
The Hitchhiker's Guide to Performance Portability

Giorgio Amati, g.amati@ Cineca.it
Piero Lanucara, p.lanucara@ Cineca.it

Agenda

- ✓ Description of the work
- ✓ Figure of Merit (FOM)
- ✓ HW characterization & Roofline model
- ✓ Looking for the limit
- ✓ An example: BGK2D
 - Application efficiency
 - Architectural efficiency
 - Performance portability plot

Description of the work

The “naive” idea is to find a reply to the following questions:

- ✓ Fixing a generic HW, e.g. a GPU, which is the theoretical performance that we can achieve for a defined code/mini app?
- ✓ How far are we from the maximum performance achievable?
- ✓ How much of this performance is portable across different HW?
- ✓ How much of this performance is portable with different parallel paradigm?

Figure of Merit (FOM)

- **Time to Solution (TTS):** It depends on the size of the problem; it could be not useful if running on different HW with different problem size
- **Single Iteration Time:** It depends on the size of the problem and for many situations it could not be a constant for each iteration (e.g. adaptive timestep for CFD simulations).
- **Lattice (or Grid) update for second (LUPs):** It is a velocity. It doesn't depend on the problem size (if it is big enough not to stay entirely on cache) if the complexity of the problem is $O(n)$. If not, it still depends on the problem size.
- **Sustained MFlops:** It is a velocity; it gives information of how many floating-point operations are done. It is a valid FOM if and only if the operations performed by the code are the minimum necessary to solve the problem¹.
- **Memory Troughput:** it is a velocity; like LUPs it is a valid FOM, if all the unnecessary data movement (e.g. temporary arrays abuse) are removed.
- Other???

HW Characterization & Roofline model

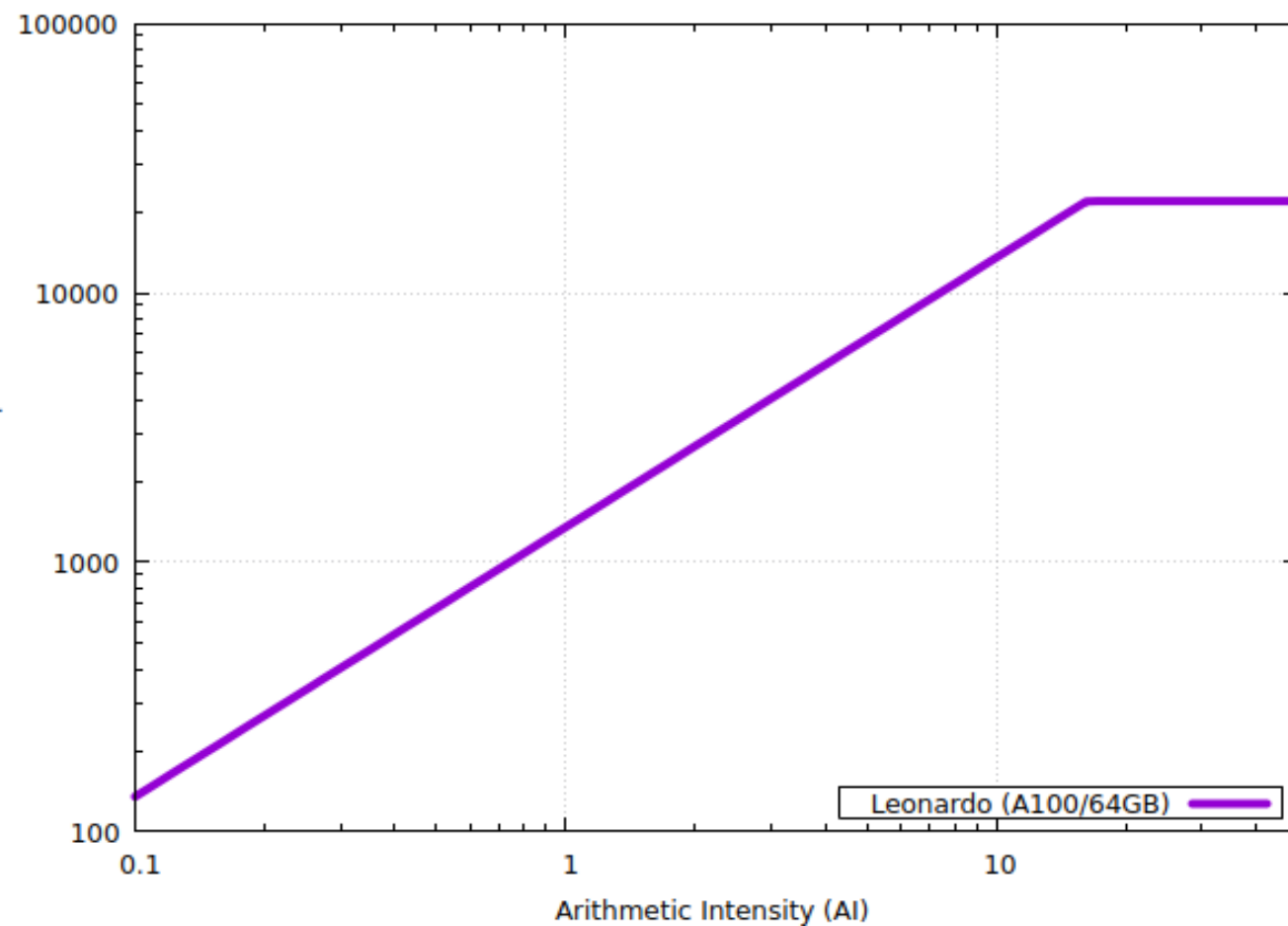
- ✓ A HW, both CPU and GPU, can be characterized in a quantitative way by:
 - **Peak FP:** how many Floating-point operations could (theoretically) be performed. It depends on many issues (e.g. # of FPUs, frequency of FPUs, width of Datapath). It could also depend on the precision used, e.g. FP32 vs P64. This is a theoretical value: the sustained figure is always lower.
 - **Bandwidth:** how many bytes "could" be moved back and forth RAM to FPU.
 - Memory throughput is different: it is the measured BW of a code, is it a % of theoretical BW (always less than 100%)
- ✓ **Caveat:** FP Peak is obtained multiplying #FPU, #Flop completed per cycle and FPU frequency. Not taking in account
 - Instructions
 - OS interactions
 - Data movement

HW Characterization & Roofline model

- ✓ The Roofline Model allows us to give qualitative information about theoretical performance that can be achieved. Performance is ordered according to Arithmetical Intensity (AI), i.e. #flops performed/#byte moved. $AI=1$ means that to perform a single floating-point operation we must move (only) 1 byte.
- ✓ This mean that to perform a single precision (FP32) sum we must
 - Load 2 numbers: i.e. load 4+4 bytes from RAM
 - Store 1 result: i.e. store 4 bytes to RAM
 - So, a code that performs only sums, with no data reuse, will present an $A.I. = 1/12 = 0.0832$. Double precision simulations (FP64) will present $AI = 0.0416$.

Roofline model

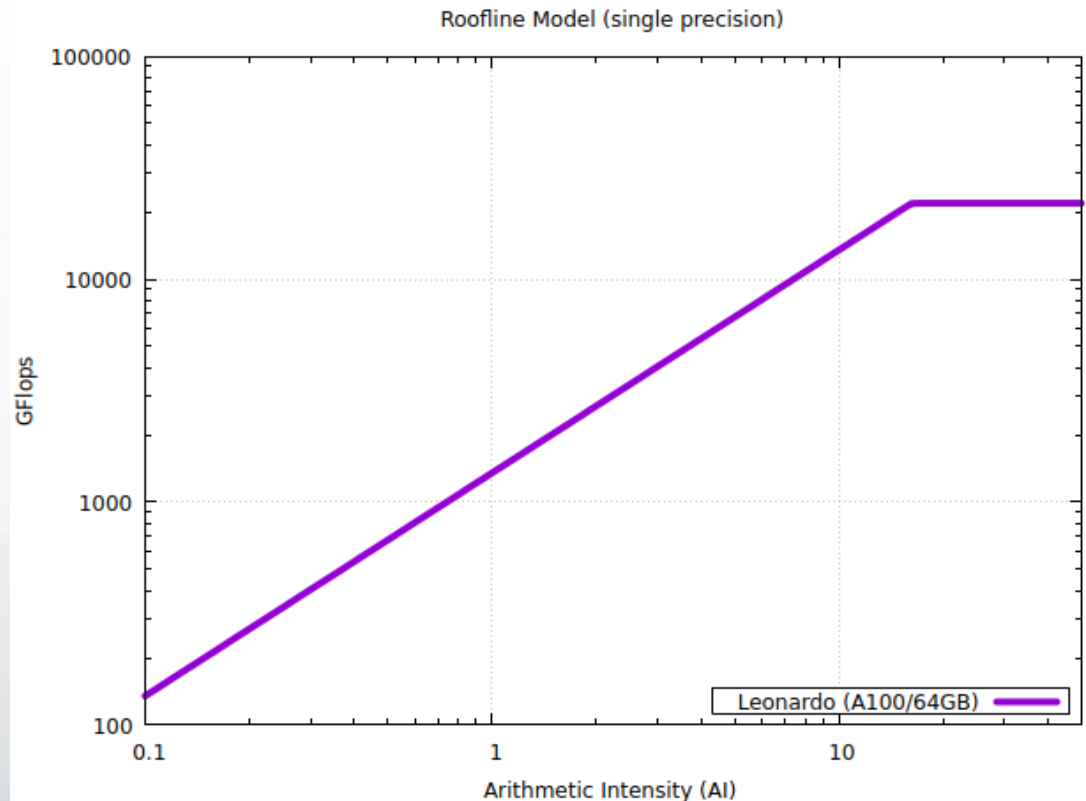
Roofline Model (single precision)



$$\text{GFLOP/s} \leq \min \begin{cases} \text{Peak GFLOP/s} \\ \text{Peak GB/s} \times \text{Arithmetic Intensity} \end{cases}$$

HW Characterization & Roofline model

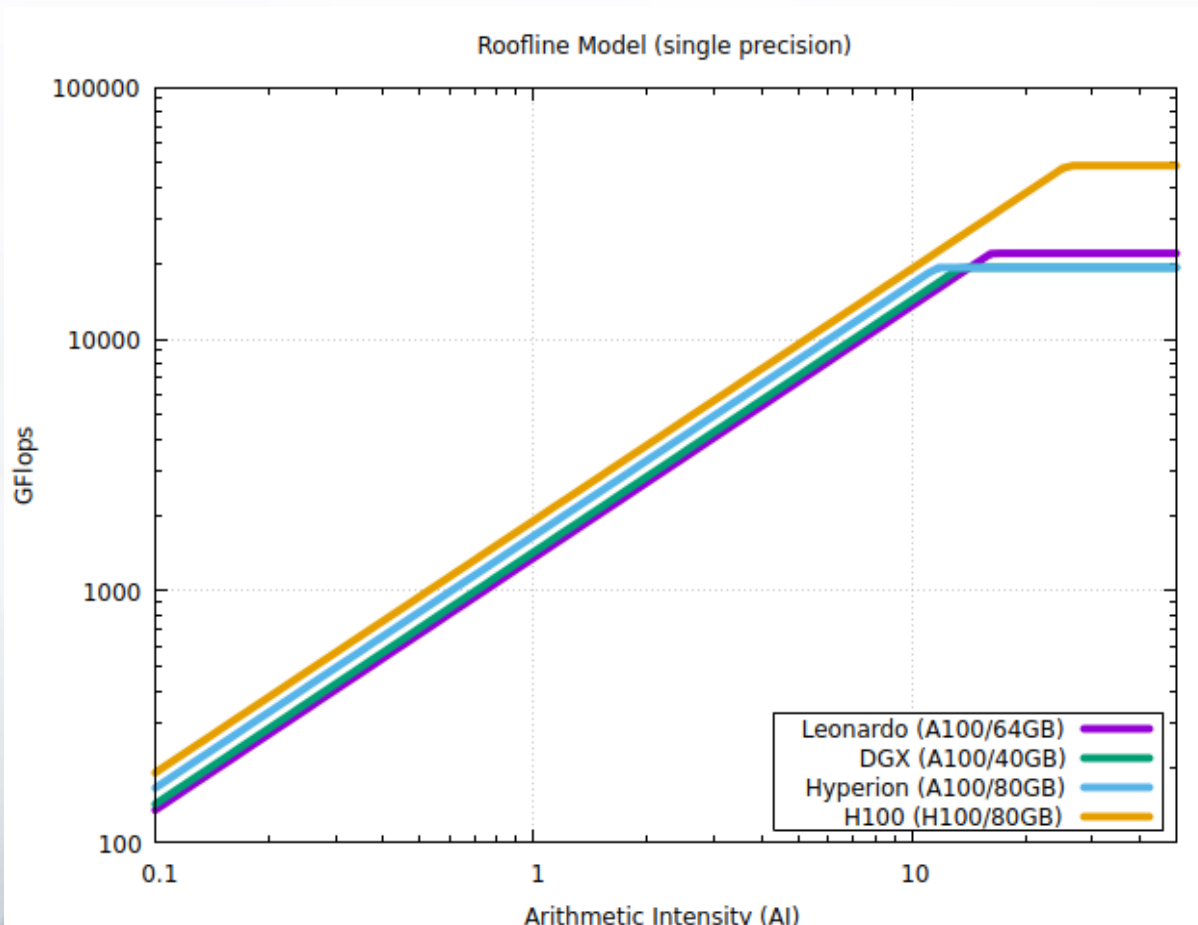
- ✓ For this model two regions can be defined.
 - **BW dominated region:** in this region data movement dominates performance, i.e. many data movement to perform and few Flops to perform. Usually, for today HW, it happens when $AI < 10$
 - **FP dominated region:** in this region floating point dominates performance, i.e. few data movements and many operations. Usually, it happens when $A.I. > 10$



A.I.=10 means that for every byte moved 10 floating point operations are performed.

HW Characterization & Roofline model

- ✓ (relatively) easy to improve FP Peak
- ✓ Harder to improve BW



A100	Leonardo		DGX		Hyperion	
	Peak	Measure	Peak	Measure	Peak	Measure
Cores	124*64	-	108*64	-	108*64	-
FP64	11.2 TF/s	10.9 TF/s	9.7 TF/s	9.46 TF/s	9.7 TF/s	9.55 TF/s
FP32	-	21.9 TF/s	19.5 TF/s	19.2 TF/s	19.5 TF/s	19.2 TF/s
BW	1.64 TB/s	-	1.55 TB/s	-	2.04 TB/s	-
BW (r)	-	1.32 TB/s	-	1.35 TB/s	-	1.59 TB/s
BW (W)	-	1.37 TB/s	-	1.50 TB/s	-	1.76 TB/s

	H100 – PCI	
	Peak	Measure
Cores	114*128	-
FP64	25 TF/s	24.3 TF/s
FP32	51 TF/s	48.8 TF/s
BW	2.0 TB/s	-
BW (r)	.	1.83 TB/s
BW (w)	.	1.94 TB/s
PCI BW	-	.025 TB/s

Looking for the limit

- ✓ With some information about the code is possible to “extrapolate” the theoretical limit achievable. Measuring (the correct) FLOPs and knowing (the exact) A.I. is possible to define
 - In which region the code is laying
 - The theoretical limit terms of FLOPs and so how far from the theoretical limit we are.
 - we will show a possible procedure is drawn in the following section, using an in-house Fortran developed code
- ✓ If is not possible for all the code maybe for the main time-consuming loops...

An example: BGK2D

- ✓ 2D CFD code written in Fortran. Using single precision, it presents $AI=1.4$. This means that it is a BW dominated code. Its complexity is $O(n)$ and this means that we can use as FOM the GLUPs (Giga Lattice Update per second). About ~ 100 Flops are used to upgrade a single Lattice, so ~ 100 GFLOPs is equivalent to 1GLups.
- ✓ The code has been used on different GPU using these parallel paradigms, if supported by compilers,
 - OpenMP Offload
 - OpenACC
 - Do concurrent
 - Sycl (under development)
- https://github.com/gamati01/BGK2D_GPU

Performance Limit for BGK2D

- BGK2D in single precision presents an AI of 1.4.
- Using an A100@64GB, with a theoretical BW of ~1.64 TB/s we get a limit of $1.64 \times 1.4 \sim 2300$ GFlops.
- Because we know that about ~100 Flops are required to upgrade a single lattice we can get the max MLups achievable and compare this one with the measured one.
- Using The A100@64GB we get a theoretical limit of about ~2300 Gflops or 23'000 MLups.

Device		Offload	OpenAcc	DC	Bw	Th.Mlups	Compilers
A100@64GB		16634	14901	14491	1640	22960	nvfortran

Performance Limit for BGK2D

✓ BGK2D in single precision presents for different HW

Device	Offload	OpenAcc	DC	Bw	Th.Mlups	Compilers
V100@32GB	9099	7779	7721	900	12600	nvfortran
A100@64GB	16634	14901	14491	1640	22960	nvfortran
A100@40GB	15467	16008	15993	1550	21700	nvfortran
H100@80GB (SXM)	26219	30619	31350	3350	46900	nvfortran
H100@80GB (PCI)	18810	22343	21492	2000	28000	nvfortran
GH200@96GB	27164	30536	31478	4000	56000	nvfortran
MI250 (1 GDC)	12880	11711	-	1600	22400	ftn
MI250 (1 GDC)	4858	-	-	1600	22400	flang
Intel 1100	5000	-	-	1220	17192	ifx

Application efficiency

- ✓ BGK2D in single precision presents for different HW
- ✓ Starting from a defined FOM we can normalize with respect the best result we get the efficiency for each device

Device	Offload (%)	OpenAcc (%)	DC
V100	100%	85%	84%
A100@64	100%	89%	87%
A100@40	96%	100%	99%
H100@80 (SXM)	83%	97%	100%
H100@80 (PCI)	84%	100%	96%
GH200@96GB	86%	97%	100%
MI250x- ftn	100%	90%	-
MI250x-flang	100%	-	-
Intel T100	100%	-	-

Architectural efficiency

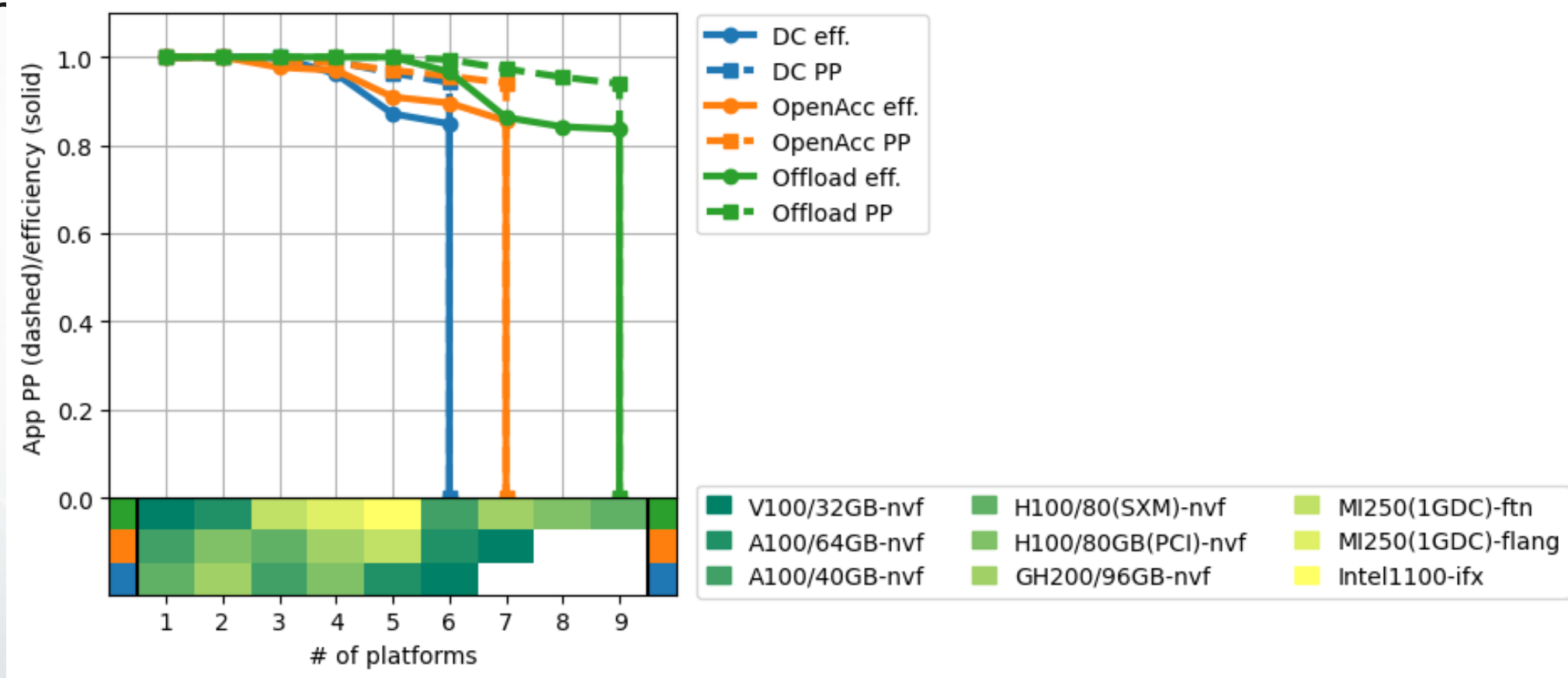
- ✓ How far we are from the limit?
- ✓ If we have the performance limit it is possible to understand how good is the whole framework (device, compiler, parallel paradigm)

Device	Offload (%)	OpenAcc (%)	DC
V100	72%	62%	61%
A100@64	72%	65%	64%
A100@40	71%	74%	74%
H100@80 (SXM)	56%	65%	67%
H100@80 (PCI)	67%	80%	77%
GH200@96GB	49%	55%	56%
MI250x- ftn	58%	52%	-
MI250x-flang	20%	-	-
Intel T100	29%	-	-

Performance portability plot

✓ https://research-information.bris.ac.uk/ws/portalfiles/portal/250081850/Visualizing_Performance_Portability_07_09_2020.pdf

✓ Tr



- ✓ The choice of the FOM is crucial
- ✓ Find the correct code's A.I. could be not so easy
- ✓ With these performance figures is possible to
 - Check device performance evolution
 - Check software stack quality (i.e. compilers)
 - Check different parallel paradigm
 -

How compute A.I.

Some tool/references that can help....

- ✓ **Accelerating performance optimization for the NERSC-9 Perlmutter system. Concurrency Computat Pract Exper. 2020;32:e554**
 - <https://doi.org/10.1002/cpe.5547>
- ✓ **ERT (Empirical Roofline Tool)**
 - <https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/roofline/software/ert/>
- ✓ **Intel Advisor**
 - <https://www.intel.com/content/www/us/en/docs/advisor/tutorial-roofline/2021-1/roofline-use-case.html>
- ✓ **Nvidia Nsight**
 - <https://www.youtube.com/watch?v=fsC3QeZHM1U>
 - <https://developer.nvidia.com/blog/using-nsight-compute-to-inspect-your-kernels/>
 - <https://docs.nvidia.com/nsight-compute/2020.1/ProfilingGuide/index.html#roofline>
 - <https://www.youtube.com/watch?v=nYSdsJE2zMs&t=1166s>
- ✓ **Likwid**
 - <https://github.com/RRZE-HPC/likwid/wiki/Tutorial:-Empirical-Roofline-Model>
- ✓ **Cache Aware Roofline Model (CARM-Pop)**
 - <https://pop-coe.eu/blog/28th-pop-webinar-the-carm-tool-cache-aware-roofline-model-for-hpc>
- ✓ **OpenCL Benchmark**
 - <https://github.com/ProjectPhysX/OpenCL-Benchmark>

Thanks!

