Humanidades, Tecnología y Ciencia, del Instituto Politécnico Nacional

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

VENTAJAS DEL PROCESAMIENTO VECTORIAL CON GPUs BAJO EL MODELO SIMT

Israel Rivera Zárate
Instituto Politécnico Nacional-CIDETEC
irivera@ipn.mx
ORCID: https://orcid.org/0000-0002-5254-120X

Patricia Pérez Romero
Instituto Politécnico Nacional-CIDETEC
promerop@ipn.mx
ORCID: https://orcid.org/0000-0003-3395-6239

Miguel Hernández Bolaños
Instituto Politécnico Nacional-CIDETEC
mbolanos@ipn.mx
ORCID: https://orcid.org/0000-0002-5622-8747

Abstract

GPU computing as a general-purpose programming model did not emerge fully formed: it required tools that could bridge the gap between raw hardware parallelism and practical programmability. This paper focuses on the first two platforms to attempt that bridge, CUDA 1.0 (NVIDIA, 2007) and CTM/Brook+ (ATI/AMD, 2007–2008), comparing their architectural assumptions, programming models, and reported performance. The analysis draws on official manufacturer documentation and peer-reviewed literature covering the period 2001–2023. Benchmarks from the period show speedups of 10× to 50× over conventional CPUs on data-parallel workloads, though these figures are specific to optimized kernels produced under manufacturer-controlled conditions and may not generalize to arbitrary workloads. CUDA's advantage lay in its accessibility: by extending standard C, it allowed developers to write parallel code without knowledge of graphics-pipeline internals. CTM/Brook+, while less accessible, foreshadowed the architectural direction AMD would later consolidate in its ROCm/HIP ecosystem. The principles underlying both platforms—SIMT execution, a hierarchical memory model, and explicit data parallelism—remain the structural foundation of every major GPU computing framework today, including deep-learning frameworks such as PyTorch and JAX.
Keywords: CUDA; CTM; GPU; GPGPU, vector processing.

Un procesador vectorial es un diseño de CPU capaz de ejecutar operaciones matemáticas sobre múltiples elementos de datos de forma simultánea, a diferencia de los procesadores escalares, que solo pueden manejar un dato a la vez. La gran mayoría de

las CPU actuales son escalares o superescalares, con extensiones vectoriales limitadas. Los procesadores vectoriales fueron durante décadas la base de las supercomputadoras científicas, especialmente durante los años ochenta y noventa. Sin embargo, la aparición de arquitecturas masivamente paralelas y el auge de las GPU como plataformas de cómputo de propósito general han revitalizado el paradigma vectorial de una forma radicalmente distinta.

Casi todas las CPU actuales incluyen conjuntos de instrucciones de procesamiento vectorial conocidos como SIMD (Single Instruction, Multiple Data), siguiendo la taxonomía propuesta por Flynn (1972). No obstante, las GPU llevan este concepto a un nivel de paralelismo masivo que las CPU convencionales no pueden igualar. Esta capacidad resulta de particular relevancia para la ingeniería de procesos y la simulación científica, áreas que demandan cómputo intensivo de alta precisión. Trabajos como el de Egri et al. (2007) en física de partículas fueron de los primeros en documentar sistemáticamente este potencial.

Hasta 2007, aprovechar ese potencial obligaba a programar en ensamblador gráfico o a adaptar los cálculos a través de interfaces diseñadas para renderizado de imágenes, no para álgebra numérica. CUDA (Compute Unified Device Architecture) de NVIDIA y CTM (Close to the Metal) de ATI respondieron a ese problema desde perspectivas muy distintas. CUDA redujo el esfuerzo de entrada al programador: bastaba conocer el lenguaje C para escribir código paralelo usando unas pocas extensiones. CTM, en cambio, eliminó toda capa intermedia y dio acceso directo al funcionamiento interno de la tarjeta, con mayor control pero también mayor complejidad. Brook+ de AMD buscó un punto intermedio entre ambos enfoques, aunque con resultados que no llegaron a consolidarse del todo en el período estudiado.

Este artículo tiene como objetivo caracterizar y comparar analíticamente las arquitecturas de procesamiento vectorial implementadas en GPU, con énfasis en las plataformas de programación CUDA y CTM/Brook+ en sus versiones fundacionales (2007–2008). Los trabajos de Owens et al. (2008) y Nickolls et al. (2008) constituyen los textos de referencia que definieron esta etapa; los manuales oficiales de NVIDIA (2007) y ATI Technologies (2007), las fuentes primarias indispensables. El trabajo se estructura en cuatro apartados: fundamentos del procesamiento vectorial; la GPU como procesador paralelo; herramientas de software CUDA y CTM; y el ecosistema GPGPU contemporáneo.

La revisión se basa en documentación técnica de los fabricantes y en publicaciones arbitradas del período 2001–2023, con preferencia por fuentes con DOI disponibles en IEEE Xplore, ACM Digital Library y Scopus. Los datos de rendimiento se presentan tal como fueron reportados en su momento, incluyendo las aclaraciones necesarias sobre las condiciones de medición y los límites de generalización de esos resultados (Owens et al., 2008).

2. Procesamiento vectorial

Un procesador vectorial trabaja con *vectores* —grupos de valores numéricos del mismo tipo almacenados en posiciones de memoria continuas— como unidad básica de cómputo. En lugar de procesar un elemento a la vez, aplica una sola instrucción a todos los elementos del grupo a la vez, lo que reduce considerablemente el trabajo de control del procesador. La transformación de código que opera elemento por elemento en código que aprovecha esta capacidad se llama *vectorización*, y determina en gran medida qué

parte del programa puede ejecutarse más rápido.

Las operaciones de un procesador vectorial pueden clasificarse en cinco tipos (donde V representa un espacio vectorial y K un valor escalar): aplicar una función a cada elemento de un vector ($f_1 : V \rightarrow V$), combinar dos vectores elemento a elemento ($f_2 : V \times V \rightarrow V$), reducir un vector a un solo valor ($f_3 : V \rightarrow K$), combinar dos vectores en un solo valor ($f_4 : V \times V \rightarrow K$) y multiplicar un vector por un número escalar ($f_5 : K \times V \rightarrow V$). La Tabla 1 presenta ejemplos de cada tipo. Dentro de f_2 destacan las operaciones de empaquetamiento y desempaquetado, útiles cuando se trabaja con matrices dispersas.

Tabla 1. Ejemplos de operadores vectoriales

Operador	Expresión	Tipo
$c = a + b$	$c_i = a_i + b_i$	Vectorial binaria (f_2)
$s = \Sigma_{i=1}^{n} a_i$, $s \in K$	Suma de componentes	Reducción unitaria (f_3)
$c = \alpha \cdot a$	$c_i = \alpha \cdot a_i$	Escalado (f_5)
pack(a, mask)	Compacta no nulos	Empaquetamiento

Nota. V = espacio vectorial; K = cuerpo escalar; α ∈ K.

2.1 Ventajas del procesamiento vectorial sobre el escalar

Cuatro características distinguen la ejecución vectorial de la escalar. La primera es la **independencia entre resultados**: como los elementos de un vector no dependen entre sí, el hardware puede calcularlos en paralelo sin conflictos de datos. La segunda es la **reducción de instrucciones**: una sola operación vectorial reemplaza a un ciclo con N instrucciones individuales. La tercera es la **regularidad en los accesos a memoria**: los accesos contiguos aprovechan mejor el caché. La cuarta es la **eliminación de saltos de control**: sin ciclo explícito, el procesador no necesita predecir si el bucle continúa o no.

2.2 Ejemplo: programación escalar vs. vectorial en GPU

Para ilustrar las ventajas descritas, considérese la suma de vectores —operación f_2— en dos implementaciones. El Listado 1 muestra la versión escalar en C estándar; el Listado 2 muestra su equivalente en CUDA bajo el modelo SIMT (Single Instruction, Multiple Threads), donde cada hilo procesa un elemento de forma simultánea (Nickolls et al., 2008).

Listado 1. Suma de vectores — C escalar:

```
void addVectors(float *a, float *b,
               float *c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Listado 2.

```
__global__ void addVectors(
    float *a, float *b,
    float *c, int N) {
    int i = blockIdx.x * blockDim.x
          + threadIdx.x;
    if (i < N)
        c[i] = a[i] + b[i];
}
// Invocación:
// addVectors<<<(N + 255) / 256, 256>>>(
//   d_a, d_b, d_c, N);
```

En el Listado 2, la GPU lanza N hilos simultáneos en bloques de 256, eliminando el ciclo y distribuyendo el trabajo aritmético entre miles de unidades de cálculo (Nickolls et al., 2008).

3. La GPU como procesador de datos paralelo

La GPU llegó al ámbito del cómputo científico por una razón práctica: la industria de los videojuegos había invertido durante décadas en hardware con un nivel de paralelismo que difícilmente habría podido financiar la investigación académica (Vlachos & Peters, 2001). Cuando Egri et al. (2007) usaron ese hardware en simulaciones de física de partículas, estaban aprovechando para la ciencia una capacidad construida con otros

HCT Humanidades, Tecnología y Ciencia,
del Instituto Politécnico Nacional
Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

fines. La Figura 1 muestra un comparativo representativo de las diferencias de velocidad entre ambos procesadores.

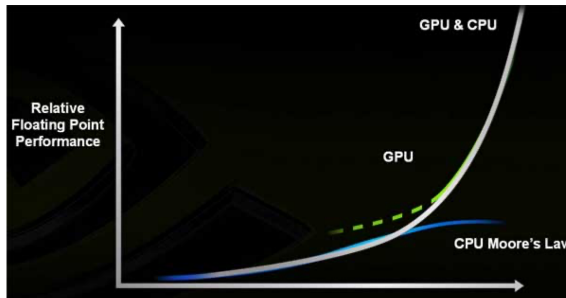

Figura 1. *Comparativo de velocidad de procesamiento entre GPU y CPU convencional. Fuente: Egri et al. (2007).*

La diferencia entre una GPU y un procesador convencional es de prioridades: mientras el procesador dedica gran parte de sus recursos a caché, predicción de instrucciones y ejecución anticipada, la GPU los dedica a unidades de cálculo. La Figura 2 ilustra este potencial con un ejemplo de renderizado 3D de la época. Cuando un grupo de hilos de la GPU espera datos de memoria, otro grupo aprovecha ese tiempo para calcular, lo que mantiene las unidades aritméticas ocupadas sin necesitar grandes memorias caché.

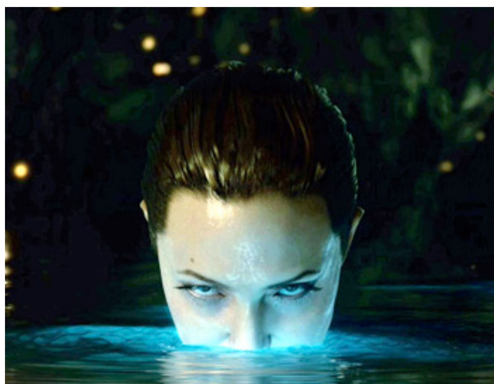

Figura 2. *Ejemplo del poder de renderizado 3D con GPU: fotograma de Beowulf (Paramount Pictures, 2007), que ilustra la capacidad de procesamiento paralelo de las GPU de la época.*

Sin embargo, antes de 2007 esta arquitectura tenía limitaciones importantes. Como se aprecia en la Figura 3, la GPU concentra la mayoría de sus transistores en unidades aritméticas a diferencia del CPU, lo que explica tanto su fortaleza para operaciones paralelas como sus restricciones: no era posible escribir en posiciones de memoria arbitrarias; programar la GPU requería conocer el detalle interno del proceso de renderizado; y ciertos patrones de acceso a los datos generaban cuellos de botella que anulaban las ventajas del paralelismo. Resolver estos tres problemas fue precisamente lo que CUDA y CTM intentaron hacer.

Figura 3. *Distribución de transistores en GPU y CPU: la GPU asigna la mayor parte al procesamiento aritmético paralelo, mientras la CPU los destina a caché y unidades de control de flujo. Fuente: NVIDIA Corporation (2007).*

4. Software para desarrollo en GPU

4.1 CUDA (Compute Unified Device Architecture)

CUDA representa, al mismo tiempo, una decisión técnica y una decisión estratégica. La decisión técnica fue hacer accesible la GPU a través de extensiones al lenguaje C, de manera que un programador con experiencia en ese lenguaje pudiera escribir código paralelo sin necesidad de aprender desde cero cómo funciona una tarjeta gráfica por dentro. La decisión estratégica fue privilegiar la facilidad de uso sobre el control máximo del hardware.

En términos técnicos, CUDA (NVIDIA Corporation, 2007) se organiza en tres capas, tal como muestra la Figura 4: un controlador

Humanidades, Tecnología y Ciencia, del Instituto Politécnico Nacional
Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

de hardware de bajo nivel, una interfaz de programación (*runtime*), y dos bibliotecas matemáticas especializadas —CUFFT para transformadas de Fourier y CUBLAS para álgebra lineal—. El código de un programa CUDA se divide en tres partes: la que corre en el procesador principal (*host*), la que corre en la tarjeta gráfica con las funciones paralelas (*kernels*), y una parte compartida con tipos de datos y funciones comunes. Todo el código se compila con la herramienta *nvcc*.

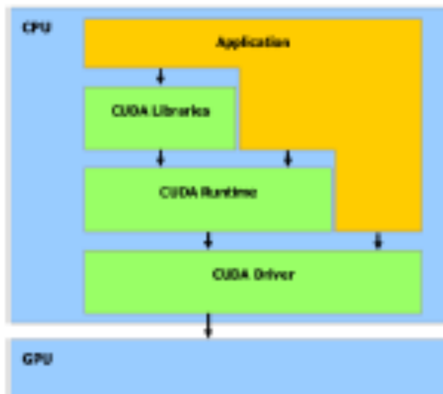

Figura 4. *Arquitectura de software de CUDA: modelo de capas con controlador de hardware, runtime y bibliotecas CUFFT y CUBLAS. Fuente: NVIDIA Corporation (2007).*

Un avance técnico importante de CUDA fue permitir el acceso directo y flexible a la memoria de la tarjeta gráfica, algo que no era posible antes de 2007. Esto habilitó operaciones de lectura y escritura en posiciones no consecutivas de memoria (*gather* y *scatter*) y la posibilidad de usar una porción de la memoria como caché compartida bajo control del programador. En modo coprocesador —un ejemplo típico se ilustra en la Figura 5—, la GPU podía procesar bloques de datos de 64 bits con hasta 256 tareas en paralelo mientras el procesador principal se ocupaba de otras tareas.

Figura 5. *Uso de GPU como coprocesador en entorno de escritorio: Compiz-Fusion sobre Ubuntu Linux delega trabajo computacional a la GPU, liberando recursos del CPU.*

Tabla 2. Número de multiprocesadores por modelo NVIDIA (2007)

Modelo GPU	Multiprocesadores
GeForce 8800 GTX	16
GeForce 8800 GTS	12
GeForce 8600 GTS	4
GeForce 8600 GT	4
GeForce 8500 GT	2

Nota. Fuente: NVIDIA Corporation (2007). CUDA Programming Guide v1.0.

4.2 Close to the Metal (CTM)

Mientras CUDA apostaba por facilitar el acceso, CTM apostaba por lo contrario. ATI Technologies (2007) diseñó CTM para dar acceso directo a las capacidades de cálculo de punto flotante de su hardware, sin capas de por medio. El resultado podía ser más eficiente, pero exigía al programador conocer la arquitectura interna de la tarjeta con un nivel de detalle que la mayoría no estaba dispuesta a aprender.

El modelo de CTM se organiza alrededor de tres unidades. El procesador de datos paralelos (DPP) agrupa los procesadores programables; la unidad de ejecución de procesos (PE) asigna a cada procesador coordenadas (i, j) dentro de la cuadrícula de cómputo; y la unidad controladora de memoria

(MC) gestiona dos espacios de memoria separados: uno privado y uno remoto compartido con el procesador principal.

La unidad de operación condicional (CO) completa el esquema: evalúa condiciones para PE y DPP, y si la condición se cumple, escribe el resultado en el buffer de salida. La Figura 6 muestra el diagrama completo de este bloque funcional. Este mecanismo permite cierto control de flujo dentro del modelo SIMD, aunque con menos flexibilidad que la que CUDA ofrecería con el modelo SIMT.

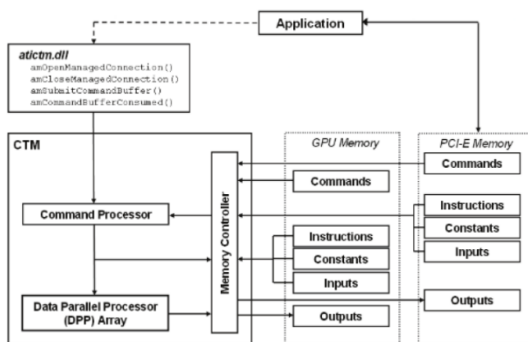

Figura 6. *Diagrama del bloque funcional de CTM: el procesador de datos paralelos (DPP) integra las unidades PE, CO y MC. Fuente: ATI Technologies (2007).*

4.3 AMD FireStream y Brook+: la evolución de CTM

La misma razón que hacía a CTM poderoso lo limitaba como plataforma de uso masivo: programar en ensamblador de GPU es técnicamente exigente. AMD reconoció esta limitación y respondió con Brook+ (AMD, 2007), un lenguaje de alto nivel basado en C, derivado del proyecto Brook de la Universidad de Stanford, al que se añadieron extensiones propias y una capa de compatibilidad llamada CAL (Compute Abstraction Layer), diseñada para proteger al programador de los cambios entre generaciones de hardware.

La tarjeta de referencia para esta plataforma fue la AMD FireStream 9170 (Figura 7): 320 procesadores de *stream*, 2 GB de memoria DDR3 de 128 bits, un rendimiento máximo de 500 gigaflops en precisión simple y un consumo inferior a 150 W. Su característica más relevante para la comunidad científica era ser el primer producto GPGPU con soporte de hardware para aritmética de doble precisión (hasta 15–16 dígitos significativos), una capacidad que la GeForce 8800 de NVIDIA no ofrecía. En Supercomputing'07, AMD presentó resultados que mostraban aceleraciones de 10× a 50× respecto a procesadores convencionales en kernels específicamente optimizados; estas cifras no son generalizables a cualquier tipo de aplicación (Owens et al., 2008).

Figura 7. *Tarjeta AMD FireStream 9170: primer producto GPGPU con soporte de hardware para aritmética de doble precisión, 320 núcleos stream y 500 gigaflops de rendimiento pico en precisión simple. Fuente: AMD (2007).*

5. Ecosistema GPGPU contemporáneo

Una manera de medir el impacto de las decisiones de 2007 es observar qué ha permanecido sin cambios esenciales casi veinte años después. El modelo SIMT, la memoria compartida jerárquica y el paralelismo de datos explícito que CUDA introdujo siguen siendo la base sobre la que se construyen las arquitecturas más avanzadas.

Las GPUs Ampere y Hopper de NVIDIA (2020–2022) incorporaron núcleos especializados para operaciones de inteligencia artificial capaces de alcanzar hasta 3,958 TFLOPS en precisión FP8, memoria de alta capacidad con más de 3 TB/s de velocidad de transferencia, e interconexiones NVLink (NVIDIA Corporation, 2022).

AMD dividió formalmente su línea de GPUs en gráficas (RDNA) y de cómputo (CDNA). La GPU Instinct MI300X integra procesador y tarjeta gráfica en un mismo chip con memoria HBM3 y 1.3 TB/s de velocidad de transferencia (AMD, 2023). HIP de AMD ofrece compatibilidad con el código escrito para CUDA, de modo que puede migrarse con cambios mínimos. OpenCL 3.0 y SYCL del grupo Khronos (2023) amplían el alcance a entornos heterogéneos con tarjetas de distintos fabricantes.

TensorFlow, PyTorch (Paszke et al., 2019) y JAX permiten entrenar modelos con miles de millones de parámetros sin escribir una sola línea de CUDA. Aplicaciones como AlphaFold (Jumper et al., 2021), la dinámica molecular y la fluidodinámica computacional dependen de esa capa de abstracción que, a su vez, se apoya en los mismos fundamentos arquitectónicos descritos en los manuales de 2007.

Tabla 3. Comparativa entre plataformas GPGPU: iniciales, intermedias y contemporáneas

Dimensión	CUDA 1.0 (2007)	CTM/Brook+ (2007–08)	OpenCL 1.2 (2011)	CUDA 12.x/HIP (2024)
Abstracción	Alto — ext. C	Bajo (ASM) / Medio (Brook+)	Medio — C + ext.	Alto — C++17
Acceso DRAM	Sí (gather/scatter)	Restr. (CTM) / Parcial (Brook+)	Sí, vía buffers	Sí, unificado CPU-GPU
Doble precisión HW	No (GeForce 8800)	Sí (FireStream 9170)	Sí (estándar)	Sí + FP8 Tensor Cores
Modelo de hilos	SIMT — bloques/grillas	SIMD — grilla 2D	NDRange	SIMT + coop. groups
Modelo de memoria	Reg./shared/global	Privado/remoto	Priv./local/global	UVM + HBM3
Bibliotecas matemáticas	CUFFT, CUBLAS	FFT, MatMul (Brook+)	clBLAS, clFFT	cuDNN, rocBLAS
Aceleración rep.†	10×–50×	10×–50×	Variable	3,958 TFLOPS FP8
Curva aprendizaje	Baja (C estándar)	Muy alta / Media	Media-alta	Media (doc. extensa)

Nota. † Benchmarks de fabricante sobre kernels optimizados; no generalizables. Elaboración propia a partir de NVIDIA Corporation (2007, 2022), ATI Technologies (2007), AMD (2007, 2023) y Khronos Group (2023).

6. Discusión

La comparación entre CUDA 1.0 y CTM/Brook+ no es, en esencia, una comparación técnica: es una comparación entre dos ideas sobre cómo se adopta la innovación. ATI apostaba por ofrecer el mejor rendimiento posible; NVIDIA apostaba por que los programadores adoptaran la plataforma primero y la optimizaran después. Owens et al. (2008) identificaron la diferencia clave: CUDA ofrece dos niveles de paralelismo —de datos y de hilos— con una jerarquía de memoria completa, mientras que Brook+ solo trabajaba con el paralelismo de datos del modelo de *streaming* de Stanford. Esta diferencia de fondo fue lo que determinó qué plataforma tenía más potencial de crecimiento.

Nickolls et al. (2008) documentaron la ventaja práctica de CUDA: un programador que conocía C podía convertir código secuencial en código paralelo eficiente con cambios relativamente pequeños. CTM pedía algo cualitativamente distinto: aprender el ensamblador específico de la tarjeta gráfica. Brook+ intentó cubrir ese hueco, pero su compilador *brcc* generaba código sistemáticamente menos eficiente que el de *nvcc* en las comparaciones del período (Owens et al., 2008).

Las aceleraciones de 10× a 50× reportadas por AMD (2007) en Supercomputing'07 son reales, pero aplican a situaciones muy específicas: programas con alto grado de paralelismo de datos, diseñados para sacar el máximo provecho del hardware del fabricante, bajo condiciones controladas. La variación de los resultados según el tipo de algoritmo, el patrón de acceso a memoria y el grado de divergencia en el flujo de control puede ser muy grande (Owens et al., 2008). La ventaja de FireStream 9170 en doble precisión era relevante para simulación científica, pero no compensó la brecha de accesibilidad frente a CUDA.

El dato más interesante de esta comparación es de continuidad: la arquitectura SIMT y la organización jerárquica de la memoria que NVIDIA diseñó en 2007 se encuentran, en sus principios esenciales, en CUDA 12.x, HIP/ROCm y SYCL. La Tabla 3 permite seguir esta evolución. Desde el punto de vista del ecosistema, la adopción masiva de CUDA generó un efecto de acumulación: bibliotecas, tutoriales, herramientas de depuración y marcos como PyTorch (Paszke et al., 2019) y JAX fueron desarrollados tomando CUDA como base. Cuando AMD buscó recuperar terreno con HIP/ROCm, el reto no era de compatibilidad de código sino de todo ese ecosistema acumulado.

7. Conclusiones

CUDA 1.0 y CTM/Brook+ representaron dos respuestas distintas al mismo desafío: cómo hacer que el paralelismo masivo de la GPU fuera accesible para programadores que no son especialistas en gráficos. CUDA resolvió ese problema facilitando el acceso; CTM priorizó el control total del hardware; Brook+ intentó un punto intermedio que no terminó de madurar en el período analizado.

La facilidad de uso fue un factor más decisivo que las características técnicas específicas. CUDA no ganó por ser la más rápida en las pruebas de 2007 —FireStream 9170 la superaba en precisión de doble cálculo—, sino porque permitió que un grupo mucho más amplio de programadores pudiera utilizarla. Esa misma razón explica por qué HIP/ROCm de AMD no ha logrado desplazar a CUDA: la compatibilidad de código no es suficiente cuando el otro ecosistema lleva años acumulando bibliotecas, herramientas y comunidad de usuarios.

Los principios de diseño que surgieron en ese período —modelo SIMT, memoria compartida organizada en niveles, paralelismo de datos explícito— son hoy la base sobre la que funcionan PyTorch (Paszke et al., 2019), JAX y cualquier marco de aprendizaje profundo de uso extendido. Para quienes trabajan en simulación numérica, bioinformática o aprendizaje automático, entender estos fundamentos no es un ejercicio histórico: es una condición para comprender por qué el hardware se comporta como lo hace y cuáles son los límites reales de la aceleración.

Una limitación de este análisis que vale la pena mencionar es que los datos de rendimiento provienen principalmente de pruebas publicadas por los propios fabricantes. Los resultados obtenidos en condiciones optimizadas de laboratorio no necesariamente reflejan el comportamiento promedio en aplicaciones reales. Trabajos futuros podrían incorporar evaluaciones independientes y ampliar la comparación hacia plataformas posteriores —OpenCL 3.0, CUDA 12.x, entornos de cómputo en la nube— para trazar con mayor detalle la trayectoria de decisiones que ha dado forma al ecosistema actual.

Referencias

AMD. (2007). *Brook+ SDK: Open-source stream computing*. Advanced Micro Devices.

AMD. (2023). *AMD Instinct MI300X accelerator*. Advanced Micro Devices. https://www.amd.com/en/products/accelerators/instinct/mi300/mi300x.html

ATI Technologies. (2007). *ATI CTM guide: Close to the metal*. Advanced Micro Devices. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

Egri, G. I., Fodor, Z., Hoelbling, C., Katz, S. D., Krieg, S., & Szabo, K. K. (2007). Lattice simulations on graphics cards. *Computer Physics Communications*, *177*(8), 631–639. https://doi.org/10.1016/j.cpc.2007.06.005

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, *C-21*(9), 948–960. https://doi.org/10.1109/TC.1972.5009071

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., & Hassabis, D. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, *596*(7873), 583–589. https://doi.org/10.1038/s41586-021-03819-2

Khronos Group. (2023). *OpenCL 3.0 reference guide*. https://www.khronos.org/opencl/

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue*, *6*(2), 40–53. https://doi.org/10.1145/1365490.1365500

NVIDIA Corporation. (2007). *NVIDIA CUDA programming guide, version 1.0*. https://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

NVIDIA Corporation. (2022). *NVIDIA H100 Tensor Core GPU architecture* (Technical White Paper). https://resources.nvidia.com/en-us-tensor-core

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, *96*(5), 879–899. https://doi.org/10.1109/JPROC.2008.917757

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*. https://doi.org/10.48550/arXiv.1912.01703

Strong, J. P. (1991). Computations on the massively parallel processor at the Goddard Space Flight Center. *Proceedings of the IEEE*, *79*(4), 548–558. https://doi.org/10.1109/5.73546

Vanneschi, M. (1998). Variable grain architectures for MPP computation and structured parallel programming. En *Proceedings of*

the 1998 IEEE International Conference on Parallel Processing (pp. 154–161). IEEE. https://doi.org/10.1109/ICPP.1998.708501

Vlachos, A., & Peters, J. (2001). Curved PN triangles. En *Proceedings of the 2001 Symposium on Interactive 3D Graphics* (pp. 159–166). ACM. https://doi.org/10.1145/364338.364387