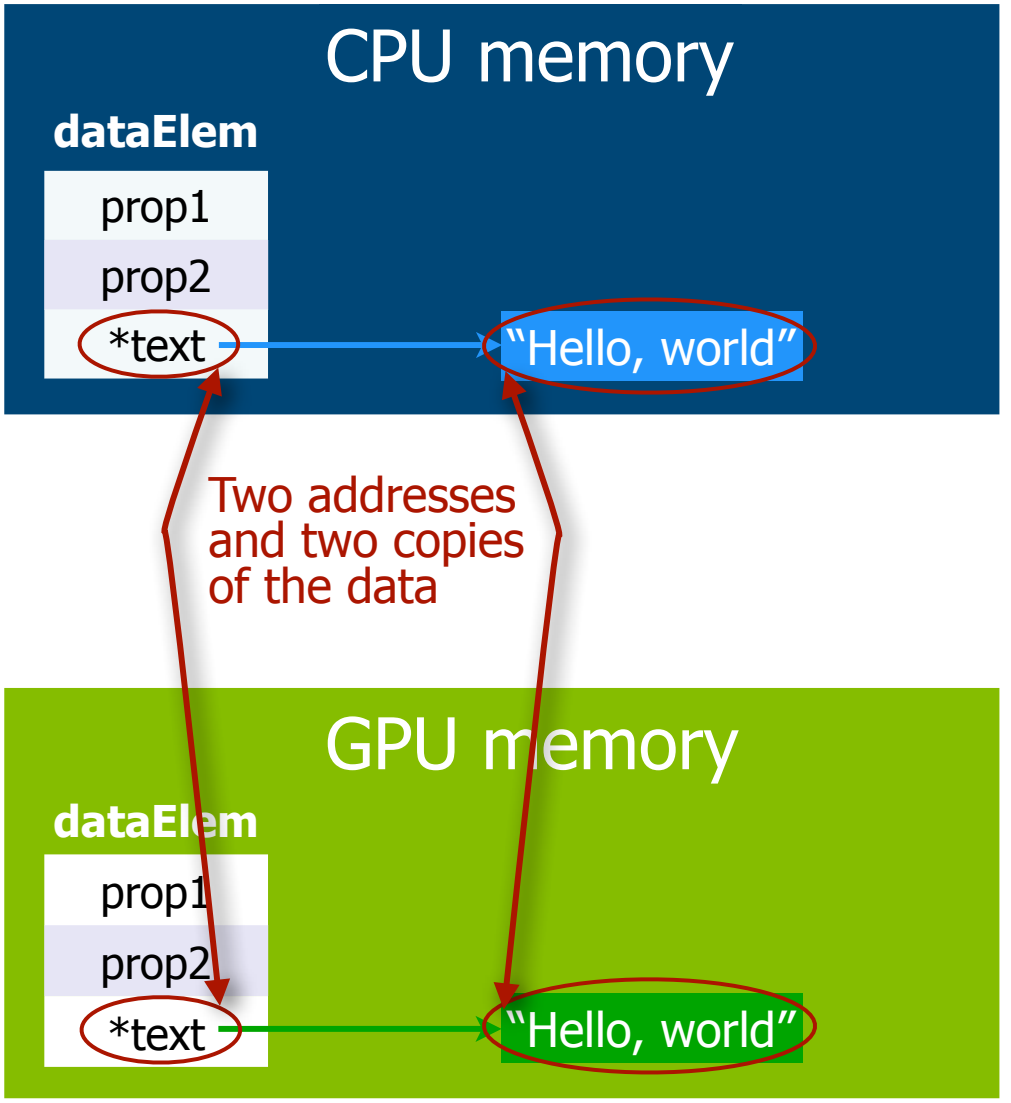
CPU code in C	GPU code from CUDA 6.0 on

`void sortfile (FILE *fp, int N)`
`{`
`  char *data;`
`  data = (char *) malloc(N);`
 	
`  fread(data, 1, N, fp);`
 	
`  qsort(data, N, 1, compare);`
 	
`  use_data(data);`
 	
`  free(data);`
`}`
	`void sortfile (FILE *fp, int N)`
	`{`
	`  char *data;`
	`  cudaMallocManaged(&data, N);`
	
	`  fread(data, 1, N, fp);`
	
	`  qsort<<<...>>>(data, N, 1, compare);`
	`  cudaDeviceSynchronize();`
	`  use_data(data);`
	
	`  cudaFree(data);`
	`}`

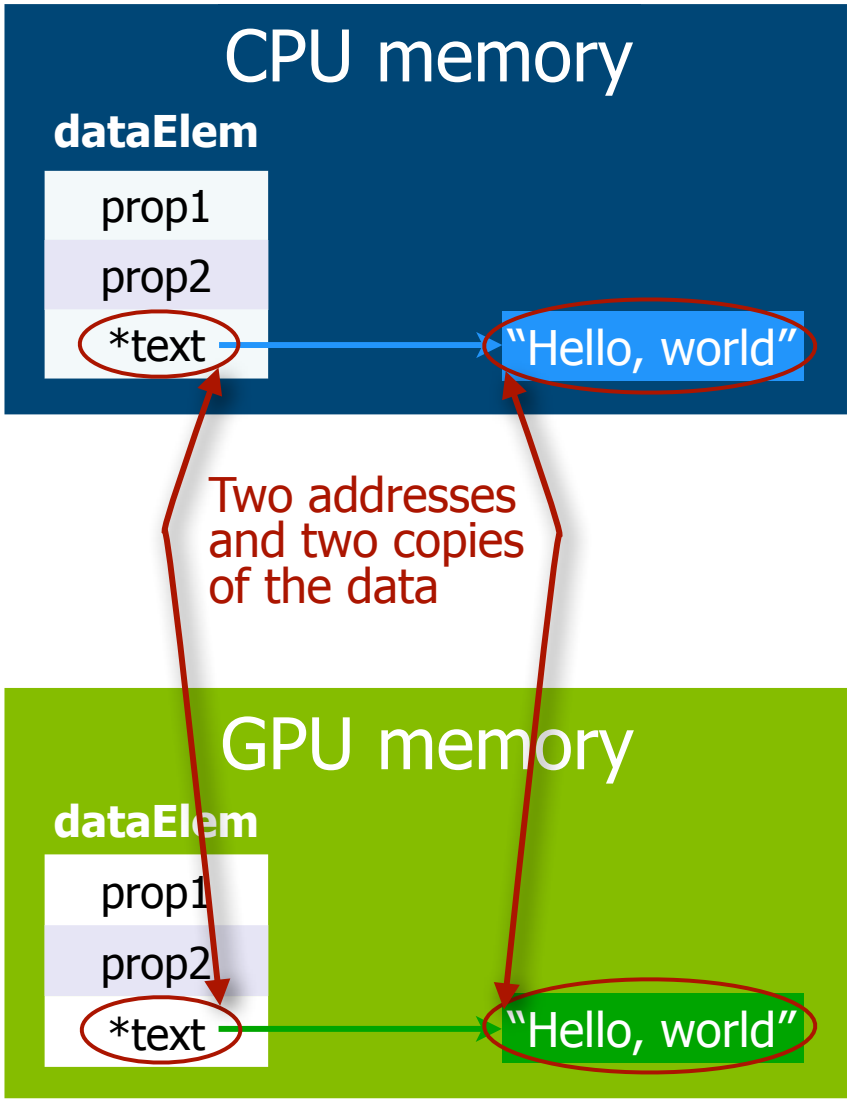
Third example: Cloning dynamic data structures WITHOUT unified memory

```
struct dataElem {
  int prop1;
  int prop2;
  char *text;
}
```

- A "deep copy" is required:
  - We must copy the structure and everything that it points to. This is why C++ invented the copy constructor.
  - CPU and GPU cannot share a copy of the data (coherency). This prevents `memcpy` style comparisons, checksumming and other validations.

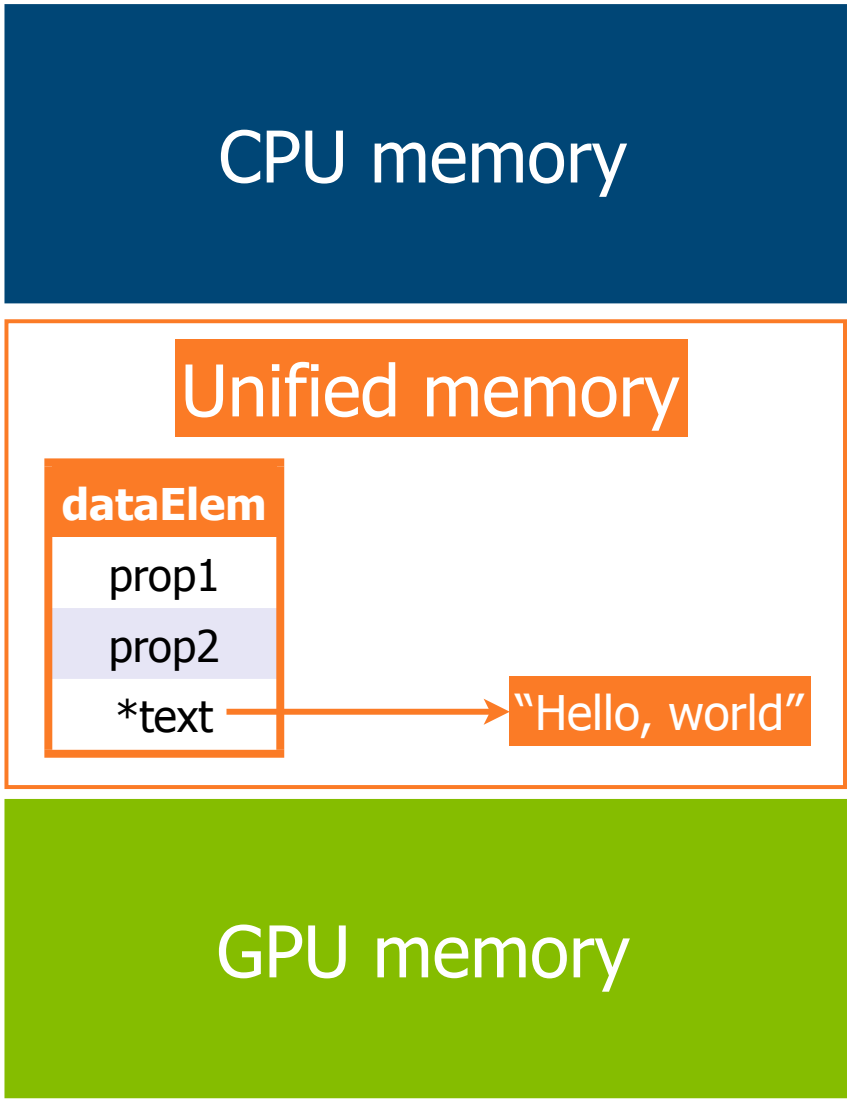


Cloning dynamic data structures WITHOUT unified memory




```
void launch(dataElem *elem) {
  dataElem *g_elem;
  char *g_text;

  int textlen = strlen(elem->text);

  // Allocate storage for struct and text
  cudaMalloc(&g_elem, sizeof(dataElem));
  cudaMalloc(&g_text, textlen);

  // Copy up each piece separately, including
new "text" pointer value
  cudaMemcpy(g_elem, elem, sizeof(dataElem));
  cudaMemcpy(g_text, elem->text, textlen);
  cudaMemcpy(&(g_elem->text), &g_text,
                                  sizeof(g_text));

  // Finally we can launch our kernel, but
  // CPU and GPU use different copies of "elem"
  kernel<<< ... >>>(g_elem);
}
```

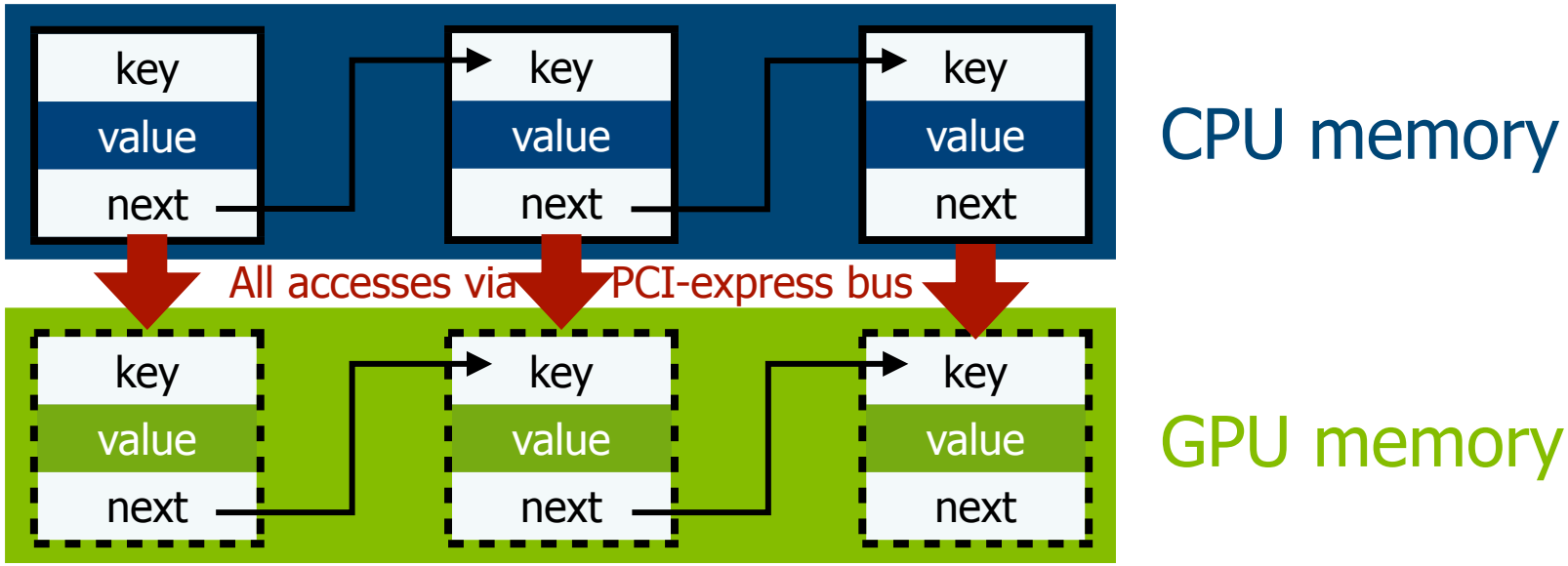
Cloning dynamic data structures WITH unified memory




```
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```

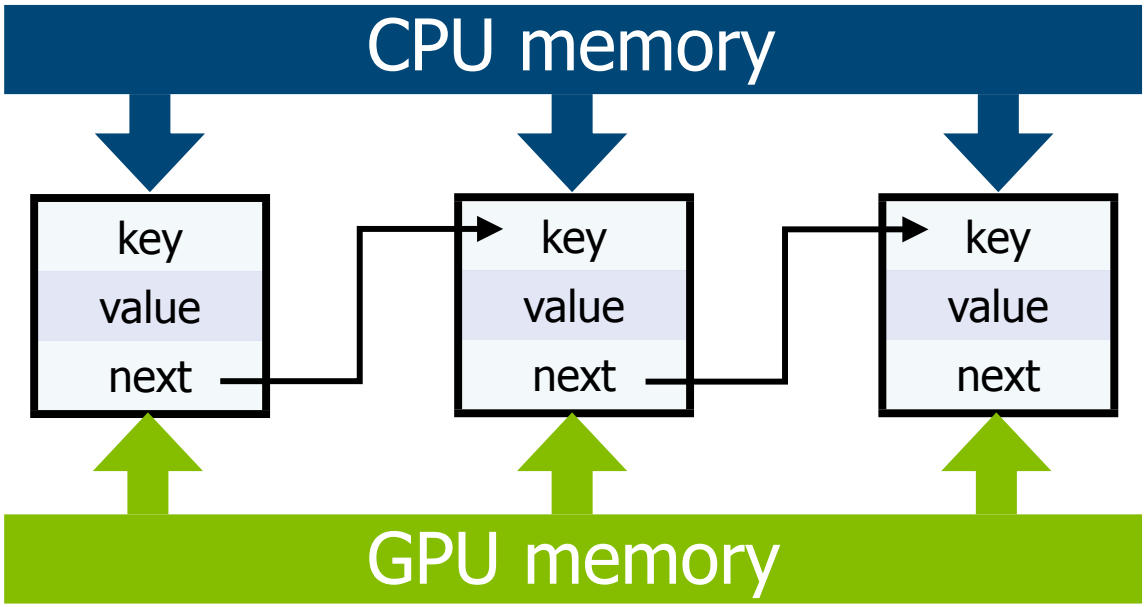
- What remains the same:
  - Data movement.
  - GPU accesses a local copy of text.
- What has changed:
  - Programmer sees a single pointer.
  - CPU and GPU both reference the same object.
  - There is coherence.
- To pass-by-reference vs. pass-by-value you need to use C++.

Fourth example: Linked lists



- Almost impossible to manage in the original CUDA API.
- The best you can do is use pinned memory:
  - Pointers are global: Just as unified memory pointers.
  - Performance is low: GPU suffers from PCI-e bandwidth.
  - GPU latency is very high, which is critical for linked lists because of the intrinsic pointer chasing.

Linked lists with unified memory



- Can pass list elements between CPU & GPU.
  - No need to move data back and forth between CPU and GPU.
- Can insert and delete elements from CPU or GPU.
  - But program must still ensure no race conditions (data is coherent between CPU & GPU at kernel launch only).

IV. Final remarks

NVIDIA

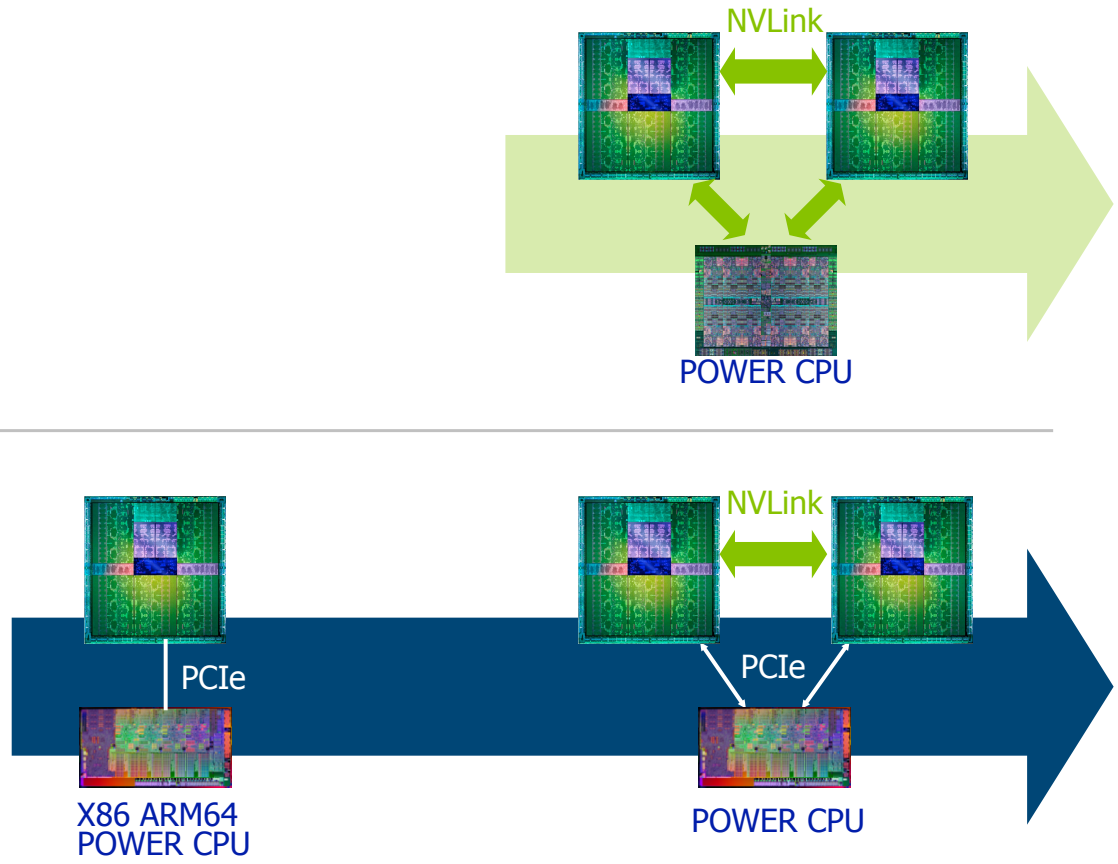
Unified memory: Summary

- Drop-in replacement for `cudaMalloc()` using `cudaMallocManaged()`.
  - `cudaMemcpy()` now optional.
- Greatly simplifies code porting.
  - Less Host-side memory management.
- Enables shared data structures between CPU & GPU
  - Single pointer to data = no change to data structures.
- Powerful for high-level languages like C++.

Unified memory: The roadmap. Contributions on every abstraction level

Abstraction level	Past: Consolidated in 2014	Present: On the way during 2015	Future: Available in coming years
High	Single pointer to data. No `cudaMemcpy()` is required	Prefetching mechanisms to anticipate data arrival in copies	System allocator unified
Medium	Coherence @ launch & synchronize	Migration hints	Stack memory unified
Low	Shared C/C++ data structures	Additional OS support	Hardware-accelerated coherence

NV-Link: High-speed GPU interconnect



2014/15: Kepler

2016/17: Pascal

Final summary

- Kepler is aimed to irregular computing, enabling the GPU to enter new application domains. Win: **Functionality**.
- Maxwell simplifies the GPU model to reduce energy and programming effort. Win: **Low-power**, **memory-friendly**.
- Pascal introduces 3D-DRAM and NV-Link. Win: **Transfers**, **heterogeneity**.
  - **3D memory** changes memory hierarchy and boosts performance.
  - **NV-Link** helps to communicate GPUs/CPUs in a transition phase towards SoC (System-on-Chip), where all major components integrate on a single chip: CPU, GPU, SRAM, DRAM and controllers.