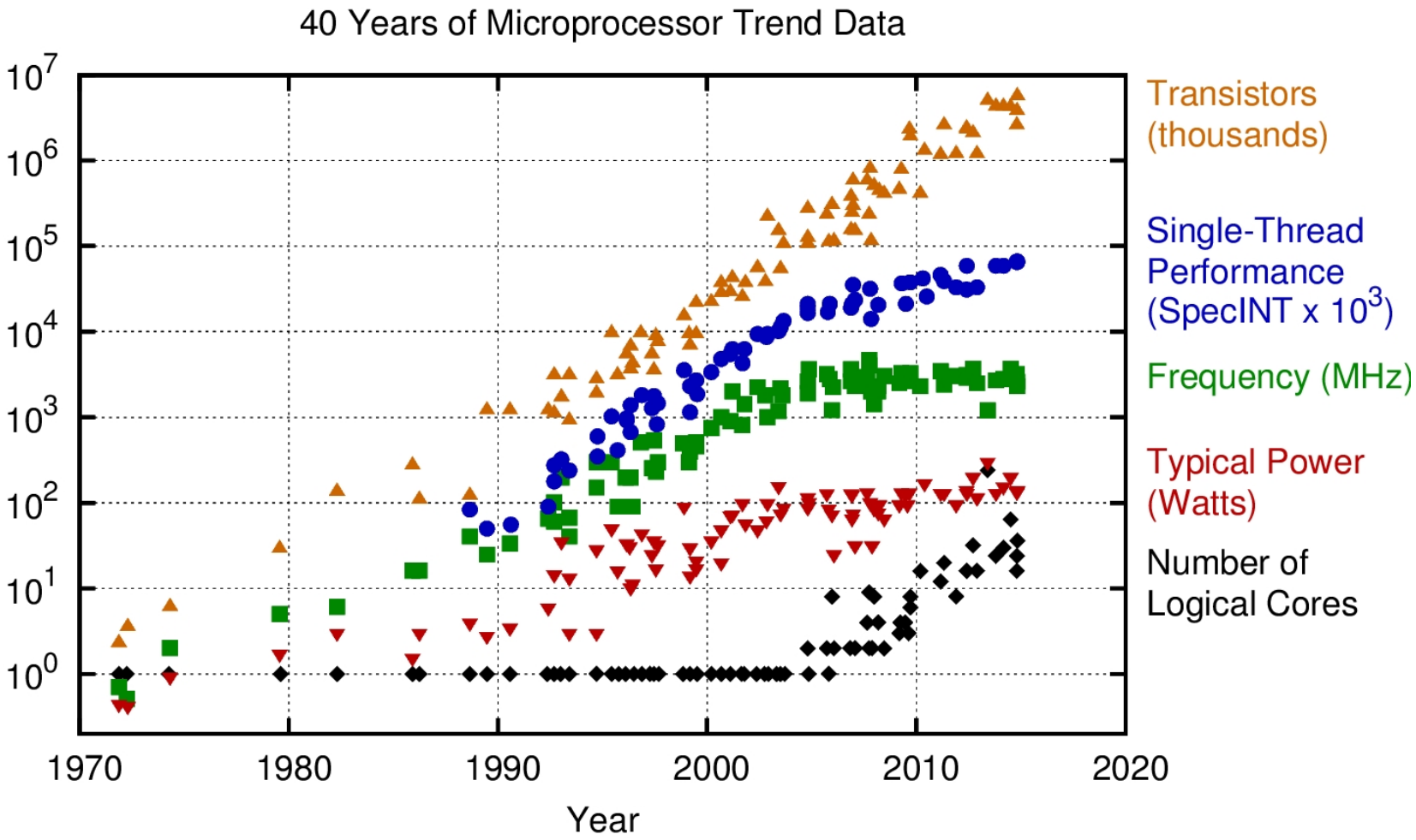
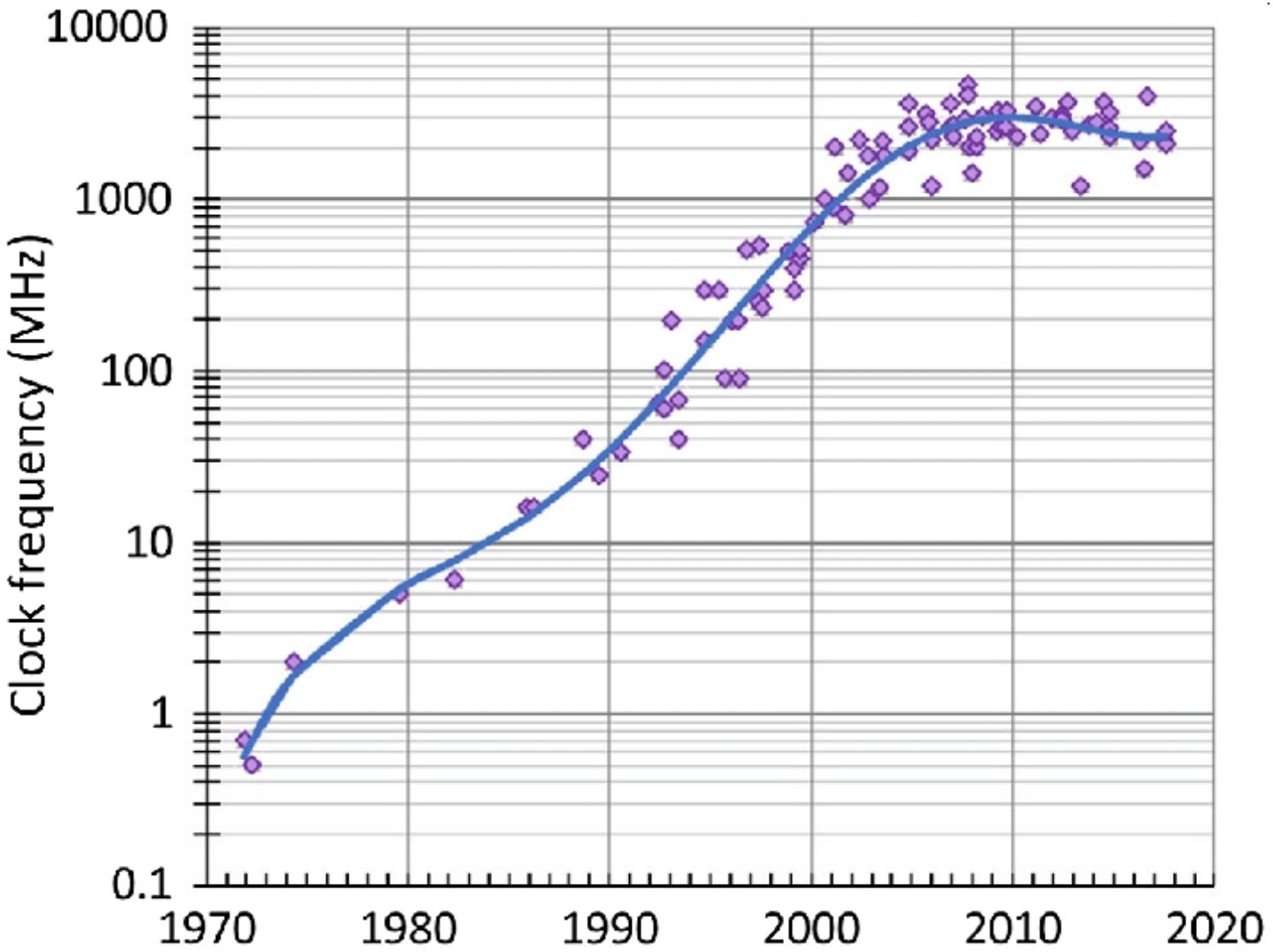
Częstotliwość zegara procesorów INTEL-a



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Częstotliwość zegara mikroprocesorów

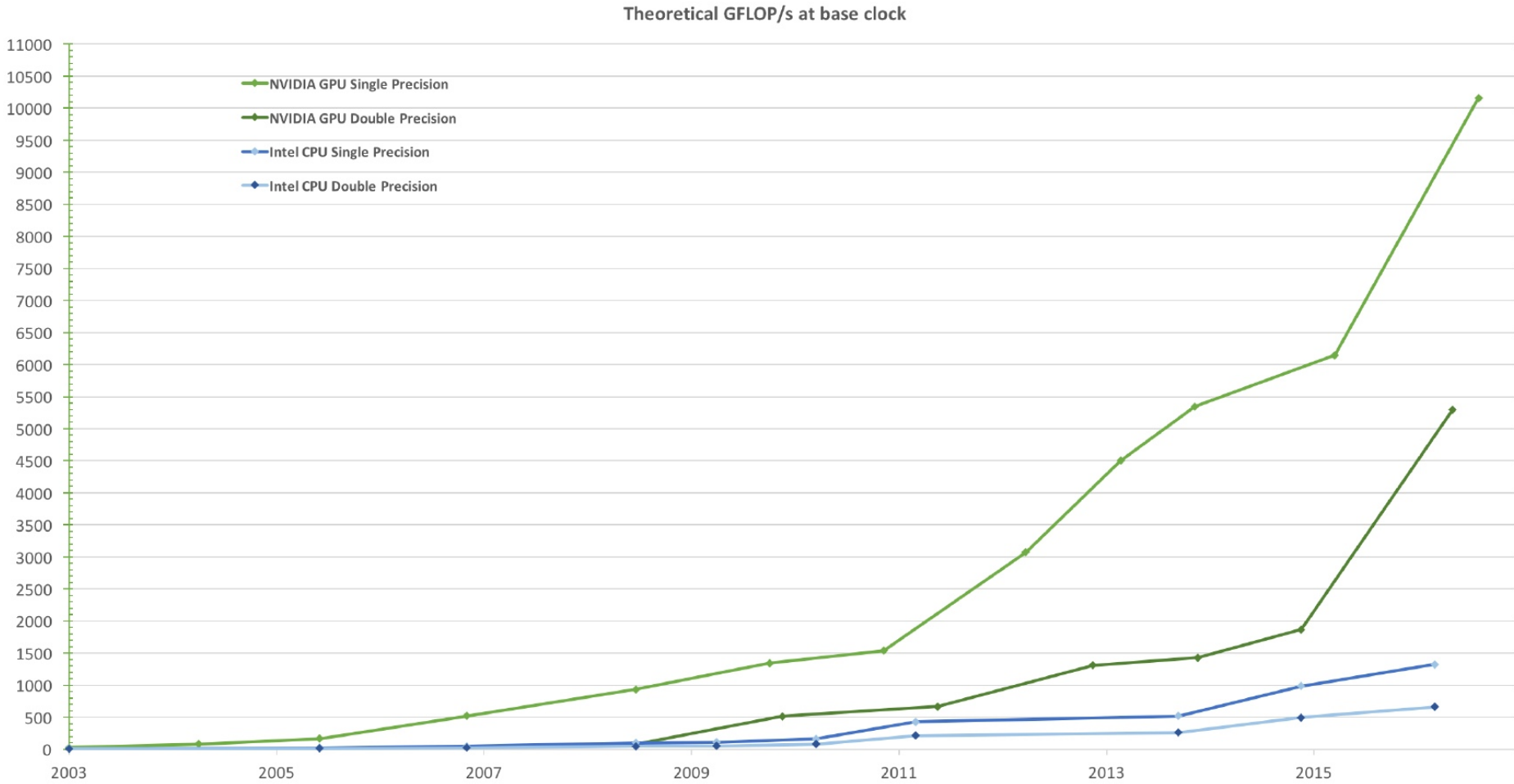


Michael L Rieger: *Retrospective on VLSI value scaling and lithography*
Journal of Micro/ Nanolithography, MEMS, and MOEMS · November 2019

Nasz bezpieczny świat legł w gruzach. . .

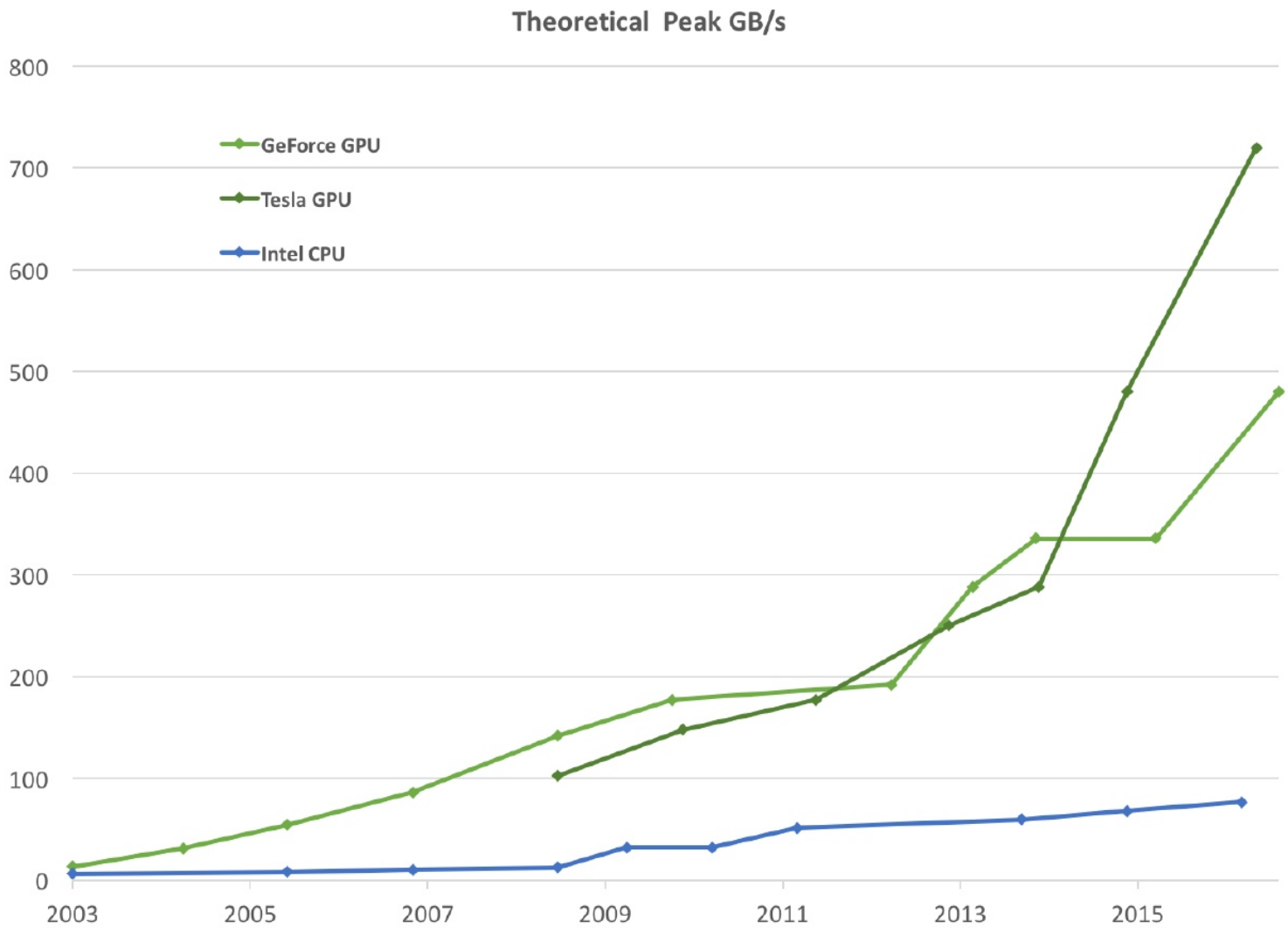
- W 2003 r. wzrost szybkości zegara procesorów w zasadzie się zatrzymał z powodu ograniczeń termicznych.
- Dalszy wzrost wydajności zgodnie z prawem Moore'a (moc obliczeniowa komputerów podwaja się co 24 miesiące) osiągnięto dodając do procesorów kolejne rdzenie.
- Takie podejście nie przyspiesza programów pisanych sekwencyjnie (wg podejścia von Neumanna z 1945 r.).
- Konieczna jest powszechna zmiana paradygmatu na niszowe dotąd programowanie równoległe.
- Dwie ścieżki rozwoju:
  - procesory kilkurdzeniowe (2, 4, 6, 8,. . .) – każdy rdzeń sprzętowo wspiera kilka „ciężkich" wątków $\longrightarrow$ CPU
  - procesory wielordzeniowe (nawet tysiące) – każdy sprzętowo wspiera tysiące „lekkich" wątków $\longrightarrow$ GPU
- Aplikacje numeryczne o dużej złożoności obliczeniowej implementowane są obecnie głównie na GPU. Dlaczego?

Dlaczego GPU? – Wydajność



CUDA C Programming Guide v.10.1, NVIDIA, July 2019

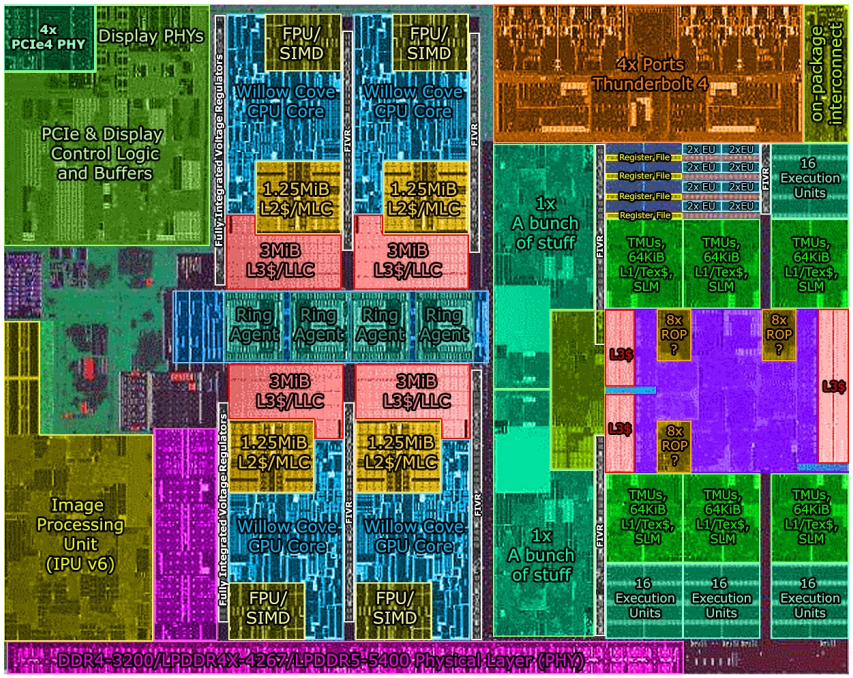
Dlaczego GPU? – Przepustowość



CUDA C Programming Guide v.10.1, NVIDIA, July 2019

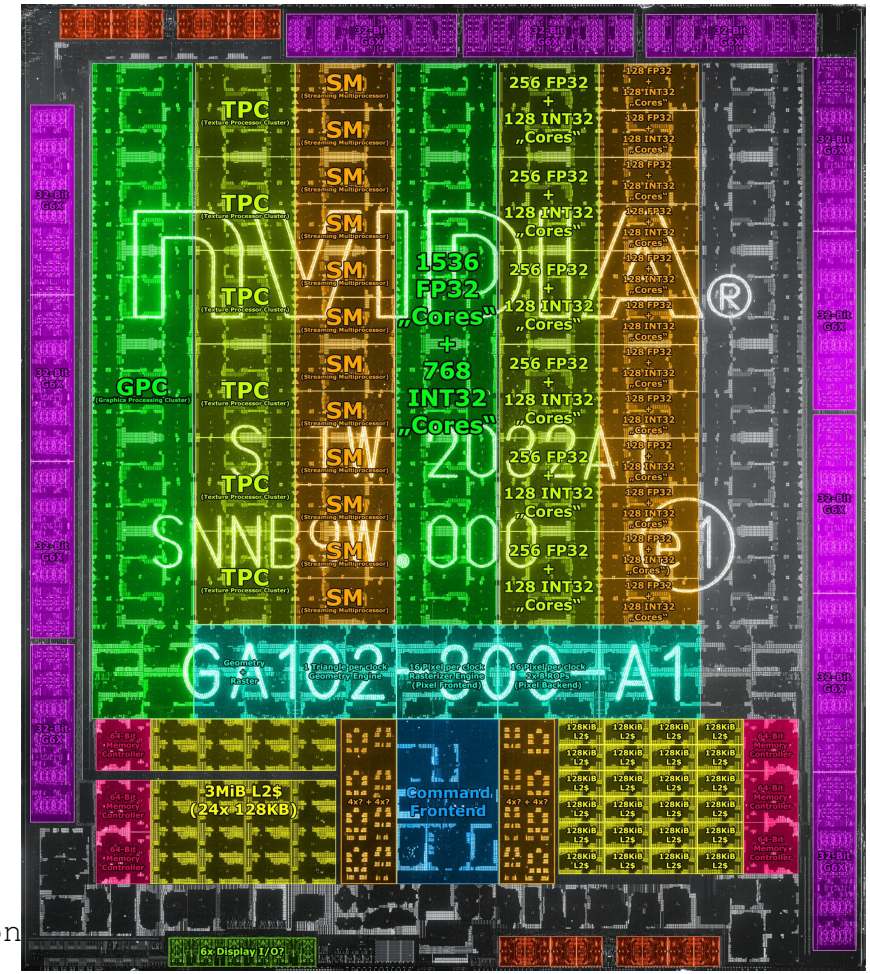
Dlaczego GPU? – Wykorzystanie krzemu

Intel Tiger Lake



www.reddit.com/r/intel/comments/kvtg6o/tiger_lakeu_and_rocket_lakes_die_shot_annotation

NVIDIA Ampere

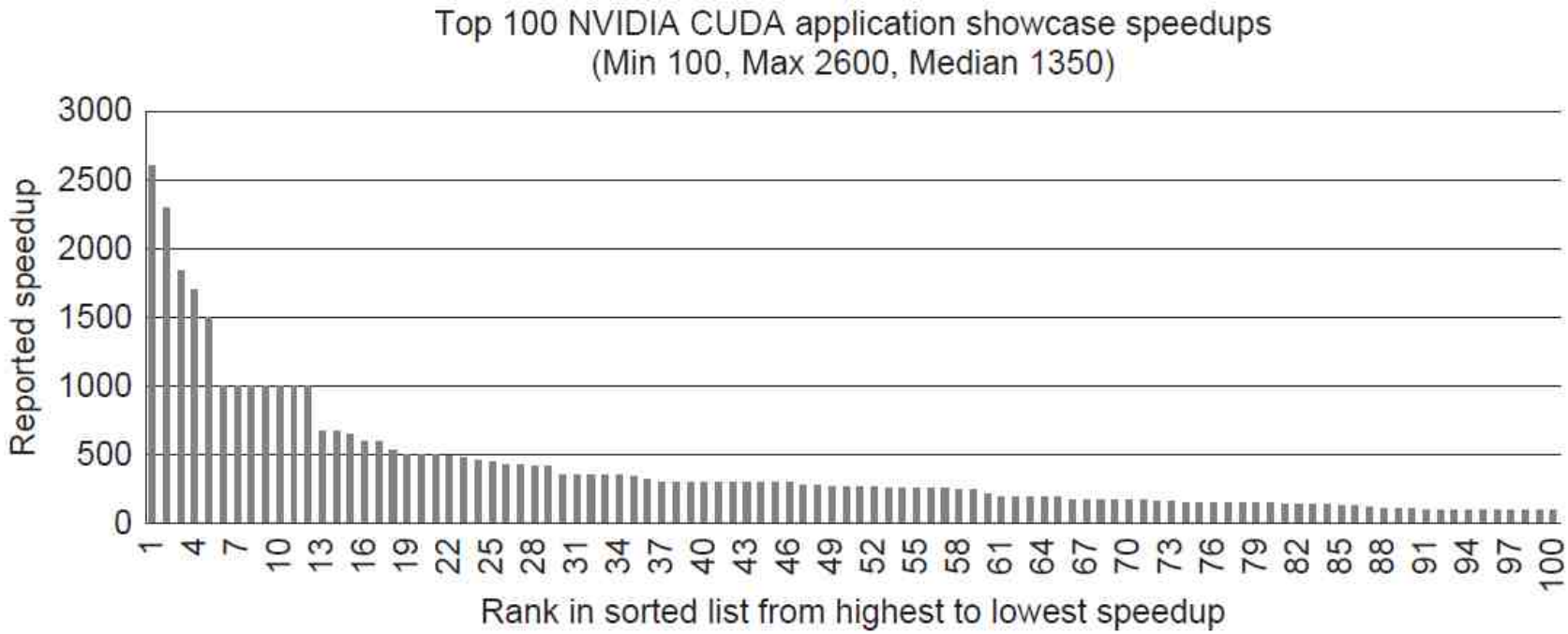


wccftech.com/nvidia-flagship-ampere-gaming-gpu-ga102-gets-beautiful-die-shot

Różnice między GPU a CPU

- Rdzenie CPU są zoptymalizowane do wykonywania kodu sekwencyjnego, więc mają dużo logiki sterującej do koordynowania wyników instrukcji wykonywanych równocześnie (*multi-issue*) i nie po kolei (*out of order*).
- CPU ma bardzo dużą pamięć podręczną (*cache*), aby zminimalizować opóźnienia w dostępie do pamięci.
- Historyczne uwarunkowania sposobu dostępu do pamięci przez CPU sprawiają, że pasmo tego dostępu jest niskie.
- Architektura GPU była optymalizowana pod kątem gier:
  - bardzo dużo prostych zmiennoprzecinkowych ALU, które grupowo wykonują te same proste operacje miliony razy
  - logika sterująca prostsza i wspólna dla całej grupy ALU
  - prostota operacji $\Longrightarrow$ bardzo „lekkie” wątki, więc może ich być bardzo dużo...
  - ...co ukrywa opóźnienia dostępu do pamięci – podczas oczekiwania uruchamiane są kolejne wątki $\Longrightarrow$ mała pamięć podręczna, przepustowość jest krytyczna

Dlaczego GPU? – Przyspieszenie



R. Farber: *CUDA Application Design and Development*. Morgan Kaufmann, 2011
Dane ze strony: `http://www.nvidia.co.uk/object/cuda_apps_flash_new_uk.html`

- Nie każdą aplikację da się znacznie przyspieszyć na GPU.
- Algorytmy sekwencyjne są lepiej wspierane przez CPU $\Longrightarrow$ konieczność rozwoju technologii heterogenicznych.
- Bez problemu można uzyskać przyspieszenie rzędu $10\times$. Słuchacz RIM powinien uzyskać $10^{(\text{ocena}-1)/2}\times$.

Współczynnik przyspieszenia (*speed-up*) S_u

Oznaczenia:

- $t(p, n)$ – czas wykonania algorytmu
- $t_s(n)$ – czas wykonania (przez jeden procesor) części sekwencyjnej algorytmu
- $t_p(n)$ – czas wykonania (przez jeden procesor) części algorytmu dającej się wykonać równolegle
- p – liczba procesorów wykonujących część równoległą
- n – rozmiar problemu (np. długość wektora danych)

Współczynnik przyspieszenia S_u...

...przy założeniu stałego rozmiaru problemu:

$$S_u = S_u(p, n) = \frac{t(1, n)}{t(p, n)} = \frac{t_s(n) + t_p(n)}{t_s(n) + t_p(n)/p}$$

Model *bardzo* uproszczony – pomijamy komunikację między procesorami, konflikty przy dostępie do pamięci itp.

Współczynnik przyspieszenia (*speed-up*) S_u – c.d.

Prawo Amdahla

$$S_u = \frac{t_s(n) + t_p(n)}{t_s(n) + t_p(n)/p} =$$

$$= \frac{1}{\underbrace{\frac{t_s(n)}{t_s(n) + t_p(n)}}_{1-\alpha} + \underbrace{\frac{t_p(n)}{t_s(n) + t_p(n)}}_{\alpha} \cdot \frac{1}{p}} = \frac{1}{1 - \alpha + \alpha/p},$$

gdzie $\alpha = \alpha(n)$ jest ułamkiem czasu wykonania części dającej się wykonać równolegle, liczonym dla pojedynczego procesora.

- $S_u < \lim_{\alpha \to 1^-} S_u = p$ – nie uzyskamy większego przyspieszenia niż liczba użytych procesorów.
- $S_u < \lim_{p \to \infty} S_u = \frac{1}{1-\alpha}$ – należy przede wszystkim starać się powiększyć część równoległą algorytmu ($\alpha \to 1^-$).
- Dla $p \to \infty$ ograniczeniem staje się pamięć (por. FFTW!).

Współczynnik przyspieszenia (*speed-up*) S_u – c.d.

Prawo Gustafsona–Barsisa

$$S_u = \frac{t_s(n) + t_p(n)}{t_s(n) + t_p(n)/p} =$$

$$= \underbrace{\frac{t_s(n)}{t_s(n) + t_p(n)/p}}_{1-\gamma} + \underbrace{\frac{t_p(n)/p}{t_s(n) + t_p(n)/p}}_{\gamma} \cdot p = 1 - \gamma + \gamma \cdot p,$$

gdzie $\gamma = \gamma(p, n)$ jest ułamkiem czasu wykonania części dającej się wykonać równolegle, liczonym dla wielu procesorów.

- W podejściu Amdahla ustalamy rozmiar problemu, a punktem odniesienia jest program sekwencyjny.
- W podejściu Gustafsona wraz ze wzrostem możliwości obliczeniowych ($p \nearrow$) zwiększamy rozmiar problemu ($n \nearrow$), utrzymując stały procent czasu części równoległej.
- Związek między podejściami: $\gamma(p, n) = \frac{\alpha(n)}{\alpha(n) + p - \alpha(n) \cdot p}$

Dlaczego GPU? – It's the economy, stupid! (Bill Clinton)

- Rozpowszechnienie: Na świecie są obecnie setki milionów GPU zdolnych do wykonywania obliczeń równoległych nie związanych z grafiką 3D.
- Cena: Dzięki dużemu rozpowszechnieniu karty GPU są tanie (w granicach 200–20000 zł). Dziś superkomputery często buduje się z procesorów GPU, bo jest to najtańsze rozwiązanie (#3, #6, #7, #8, #9, #10 w 10-ce TOP500'24).
- Praktyczność instalacji: Karty GPU (np. PCIe) są łatwe w instalacji sprzętowej i programowej.
- Standaryzacja liczb zmiennoprzecinkowych (IEEE 754).
- Łatwość programowania: Od 2007 r. po wprowadzeniu NVIDIA CUDA (*Compute Unified Device Architecture*), będącej zarówno modelem programowym, jak i wspierającą go architekturą sprzętową, nie trzeba już do programowania GPU wykorzystywać graficznego API, co było *bardzo* uciążliwe.

Graficzne API a CUDA

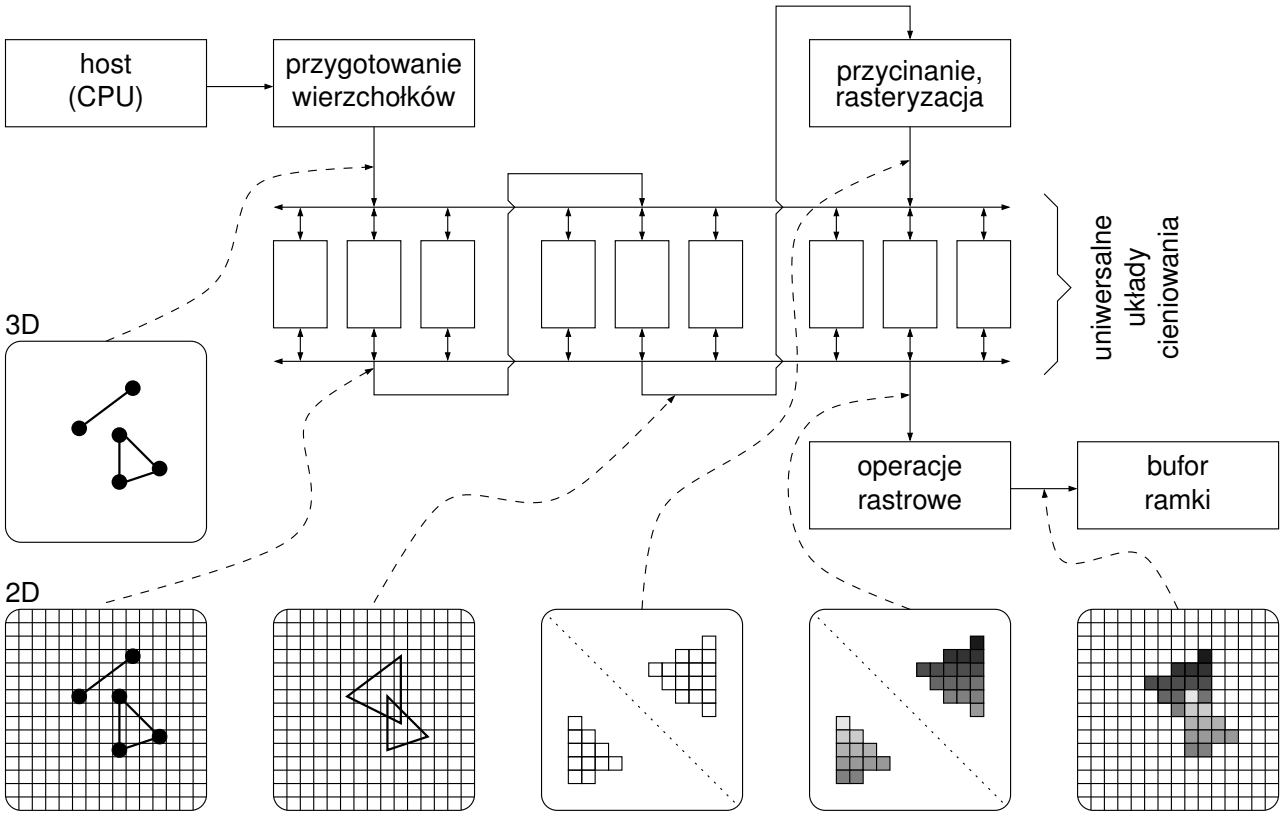
- Problemy z programowaniem obliczeń za pomocą graficznego API (tzw. podejście GPGPU – *General Purpose Computing on Graphics Processing Units*):
  - dane były przetwarzane jako tekstury na prostokącie
  - tekstury można było tylko czytać, zapis do bufora ramki
  - niemożność dostępu do danych pod *obliczonym* adresem
  - typy danych (np. 2-, 3- i 4-wektory) i ich precyzja (np. 16-bitowa mantysa) wymuszone przez sprzęt
- Osiągnięcia technologii CUDA:
  - duża i buforowana (*cache*) pamięć programu
  - pełna lista instrukcji, programowanie w uproszczonym C
  - logika sterująca wykonaniem programu – rozbudowana, ale dla oszczędności krzemu wspólna dla grupy rdzeni
  - swobodne adresowanie pamięci (obliczanym adresem)
  - model programowania wielką liczbą hierarchicznie zorganizowanych wątków, które mogą się synchronizować na barierze i komunikować przez pamięć współdzieloną
  - „przezroczysta” skalowalność i zgodność „w przód”

Języki i modele programowania równoległego

Jeśli oprogramowanie ma być wydajne, to nie może ukrywać przed programistą szczegółów implementacji sprzętowej.

- MPI – model klastra procesorów bez współdzielonej pamięci, każdy węzeł ma własną przestrzeń adresową, jawne przesyłanie danych komunikatami
- OpenMP – model systemu wieloprocesorowego ze współdzieloną pamięcią, wszystkie wątki mają wspólną przestrzeń adresową (o którą konkurują...)
- Fortran – model z pojedynczą przestrzenią adresową, ale **jawnie** podzieloną pomiędzy węzły (PGAS – *Partitioned Global Address Space*)
- CUDA – procesor wielordzeniowy (GPU) z hierarchicznie zorganizowaną współdzieloną pamięcią, w najnowszych wersjach implementuje pojedynczą przestrzeń adresową
- OpenCL – bardzo podobny do niskopoziomowego wariantu CUDA (tzw. *CUDA Driver API*), niezależny od platformy

Potok przetwarzania graficznego – historia



2006 – „macierz” zunifikowanych (CUDA) procesorów, odwiedzana 3× w ramach potoku graficznego, pomiędzy ustalonymi sprzętowo funkcjami (GeForce 8800)

2007 – wprowadzenie narzędzi i „ekosystemu” CUDA

Generacje GPU zgodnych z technologią CUDA

Rok	C_c	Wybrane procesory GPU	Rodzina
2007	1.0 / 1.1	80 / 84, 86, 92, 94, 96, 98	— (G)
2009	1.2	215, 216, 218	Tesla (GT)
2008	1.3	200, 200b	
2010	2.0 / 2.1	100, 110 / 104, 106, 108, 114, 116, 119	Fermi (GF)
2012	3.0	104, 106, 107	Kepler (GK)
2013	3.5	110, 208	
2014	3.7	210	
2014	5.0 / 5.2	107, 108 / 200, 204, 206	Maxwell (GM)
2016	6.0 / 6.1	100 / 102, 104, 106, 107, 108	Pascal (GP)
2017	7.0	100	Volta (GV)
2018	7.5	102, 104, 106, 116. 117	Turing (TU)
2020	8.0	100	Ampere (GA)
2021	8.6	102, 103, 104, 106, 107	
2022	8.9	102, 103, 104, 106, 107	Ada Lovelace (AD)
2022	9.0	100	Hopper (GH)
2024	*10.0*	*100*	*Blackwell (GB)*

C_c – możliwości obliczeniowe (*compute capability*) $\rightarrow$ W2.

Lab.: GeForce GTX 1650 (C_c=7.5, 896 rdzeni CUDA, 4 GB).