# GPU Programming Concepts

## Assessing Parallelism

Prof. Dr. Matthias S. Müller
Dr. Christian Terboven
Dr. Sandra Wienke
Julian Miller

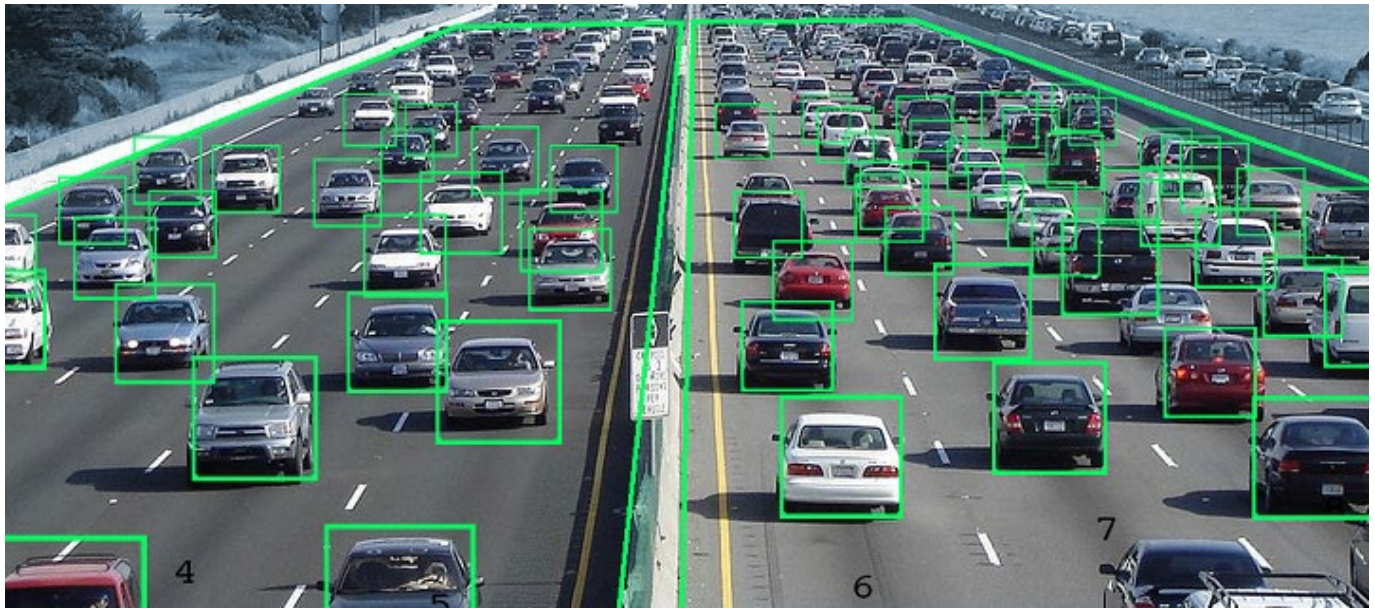
i12 High Performance Computing | RWTH AACHEN UNIVERSITY

# What is This Chapter About?

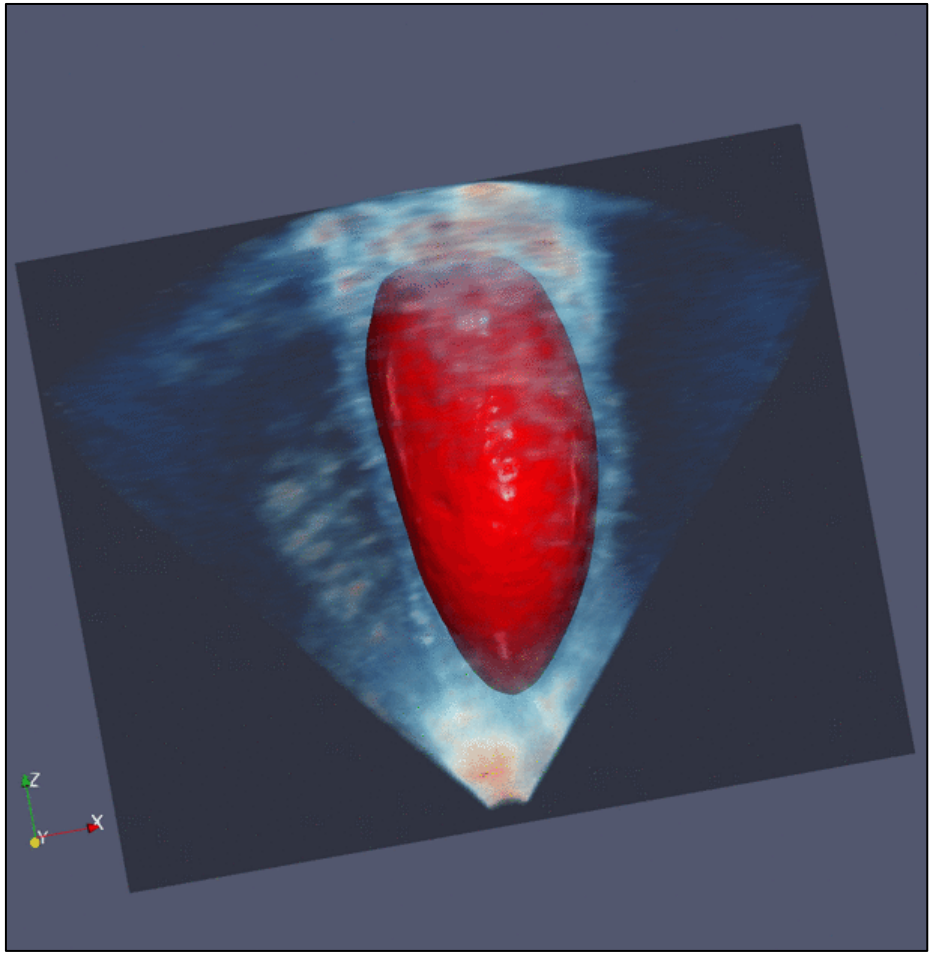
- Important concepts of programming GPUs
  - GPU application areas
  - Performance modelling
  - Occupancy

# GPU Application Areas – Imaging and Computer Vision

- Characteristics
  - Lots of images
  - Lots of independent computations, typically matrix operations



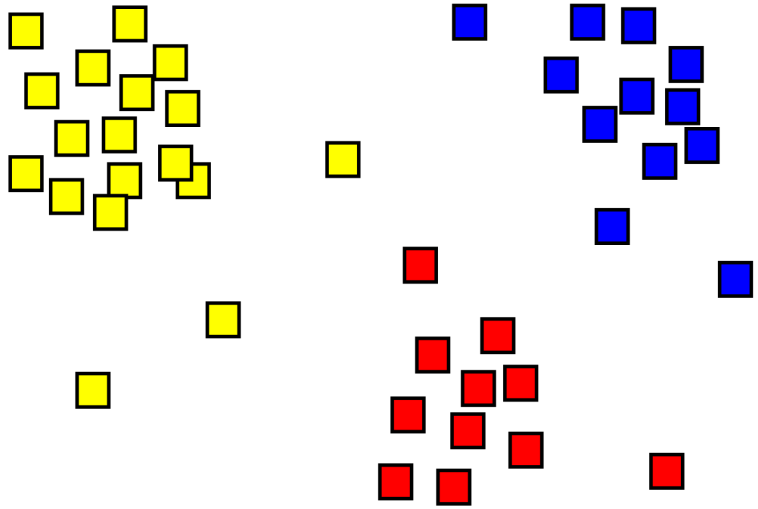

Automatic traffic surveillance. Image courtesy of NUST School of Electrical Engineering and Computer Science.



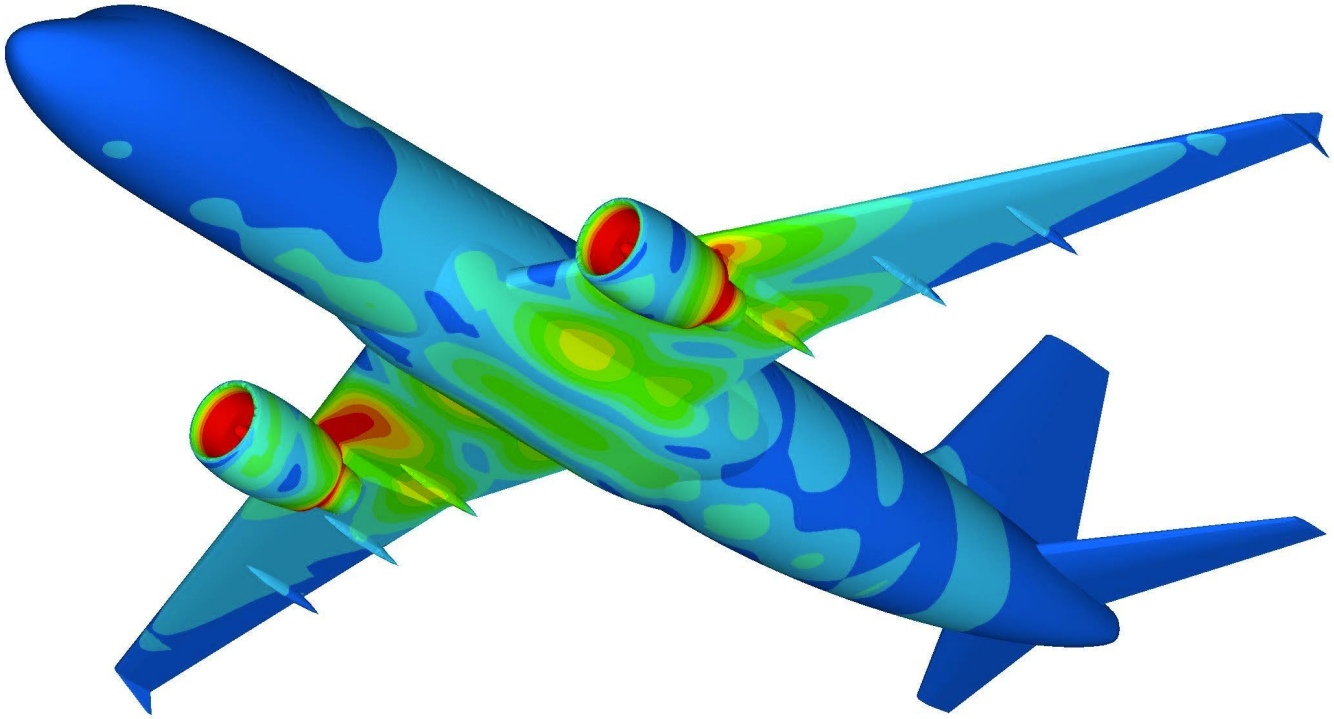

3D ultrasound of the left ventricle of the hear. Image courtesy of NVIDIA.

# GPU Application Areas – Linear Algebra

- Characteristics
  - Lots of data points, typically large matrices
  - Lots of independent computations



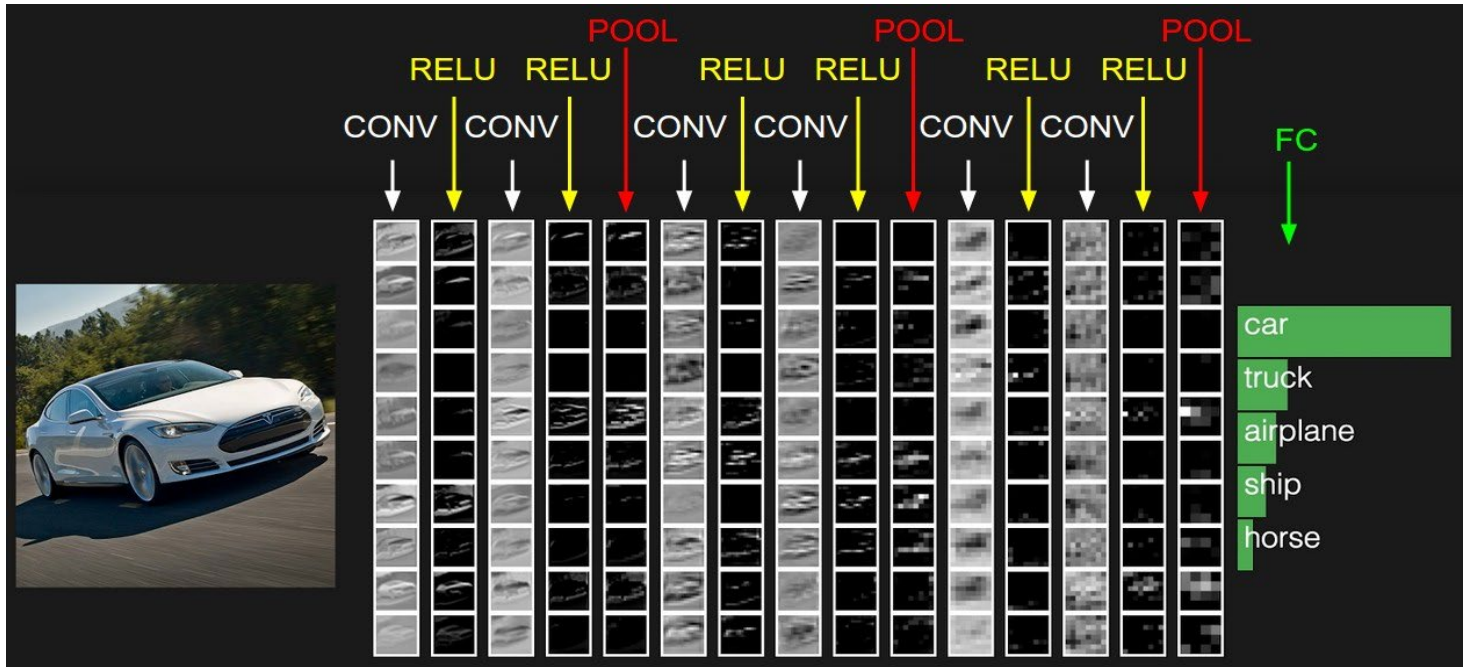
Cluster analysis of 2D data points. Image courtesy of Wikipedia.

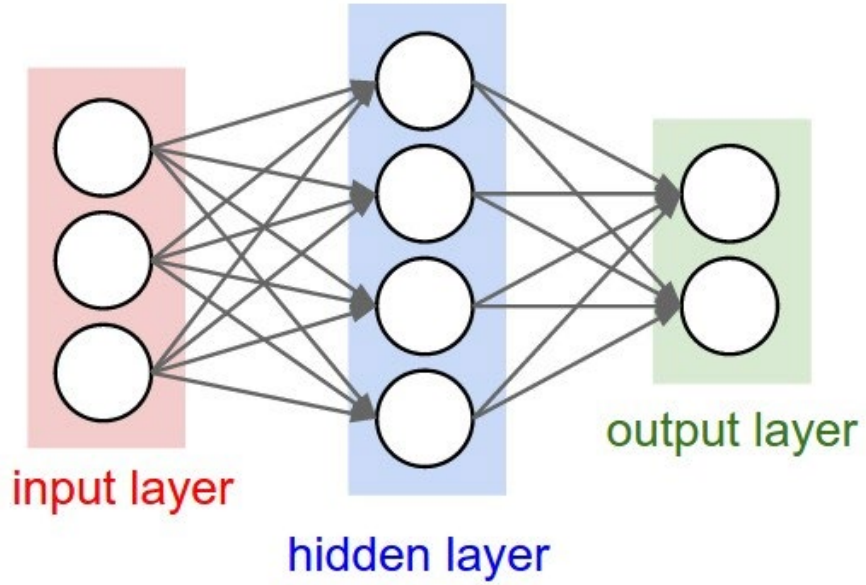


Finite element analysis of the structure of an airplane. Image courtesy of Stress Ebook LLC.

# GPU Application Areas – Machine learning

- Characteristics
  - No good closed-form model for it
  - Lots of data
  - Lots of independent computations



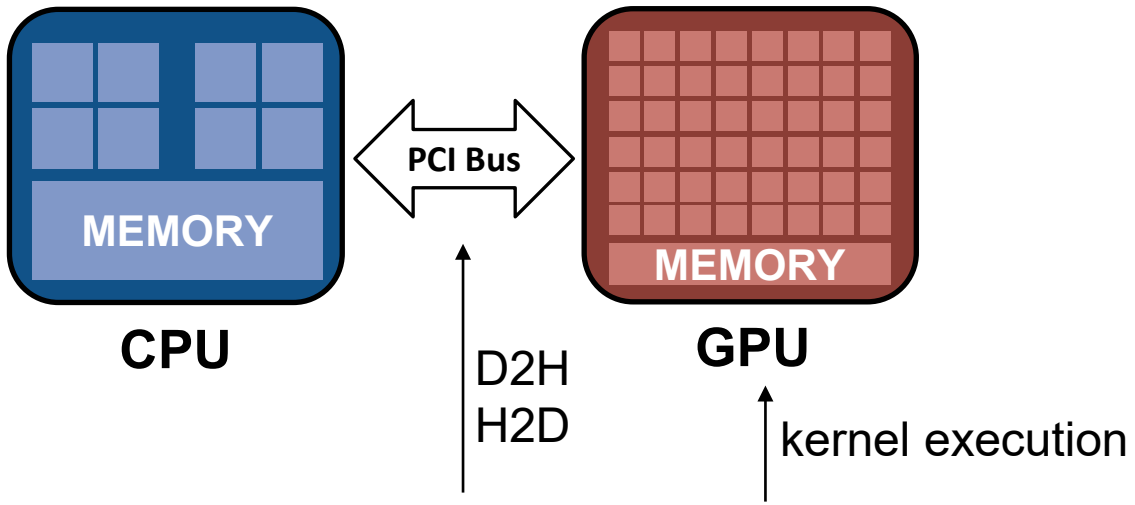

Images courtesy of CS Stanford University course cs231n

# What Kind of Problems are Well Suited for the GPU?

- Data volume
  - Goal: Large problem sizes otherwise memory transfers limiting
- Data dependency
  - Goal: structured data and no data dependencies
- Data processing order
  - Goal: consecutive order
- Data type
  - Some GPUs are much slower for double precision operations
- Code/ algorithmic complexity
  - Goal: simple instructions
- Computational intensity
  - Goal: higher intensity (Flops per byte) yields typically higher performance

# Performance Models

- Is it profitable to use a GPU?
- What performance to expect on a GPU?
- Which impact do (certain) performance optimizations have?




$$\text{runtime } t_{GPU} = t_{data} + t_{kernel}$$

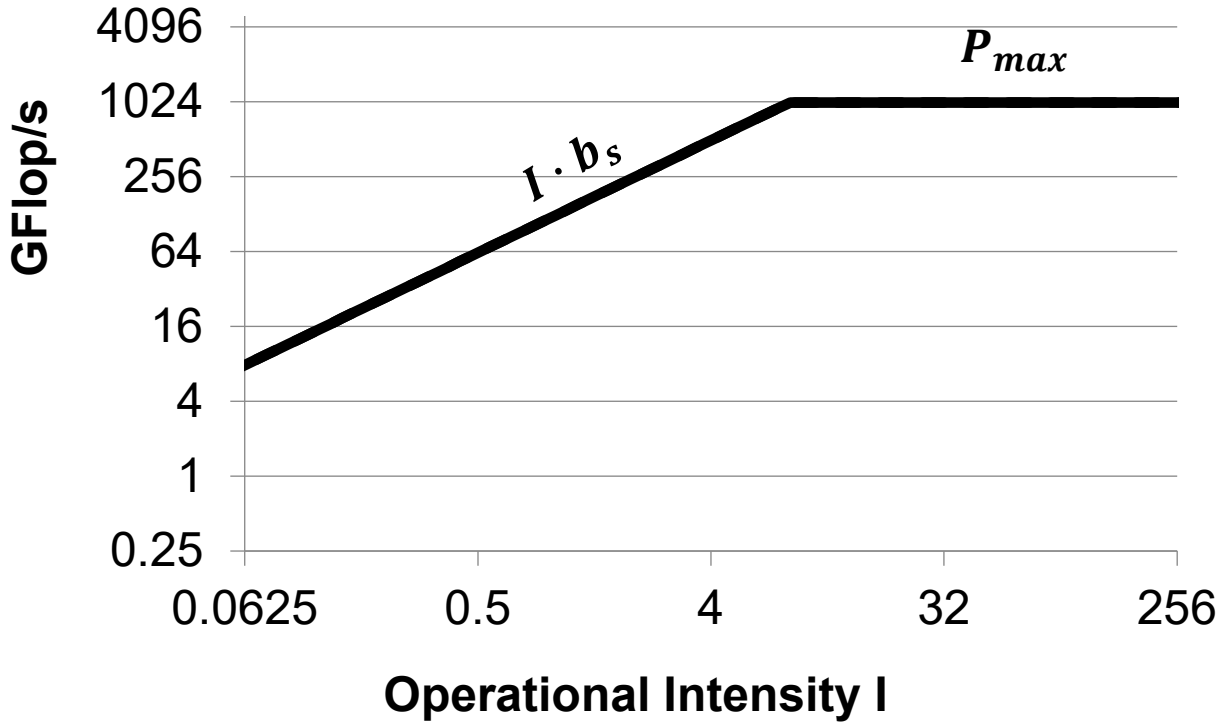
# Kernel Performance Model

- Goal: Model of kernel execution time $t_{kernel}$

Kernel execution time includes computation time & time to transfer data from GPU global memory to the registers.

# Roofline Model

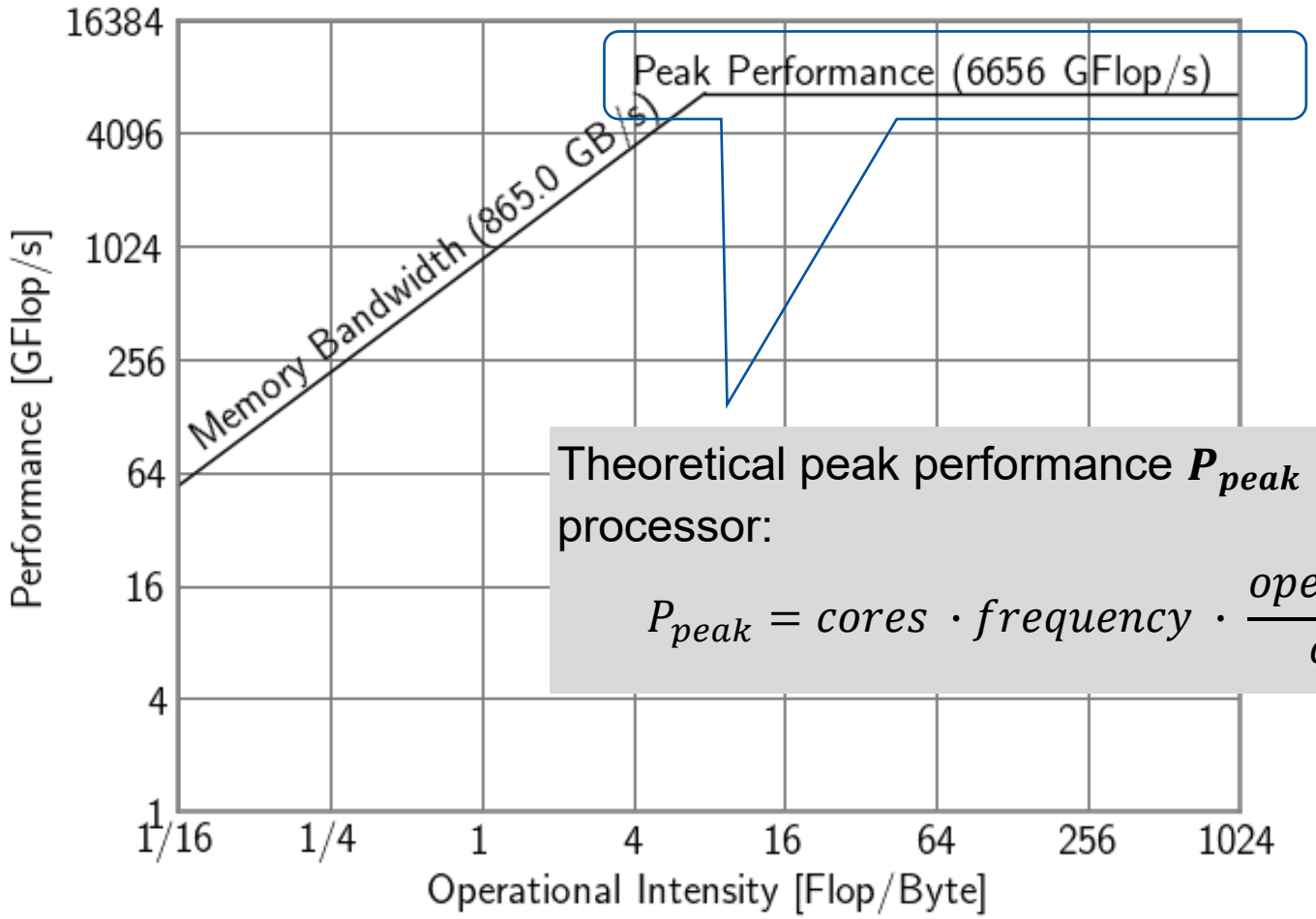
- Formal model: $P = \min(P_{\max}, I \cdot b_S)$
- Graphical model




| | |
|---|---|
| $P_{max}$: | Applicable peak performance of a loop, assuming that data comes from L1 cache (this is not necessarily $P_{peak}$) [Flop/s] |
| $I$: | operational itensity |
| $b_s$: | achievable bandwidth over slowest data path [words/s] |

# Roofline Model on V100

- Peak performance of an NVIDIA V100 (RWTH CLAIX) is 6656 GFlop/s (5120 cores * 1.3 GHz * 1 OPs/cycle)



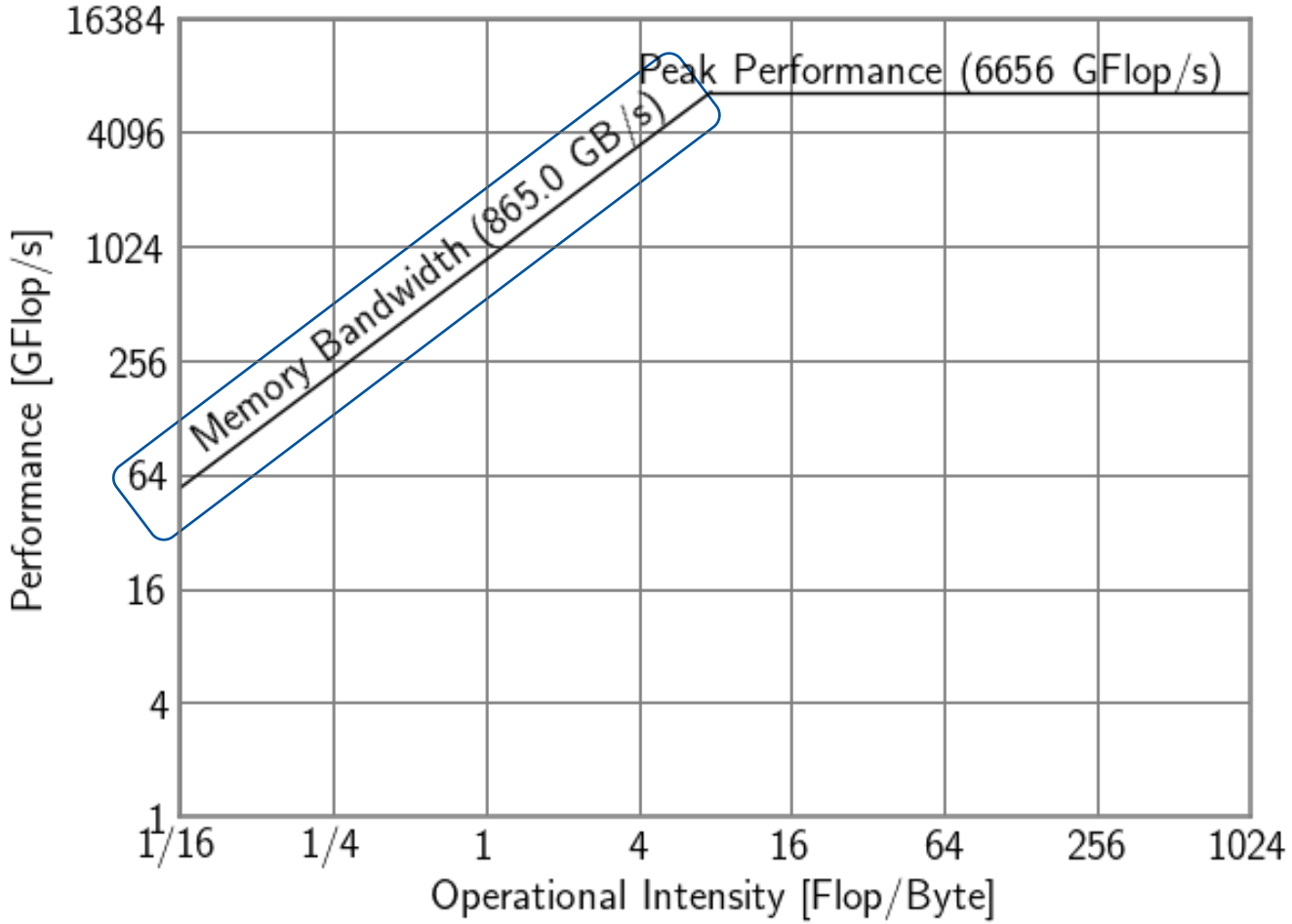

Theoretical peak performance $P_{peak}$ of a processor:

$$P_{peak} = cores \cdot frequency \cdot \frac{operations}{cycle}$$

[1] Deakin T, Price J, Martineau M, McIntosh-Smith S. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. 2016. Paper presented at P^3MA Workshop at ISC High Performance, Frankfurt, Germany.

# Roofline Model

- Sustainable memory bandwidth $b_s \approx 865$ GB/s measured with BabelStream [1]




[1] Deakin T, Price J, Martineau M, McIntosh-Smith S. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. 2016. Paper presented at P^3MA Workshop at ISC High Performance, Frankfurt, Germany.

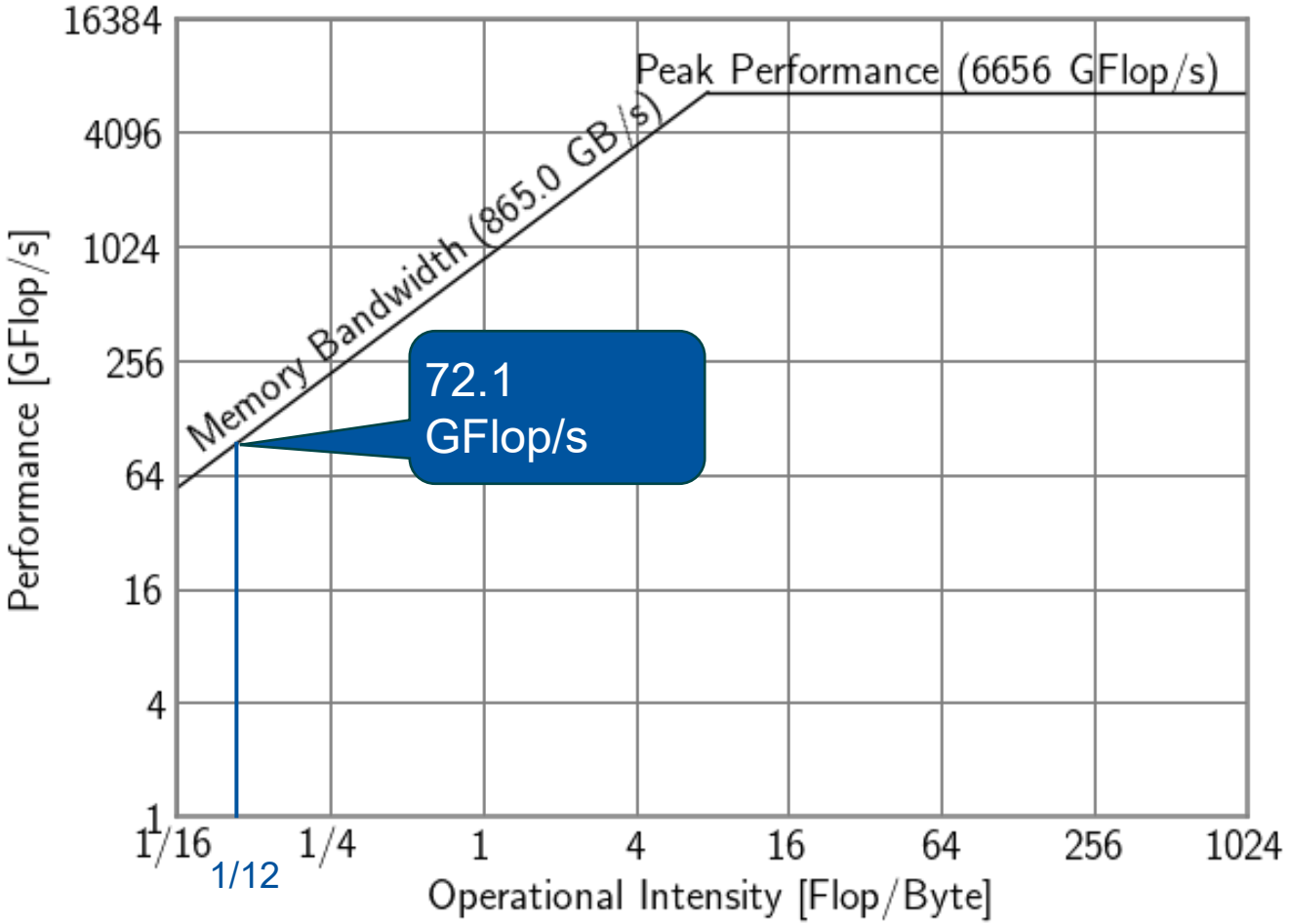
# Example SAXPY

```
void saxpy(int n, double a, double *x, double *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }

  // Invoke saxpy kernel
  saxpy(n, a, x, y);
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

LOAD x[i], y[i]
STORE y[i]
2 Flops

# SAXPY: Roofline Model

- LOAD of x[i] and y[i], STORE of y[i], 2 FLOPS → I = 2/24 Flop/ Byte




[1] Deakin T, Price J, Martineau M, McIntosh-Smith S. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. 2016. Paper presented at P^3MA Workshop at ISC High Performance, Frankfurt, Germany.

# Kernel Performance Model

- Goal: Model of kernel execution time $t_{kernel}$

Kernel execution time includes computation time & time to transfer data from GPU global memory to the registers.

- Adaption to get kernel execution time → basic idea:
  - $t_{compute} = \frac{arithmetic\ operations\ [Flop]}{P_{max}}$
  - $t_{memory} = \frac{data\ transfers(LOAD,STORE)\ [words]}{b_s}$

$$t_{kernel} = \max(t_{compute}, t_{memory})$$

  - More detailed models: GPURoofline [1], Boat Hull Model [2]

[1] Jia, H., et al. (2012). GPURoofline: A Model for Guiding Performance Optimizations on GPUs, Berlin, Heidelberg, Springer Berlin Heidelberg.
[2] Nugteren, C. and H. Corporaal (2012). The boat hull model: enabling performance prediction for parallel computing prior to code development. Proceedings of the 9th conference on Computing Frontiers. Cagliari, Italy, ACM: 203-212.

# Offloading Performance Model

- Goal: Model the total execution time $t_{GPU}$ including data transfers
  - $t_{GPU} = t_{H2D} + t_{kernel} + t_{D2H}$
- Data transfers to/ from device
  - $t_{data} = t_{H2D} + t_{D2H}$
- Basic idea: include latency $\boldsymbol{\alpha}$ and PCIe bandwidth $\boldsymbol{b_{PCI}}$
  - $t_{H2D} = \alpha + \frac{data\ transfers(LOAD)}{b_{PCI}}$
  - $t_{D2H}$ for STORES respectively
  - More details of model in [1]

[1] Boyer, M., et al. (2013). Improving GPU Performance Prediction with Data Transfer Modeling. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum.

# Occupancy

- How much “parallelism”?
- Occupancy (per SM) = $\frac{\text{active warps}}{\text{max. supported active warps}}$

- Low occupancy: indication for poor instruction issue efficiency
  - Not able to hide latencies

- Possible causes of low occupancy
  - Unbalanced workload within/ across blocks
  - Too few blocks launched