Ciclo Conferencia Grupo ArTeCS, UCM

Madrid, 22 Febrero 2018

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

R. Doallo

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- Metodología para Algoritmos Índice-Dígito:
- Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Arquitectura GPU

Programabilidad

Algoritmos

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Metodología para Algoritmos Índice-Dígito:

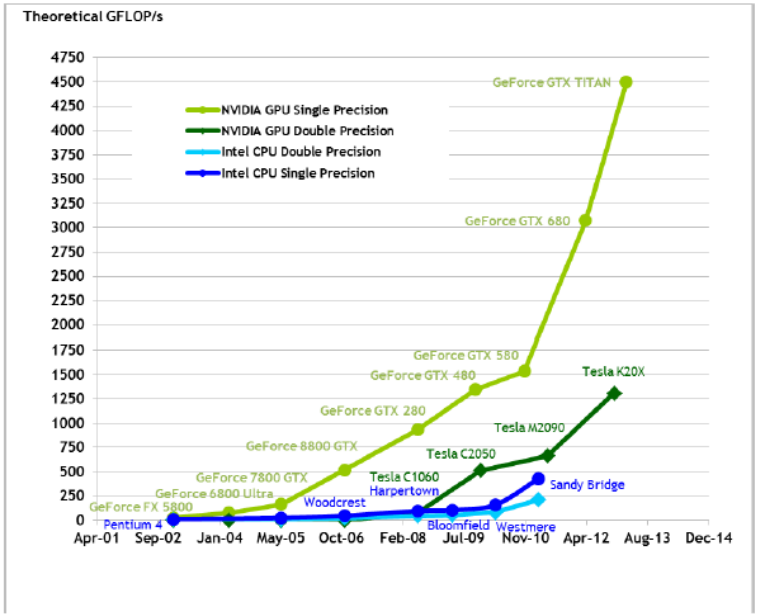
Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

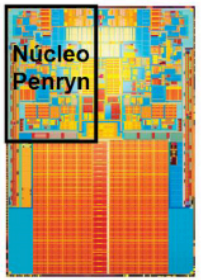
Comparativa CPU-GPU

- Las GPUs actuales están compuestas por cientos de núcleos de procesamiento en paralelo con un sistema de memoria distribuido bastante complejo pero de gran ancho de banda.
- Pueden procesar un alto número de tareas en paralelo.
- Rendimiento superior a un sistema con un procesador general multinúcleo.
- Solución de alto rendimiento no sólo para el procesamiento gráfico para las que fueron diseñadas. sino también para el procesamiento no gráfico (*GPGPU: General-Purpose Computation Using GPU*).

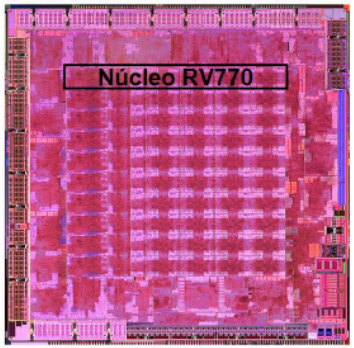
Comparativa CPU-GPU



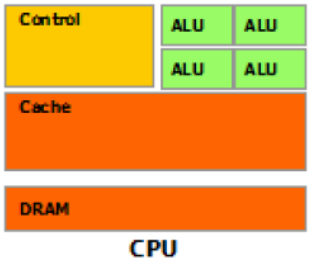
- Más transistores dedicados al procesamiento de datos más que a la caché o control de flujo.



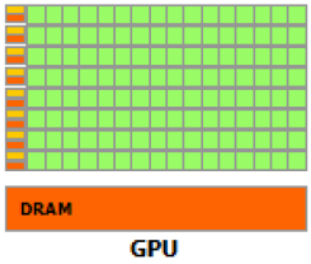
Procesador Intel Core 2 fabricado en 45 nm



Procesador AMD Radeon 4850 fabricado en 55 nm



CPU

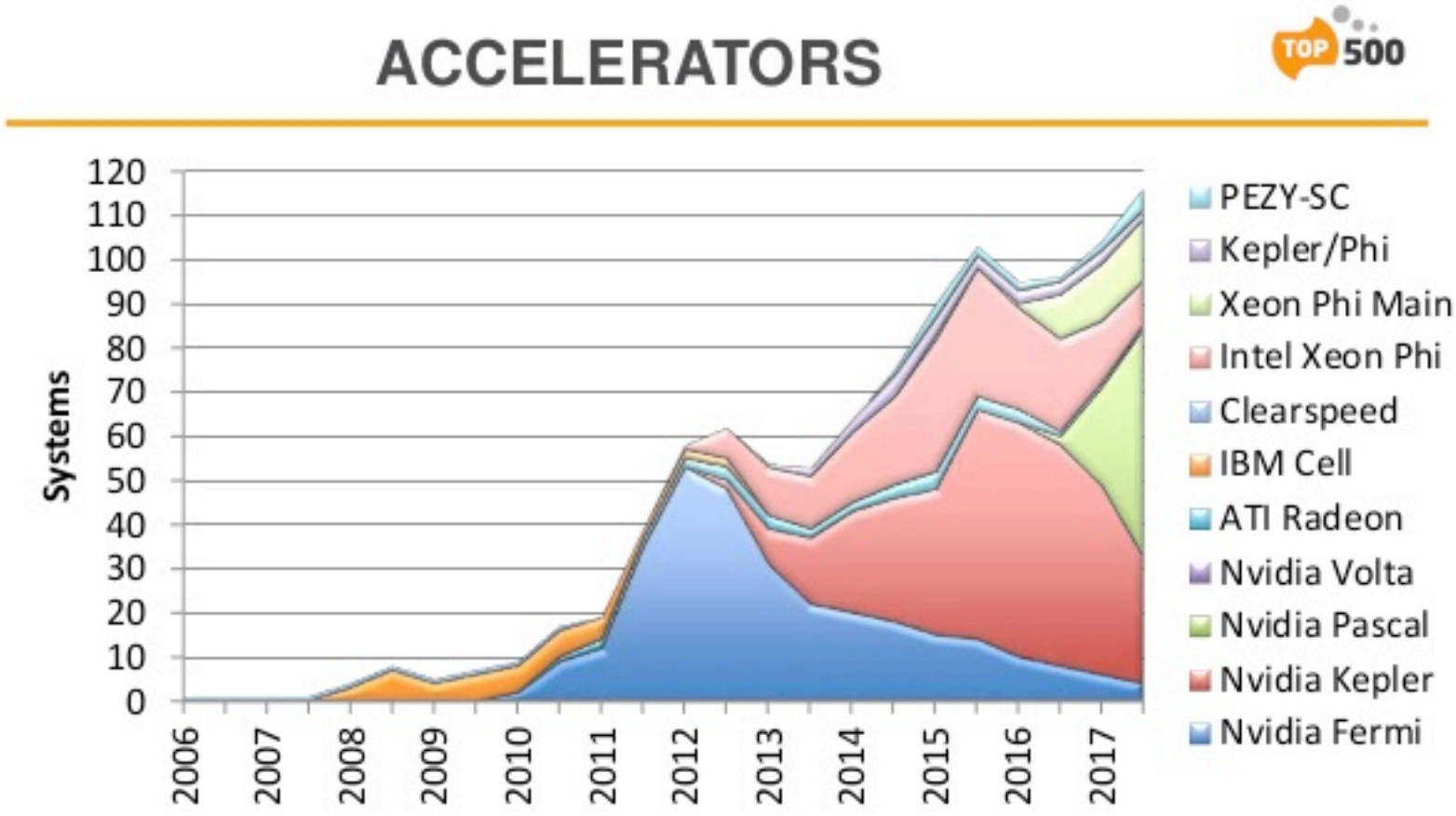


GPU

Presencia GPUs en TOP500

#	Site	Manufacturer	Computer	Country	Cores	Rmax (Pflop/s)	Power (MW)
1	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRCPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	15.4
2	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	17.8
3	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, Aries, NVIDIA Tesla P100	Switzerland	361,760	19.6	2.27
4	Japan Agency for Marine-Earth Science and Technology	ExaScaler	Gyoukou ZettaScaler-2.2 HPC System, Xeon 16C 1.3GHz, IB-EDR, PEZY-SC2 700Mhz	Japan	19,860,000	19.1	1.35
5	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	8.21
6	Lawrence Livermore National Laboratory	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	1,572,864	17.2	7.89
7	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	979,968	14.1	3.84
8	Lawrence Berkeley National Laboratory	Cray	Cori Cray XC40, Intel Xeons Phi 7250 68C 1.4 GHz, Aries	USA	622,336	14.0	3.94
9	JCAHPC Joint Center for Advanced HPC	Fujitsu	Oakforest-PACS PRIMERGY CX1640 M1, Intel Xeons Phi 7250 68C 1.4 GHz, OmniPath	Japan	556,104	13.6	2.72
10	RIKEN Advanced Institute for Computational Science	Fujitsu	K Computer SPARC64 VIIIfx 2.0GHz, Tofu Interconnect	Japan	795,024	10.5	12.7

Presencia GPUs en TOP500

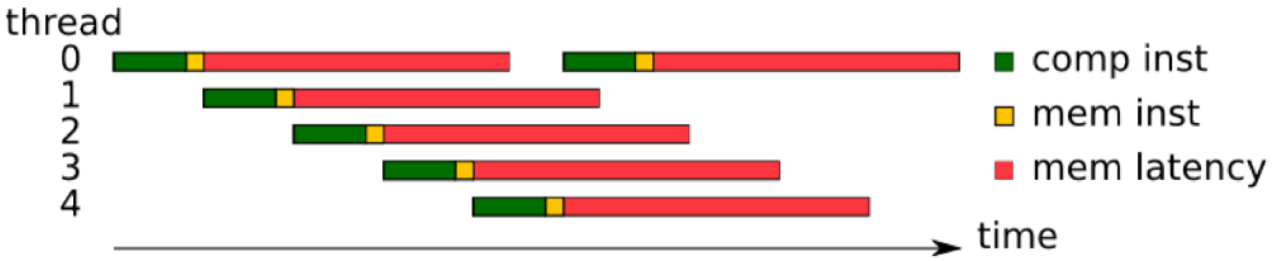


GPU: razones altas prestaciones?

- Interleaved Multi-threading
- Single Instruction Multiple Thread (SIMT)
- Núcleos sencillos

Interleaved Multithreading

Ejemplo:



Ventajas

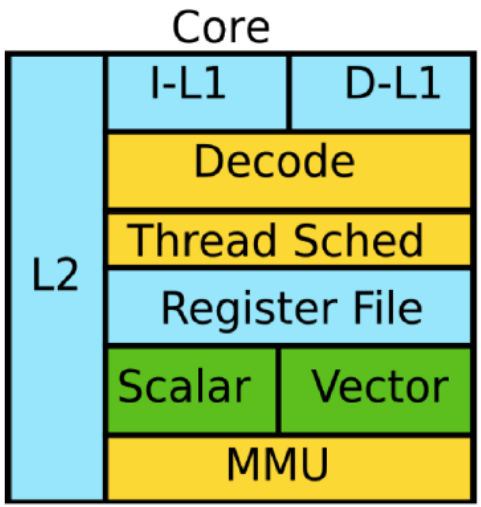
Se reduce el tamaño de la caché, no planificadores, no predicción de salto

Desventajas

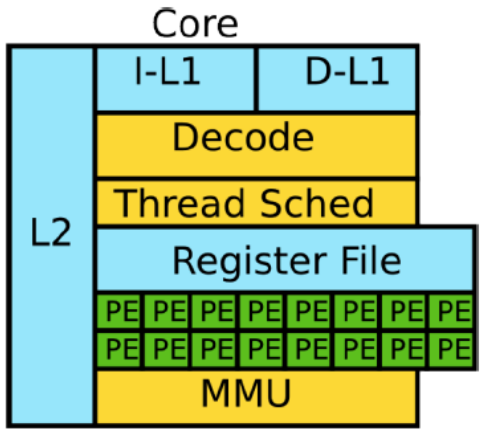
Presión de registro, thread scheduler, requiere alto paralelismo

Single Inst. Multiple Thread (SIMT)

CPU usa short SIMD: normalmente el vector es de tamaño 4

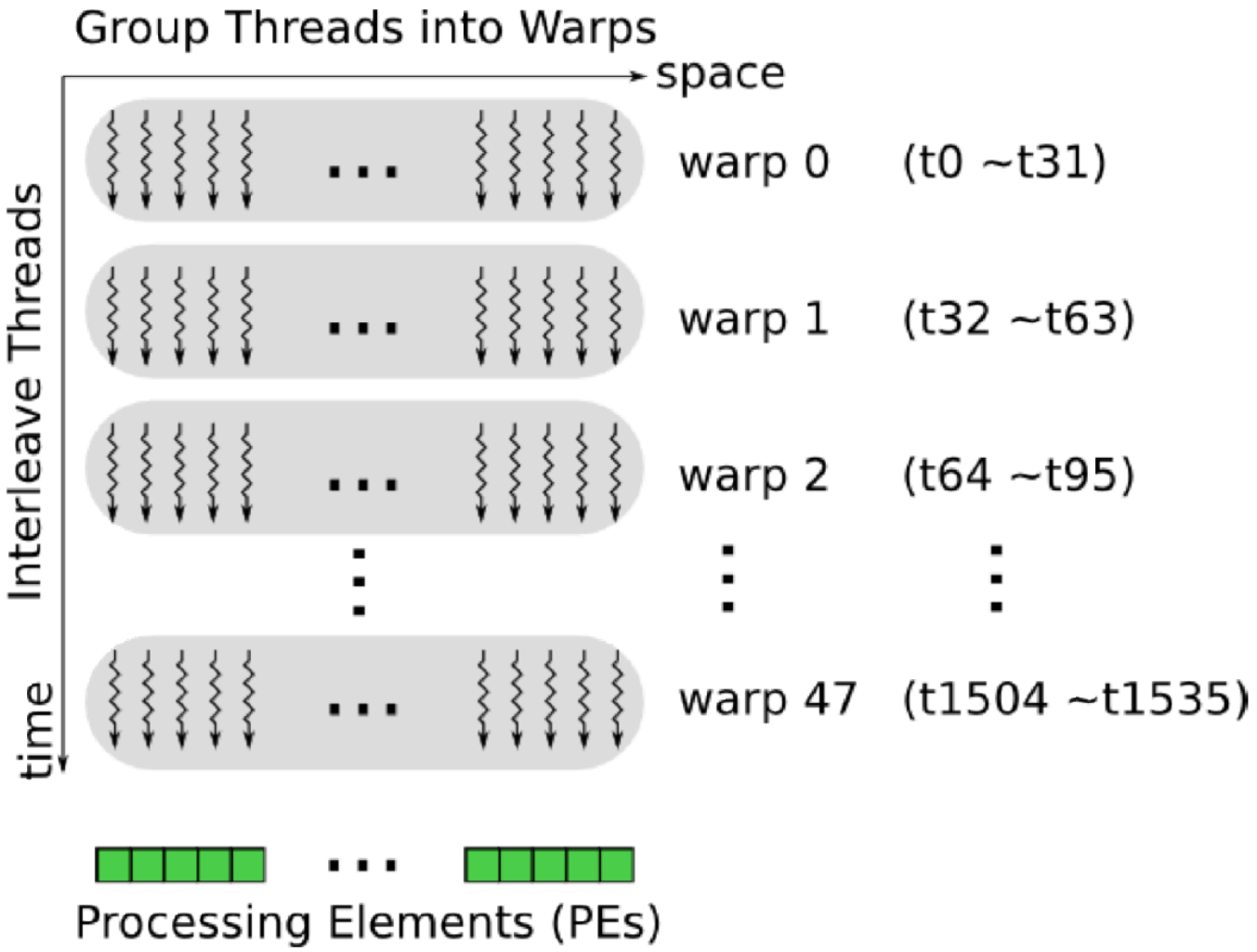


GPU usa wide SIMD: 8/16/32 ..



Single Inst. Multiple Thread (SIMT)

32 tareas agrupadas en un warp:



Núcleos sencillos (muchos)

Arquitectura (Nvidia)	Número de cores
Maxwell	1024 - 4098
Pascal	2560 - 3584
Volta	5120

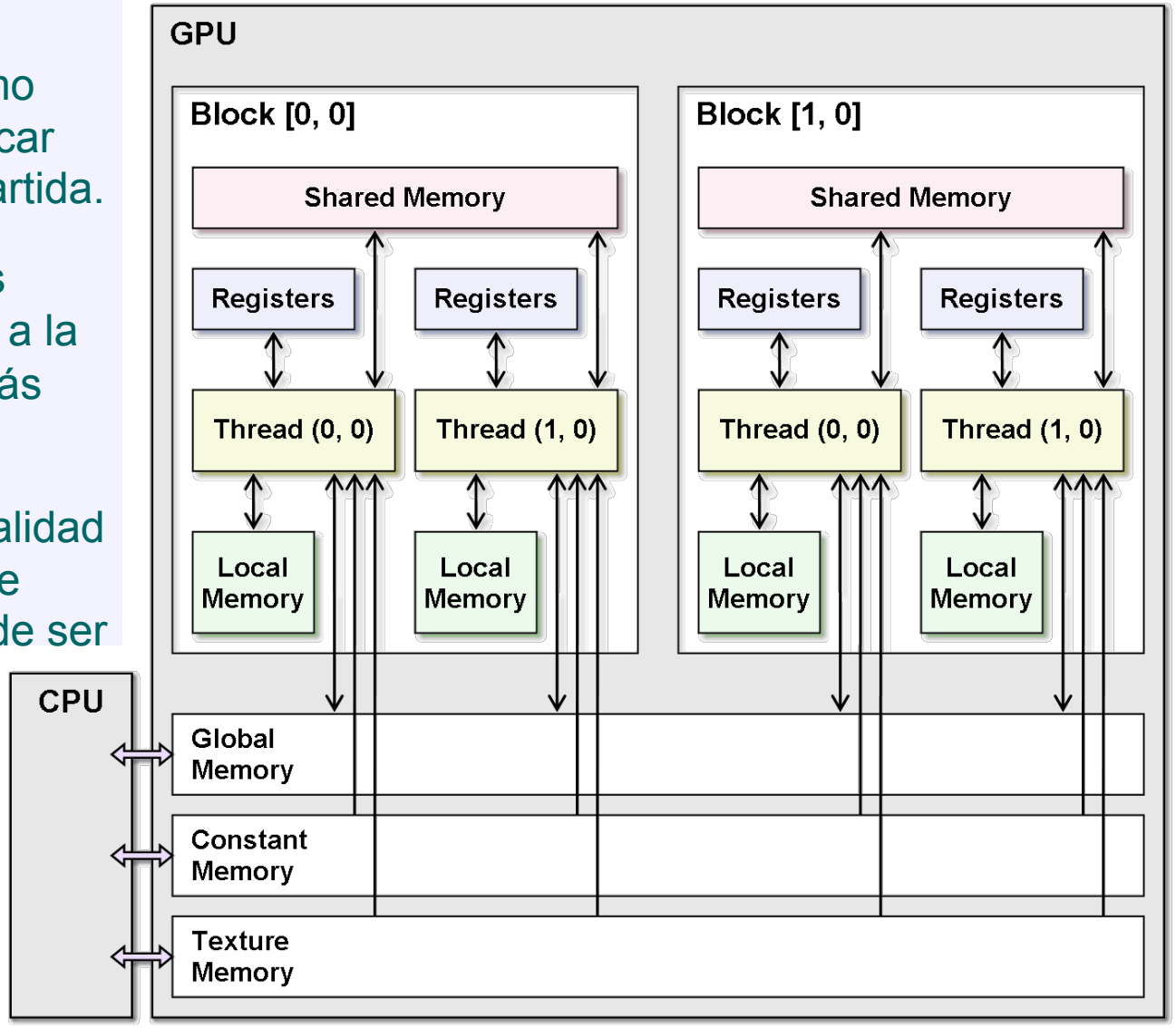
Organización básica GPU

- Jerarquía de memoria

- Streaming Multiprocessor (SM)

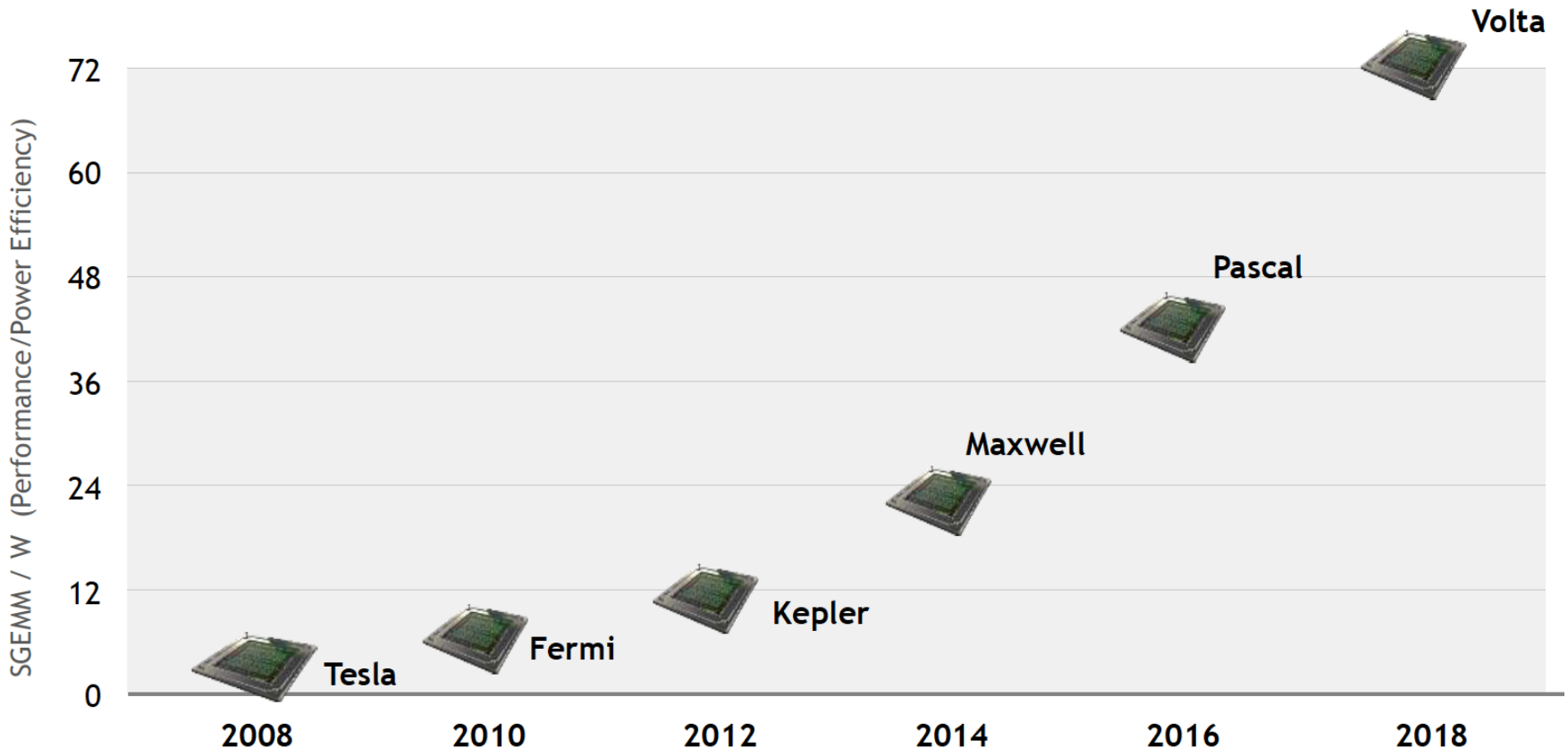
Organización básica GPU: Jerarquía de Memoria

- Los threads de un mismo bloque se pueden comunicar usando la memoria compartida.
- Los threads de distintos bloques necesitan recurrir a la memoria global, mucho más lenta.
- La memoria local en realidad es memoria global, aunque según la arquitectura puede ser cacheada.

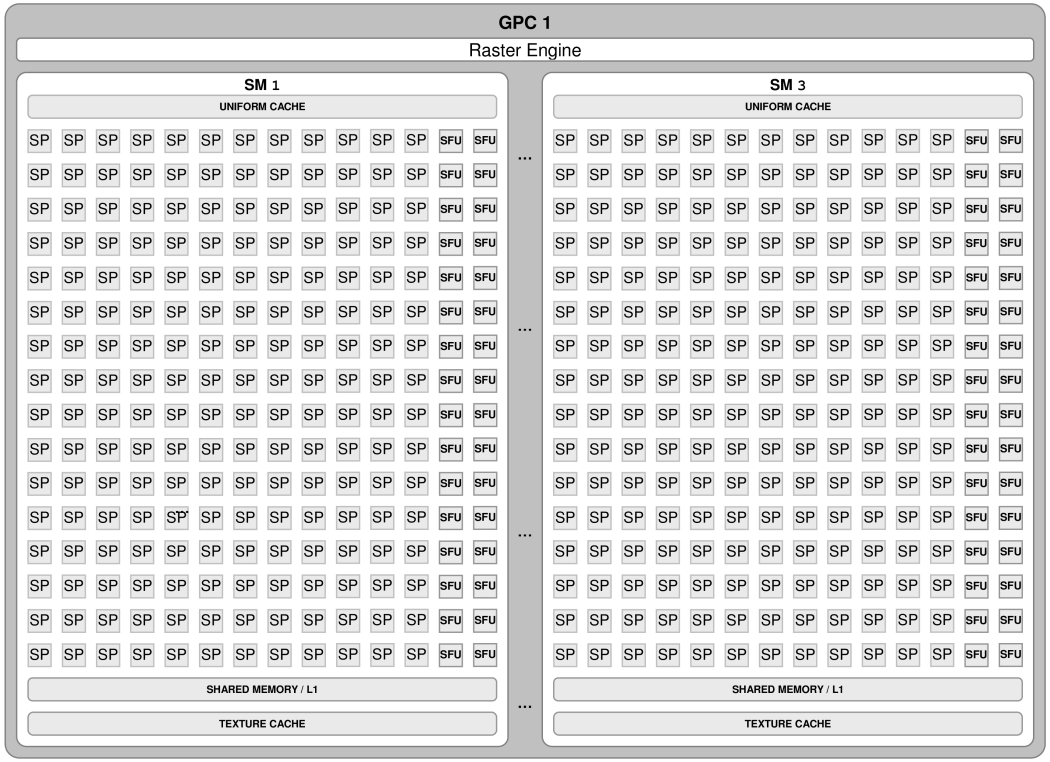


Organización básica GPU: SMs

Roadmap GPUs de Nvidia



Streaming Multiprocessor: Kepler



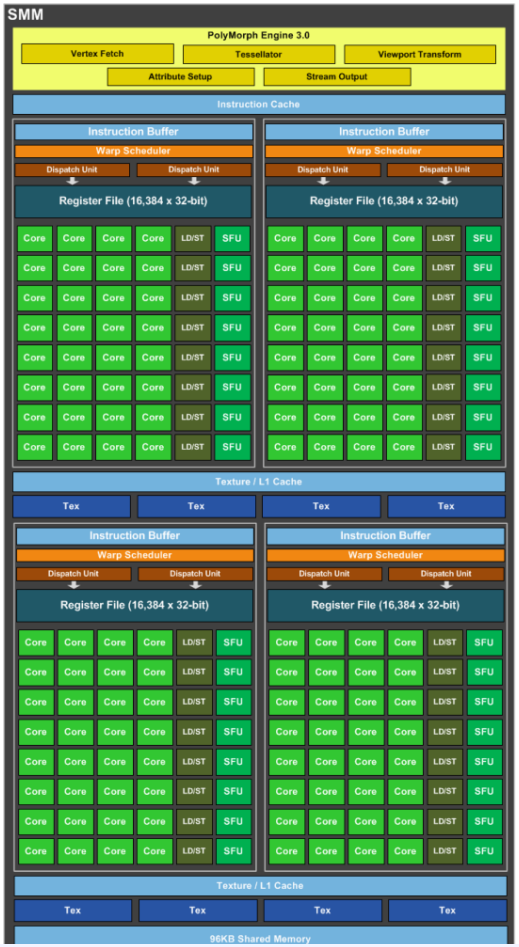
Arquitectura Kepler

Paso evolutivo: incrementa los recursos y mejora la eficiencia energética.

Añade algunas características como instrucciones shuffle o memoria unificada.

Las tarjetas con capacidades 3.5 también soportan paralelismo dinámico e Hyper-Q.

Maxwell Multiprocessor (SMM)



ArquitecturaMaxwell

Mayor eficiencia energética (performance/watt) que Kepler.

Incrementa la memoria compartida a 64 KB por SMM (aunque cada bloque seguirá usando como máximo 48 KB). Incrementa la L2.

Mejora las atomics en shared memory con soporte nativo en 32-bits.

Streaming Multiprocessor: Pascal

Arquitectura Pascal

Incrementa el número de cores de doble precisión.

Cambia memoria GDDR5 por HBM2 (incrementando muchísimo el ancho de banda). Vuelve a incrementar L2.

Introduce soporte para FP16 (deep learning).

Introduce NVLINK para high-speed interconexiones entre GPU-GPU y GPU-CPU.

Streaming Multiprocessor: Volta

Arquitectura Maxwell

Muy enfocadas para Deep Learning: introduce tensor cores.

Mejora ancho de banda HBM2, mejora ancho de banda NVLINK.

Hasta 2048 threads activos. Independent thread schedulling (cada thread tiene su propio contador de programa).

Introduce *cooperative groups*: grupos de threads a granularidades sub-block o multi-block.

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

- Arquitectura GPU
- **Programabilidad**
- Algoritmos

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- Metodología para Algoritmos Índice-Dígito:

Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

GPU Programmability

- *Directives*
  - Such as *hiCuda* or *OpenACC*
  - To have some GPU expertise

GPU Programmability

- *Directives*
  - Such as *hiCuda* or *OpenACC*
  - To have some GPU expertise

GPU Programmability

- *Directives*
  - Such as *hiCuda* or *OpenACC*
  - Have some GPU expertise
- *Automatic Compilers*
  - Such as *Bones* or *Par4all*
  - Systematic code translations lead to reduce performance

GPU Programmability

- *Directives*
  - Such as *hiCuda* or *OpenACC*
  - Have some GPU expertise

- *Automatic Compilers*
  - Such as *Bones* or *Par4all*
  - Systematic code translations lead to reduce performance

GPU Programmability

- *Directives*
  - Such as *hiCuda* or *OpenACC*
  - Have some GPU expertise
- *Automatic Compilers*
  - Such as *Bones* or *Par4all*
  - Systematic code translations lead to reduce performance
- *Accelerated Libraries*
  - Such as *SkePU* or *SkeCL*
  - HPC through simple routines without knowledge

GPU Programmability

- *Directives*
  - Such as *hiCuda* or *OpenACC*
  - Have some GPU expertise

- *Automatic Compilers*
  - Such as *Bones* or *Par4all*
  - Systematic code translations lead to reduce performance

- *Accelerated Libraries*
  - Such as *SkePU*, *SkeCL*, ***BPLG***
  - HPC through simple routines without knowledge

Diseño eficiente de algoritmos
Parallel Prefix sobre GPUs

Introducción

Arquitectura GPU

Programabilidad

Algoritmos

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

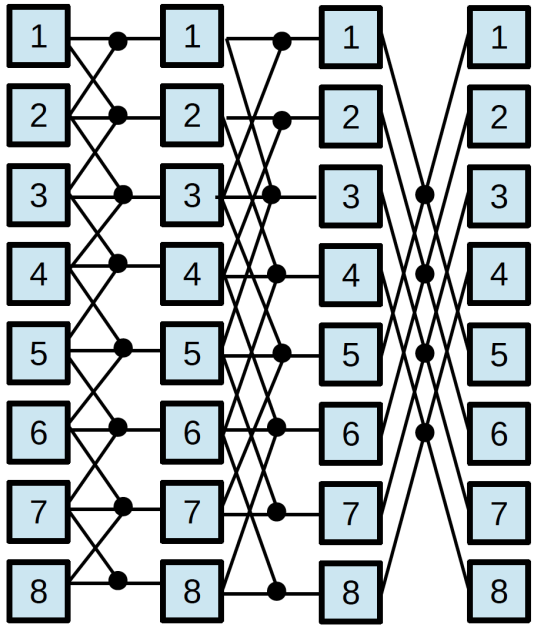
Metodología para Algoritmos Índice-Dígito:

Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

Parallel Prefix Algorithms

- Definition:
  - Solves a problem of size N in k steps, depicted by a directed acyclic oriented graph called prefix circuit
  - Computations performed by the Node operator (small circles below), and communication patterns regular and known at compile-time.



  - Regular algorithms which match well to GPU architectures
  - Access pattern does not depend on execution, it can be expressed as a linear function of each element index
  - Each resulting element is the combination of previous results of other elements.

- Algorithm examples:
  - FFT, DCT, Scan, Sorting, Tridiagonal System Solvers, ...

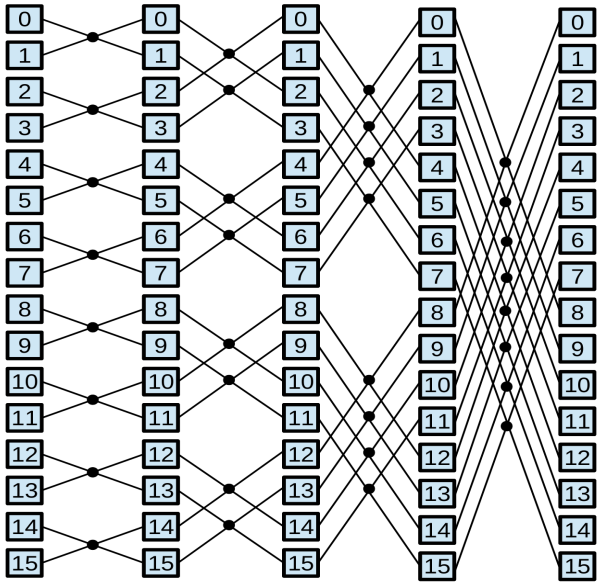
Index-Digit Algorithms

- Subset of Parallel Prefix Algorithms
  - Can be represented and modeled according to permutations of each element position's digits

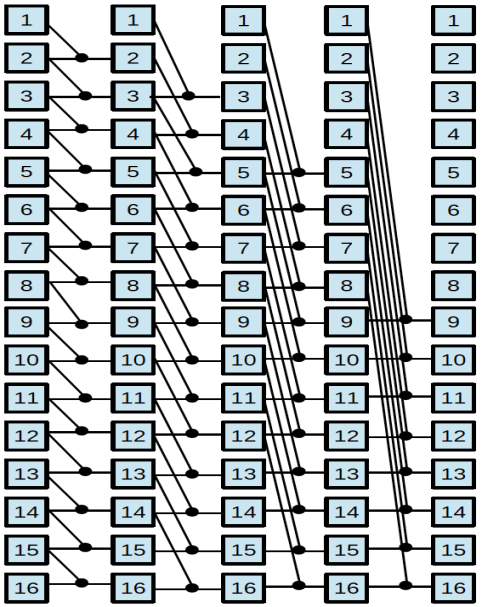
Data item x(t), being $t = t_n \cdot 2^{n-1} + \ldots + t_2 \cdot 2 + t_1$ is represented as $[t_n \ldots t_2 t_1]$

x(6) => [0110]

  - Constant number of Node operators throughout the steps
  - Elements that take part in one Node operator are not used by another Node operator in the same step



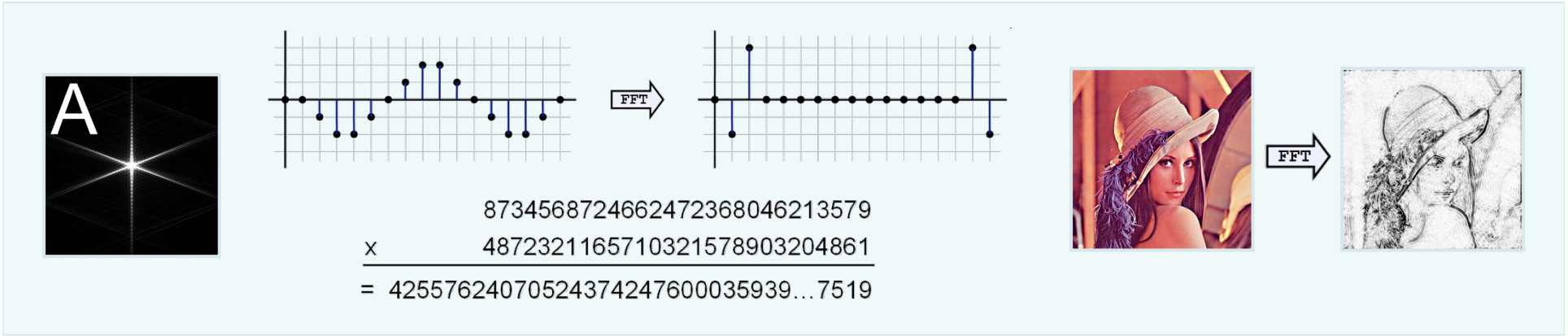
Index-Digit: FFT pattern



Parallel Prefix: scan pattern

Fast Fourier Transform (FFT)

- La transformada de Fourier es usada para procesar señales en muchas aplicaciones reales
  - Procesado de audio, reconocimiento de patrones, tratamiento de imágenes o manipulación de números largos…



- Existen múltiples implementaciones para CPU y GPU.
  - Para CPU: FFTW, Spiral, IPP de Intel…
  - Las más comunes en GPU son: ***CUFFT*** de NVIDIA, ***clMath*** de AMD.

Sistemas Tridiagonales (I)

Sistema de ecuaciones convencional

$$\left.\begin{aligned} 7x + y + 3z - 4w &= 6 \\ 4x - 9y - z + 2w &= 5 \\ x + 8y - 2z - 3w &= 0 \\ 8x - 5y + 4z - w &= 1 \end{aligned}\right\} \quad \begin{pmatrix} 7 & 1 & 3 & -4 \\ 4 & -9 & -1 & 2 \\ 1 & 8 & -2 & -3 \\ 8 & -5 & 4 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ Z \\ w \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ 0 \\ 1 \end{pmatrix}$$

Sistema de ecuaciones tridiagonal

$$\left.\begin{aligned} 7x + y + 3z - 4w &= 6 \\ 4x - 9y - z + 2w &= 5 \\ x + 8y - 2z - 3w &= 0 \\ 8x - 5y + 4z - w &= 1 \end{aligned}\right\} \quad \begin{pmatrix} 7 & 1 & 3 & -4 \\ 4 & -9 & -1 & 2 \\ 1 & 8 & -2 & -3 \\ 8 & -5 & 4 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ Z \\ w \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ 0 \\ 1 \end{pmatrix}$$

- Surgen de forma natural en aplicaciones científicas como la simulación de líquidos o en la propagación del calor.

Sistemas Tridiagonales (II)

Sistema de ecuaciones tridiagonal

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c3 & \\ & & \ddots & \ddots & \ddots \\ 0 & & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & & a_N & b_N \end{pmatrix}$$

Forma general de la ecuación E_i

$$E_i: \; a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

Uso y algoritmos de resolución

- Los S.T. aparecen en aplicaciones como la simulación de fluidos o la difusión del calor.
- Existen múltiples algoritmos de resolución tanto secuenciales como paralelos.
  - En *CUDA* son conocidos la librería ***CUSPARSE*** de *NVIDIA* y la ***CUDPP***.

Primitive Scan

Taking a binary associative operator $\oplus$, with identity *I*, and an array of *n* elements $[a_0, a_1, a_2, \ldots, a_{n-1}]$

returns: $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})\ ]$

Example with adding:

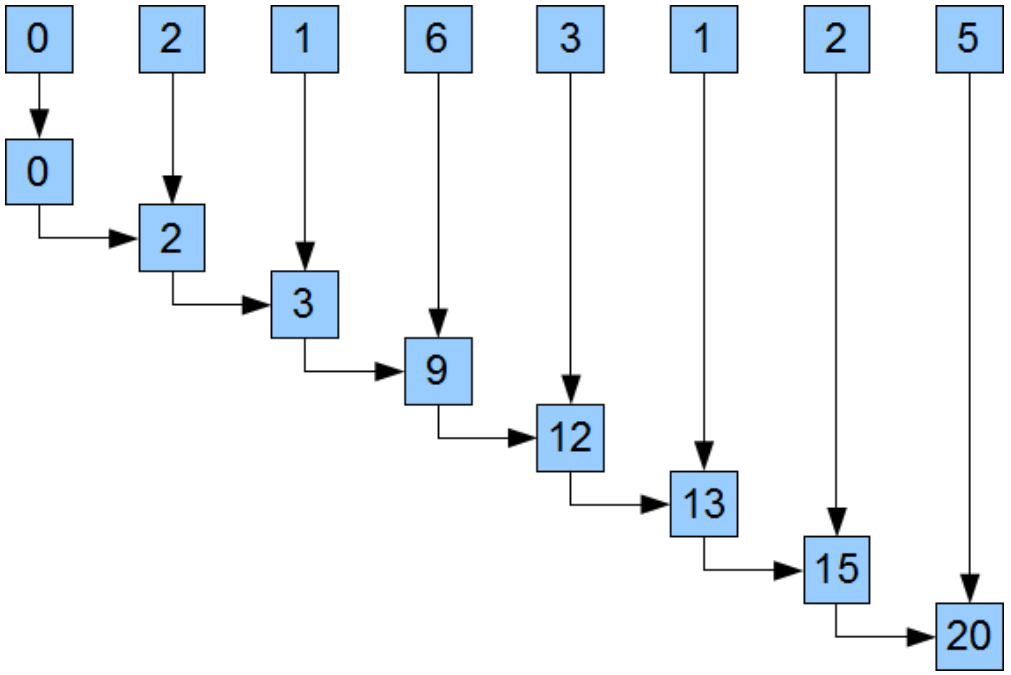
$$[3,1,7,1,4] \longrightarrow [0,3,4,11,12]$$

Example with multiplication:

$$[3,1,7,1,4] \longrightarrow [1,3,3,21,21]$$

Primitiva Scan

Cada elemento es el resultado de aplicar un operando binario (+) sobre él mismo y el elemento anterior.



- Ampliamente utilizado en bases de datos, procesado de imágenes, análisis léxico, evaluación de polinomios, etc..

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- Metodología para Algoritmos Índice-Dígito:
- Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

Estrategias de diseño para algoritmos Parallel-Prefix sobre (CUDA) GPUs

- **BPLG:** Butterfly Processing Library for GPU architectures
  - Jacobo Lobeiras, Margarita Amor, Ramon Doallo. BPLG: A Tuned Butterfly Processing Library for GPU Architectures. International Journal of Parallel Programming (2015)
- Clasification depending on **problem size**
  - **Case 1**: Problem data fit in shared memory.
    - Methodology for Index-Digit Algorithms
    - Each problem is assigned to one CUDA block
      - J.Lobeiras, M. Amor, R. Doallo. Designing Efficient Index-Digit Algorithms for CUDA GPU Architectures. IEEE Trans. Parallel Distrib. Syst. (2016)
    - New strategy for Parallel-Prefix algorithms
      - A. Pérez, M. Amor, R. Doallo. Parallel-Prefix Operations on GPU: Tridiagonal System Solvers and Scan Operators. J. Parallel and Distributed Computing (under review).
  - **Case 2**: Data larger than SHM but fit into GM
    - Methodology for Index-Digit Algorithms
    - Each problem is distributed among several blocks
    - Global synchronization needed
      - A. Pérez, M. Amor, J. Lobeiras, R. Doallo. Solving Large Problem Sizes of Index-Digit Algorithms on GPU: FFT and Tridiagonal System Solvers. IEEE Trans. Computers (2018)
  - **Case 3**: Data larger than single GPU GM
    - New strategy for Parallel-Prefix algorithms
    - Each problem is distributed among several GPUs
      - A. Pérez, M. Amor, R. Doallo, A. Nukada, S. Matsuoka. Efficient Solving of Scan Primitive on Multi-GPU Systems. IPDPS 2018 (in press).

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Metodología para Algoritmos Índice-Dígito:

Nueva estrategia (general) para Algoritmos Parallel-Prefix

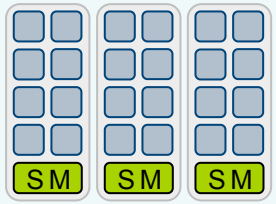
Trabajo Futuro

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

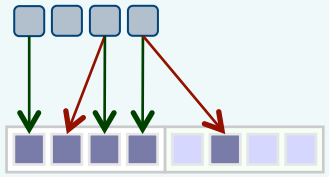
Butterfly Processing Library for GPU architectures (BPLG)

Diseño de librerías GPU: Dificultades

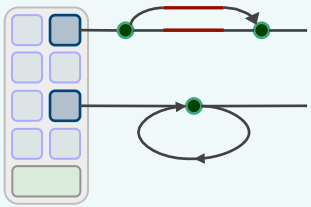
- La arquitectura especial de la GPU hace que la programación de librerías eficiente sea una tarea más complicada.



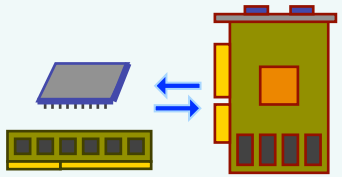
Limitaciones de memoria compartida y registros



Problemas de coalescencia



Bajo rendimiento en código secuencial o ejecución divergente



Ancho de banda y latencia del bus PCI-E

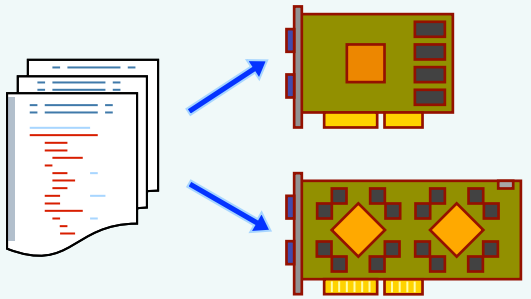
- Puede haber decisiones dependientes del hardware y la evolución rápida podría hacer que la mejor opción cambie entre generaciones.

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Diseño de librerías GPU: Parametrización

- Las *templates* de C++ son una herramienta de programación potente y flexible que permite generar múltiples versiones de un algoritmo.

```
template< int N, int D, int R, int S >
__global__ void SPLG_DCT(float* src, int size) {
   …
   __shared__ float* shm[S];
   …
   if(D < 0) scale<R, N/2>(reg);
   …
   radix<R, D>(reg, angle);
```



Único código para múltiples arquitecturas

- Las *templates* pueden usarse para ajustar la librería a la arquitectura, optimizando en tiempo de compilación códigos parametrizados.

BPLG : Butterfly Processing Library for GPUs

Implementation on CUDA

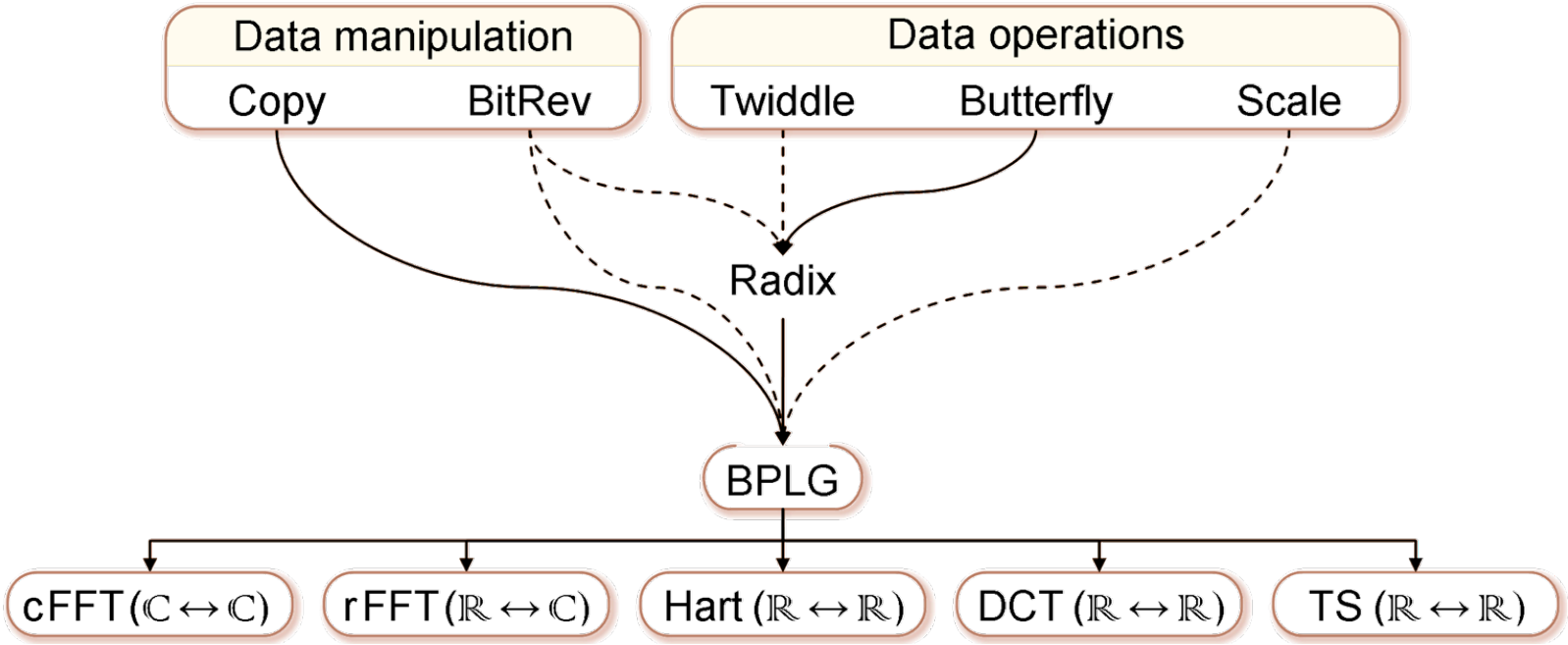
- Composed by high-level optimized *building blocks* (skeletons)
- Each block performs a small part of the whole work
- The body of the kernel remains almost costant along algorithms, just changing the Node Operator's building block implementation

BPLG : Butterfly Processing Library for GPUs

- BPLG Features
  - The use of templates, enabling Generic Programming and Template Metaprogramming.
  - Knowing additional info. at compile-time (number of threads, shared memory, ...) enables many optimizations like loop unrolling or data reordering using register renaming.
  - All building blocks were designed to operate in any GPU memory space.
  - Avoids non-uniform access
  - Hybrid communication strategy inside the block
  - Uses customized CUDA data types as *float4*, *int4*, ...

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Bloques constructivos básicos



Creación de algoritmos basada en bloques

Los algoritmos son construidos en base a una serie de bloques constructivos básicos.

Cada bloque está formado por una o más plantillas y realiza solo una parte del trabajo.

Las funciones se clasifican en dos categorías: operaciones y manipulación de datos.

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Diseño de los bloques constructivos

Ejemplos de templates usadas en los algoritmos

Copy: Transfiere 'N' elementos desde un buffer 'X' con un stride opcional 'sd' a otro buffer 'Y' con un stride opcional 'ss'.

BitReverse: Reorganiza los elementos de un buffer 'X' de tamaño 'N' con un stride opcional 'ss', de tal forma que el resultado está en orden de bit inverso.

Butterfly: Realiza la etapa de computación principal de cada paso radix permitiendo un stride opcional. Las operaciones pueden variar con el algoritmo.

```
1: template<int N> inline __device...
2: Copy(TData* Y, int sd,
3:       const TData* X, int ss = 1) {
4:      #pragma unroll
5:      for(int i = 0; i < N; i++)
6:            Y[i * sd] = X[i
7: }
```

(a) Copy

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Diseño de los bloques constructivos

Ejemplos de templates usadas en los algoritmos

Copy: Transfiere 'N' elementos desde un buffer 'X' con un stride opcional 'sd' a otro buffer 'Y' con un stride opcional 'ss'.

BitReverse: Reorganiza los elementos de un buffer 'X' de tamaño 'N' con un stride opcional 'ss', de tal forma que el resultado está en orden de bit inverso.

Butterfly: Realiza la etapa de computación principal de cada paso radix permitiendo un stride opcional. Las operaciones pueden variar con el algoritmo.

```
1:  template<int N> inline __device...
2:  Copy(TData* Y, int sd,
3:         const TData* X, int ss = 1) {
4:        #pragma unroll
5:        for(int i = 0; i < N; i++)
6:              Y[i * sd] = X[i
7:  }
```

(a) Copy

```
1:  template<int N> inline __device...
2:  BitReverse(TData* X, int ss = 1);
3:
4:  template<> inline __device__ void
5:  BitReverse< 4>(TData* X, int ss) {
6:        Swap(X[ 1 * ss], X[ 2 * ss]);
7:  }
8:
9:  template<> inline __device__ void
10: BitReverse< 8>(TData* X, int ss) {
11:       Swap(X[ 1 * ss], X[ 4 * ss]);
12:       Swap(X[ 3 * ss], X[
13: }
```

(b) BitReverse

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Diseño de los bloques constructivos

Ejemplos de templates usadas en los algoritmos

Copy: Transfiere 'N' elementos desde un buffer 'X' con un stride opcional 'sd' a otro buffer 'Y' con un stride opcional 'ss'.

BitReverse: Reorganiza los elementos de un buffer 'X' de tamaño 'N' con un stride opcional 'ss', de tal forma que el resultado está en orden de bit inverso.

Butterfly: Realiza la etapa de computación principal de cada paso radix permitiendo un stride opcional. Las operaciones pueden variar con el algoritmo.

```
1:  template<int N> inline __device...
2:  Copy(TData* Y, int sd,
3:         const TData* X, int ss = 1) {
4:        #pragma unroll
5:        for(int i = 0; i < N; i++)
6:              Y[i * sd] = X[i
7:  }
```

(a) Copy

```
1:  template<int N> inline __device...
2:  BitReverse(TData* X, int ss = 1);
3:
4:  template<> inline __device__ void
5:  BitReverse< 4>(TData* X, int ss) {
6:        Swap(X[ 1 * ss], X[ 2 * ss]);
7:  }
8:
9:  template<> inline __device__ void
10: BitReverse< 8>(TData* X, int ss) {
11:       Swap(X[ 1 * ss], X[ 4 * ss]);
12:       Swap(X[ 3 * ss], X[
13: }
```

(b) BitReverse

```
1:  template<int N, int D> inline __dev...
2:  Butterfly(TData* X, int ss = 1) { }
3:
4:  template<> inline __device__ void
5:  Butterfly<2, 1>(TData* X, int ss) {
6:        TData t0 = X[0 * ss];
7:        X[ 0 * ss] = t0 + X[1 * ss];
8:        X[ 1 * ss] = t0 - X[1 * ss];
9:  }
10:
11: template<> inline __device__ void
12: Butterfly<4, 1>(TData* X, int ss) {
14:       Butterfly<2, 1>(X + 0 * ss, 2 * ss);
15:       Butterfly<2, 1>(X + 1 * ss, 2 * ss);
16:
17:       TData t3 = X[3 * ss];
18:       X[3*ss] = make_TData(t3.y, -t3.x);
19:
20:       Butterfly<2, 1>(X + 0 * ss, 1 * ss);
21:       Butterfly<2, 1>(X +
22: }
```

(e) Butterfly

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Ejemplo de algoritmo: ***DCT (I)***

Cada bloque resuelve problemas de tamaño '*N*', trabajando sobre '*S*' elementos. Si '*S / N > 1*' varios problemas son procesados en batch.

El tamaño del radix es seleccionado por '*R*' y la dirección de la transformada por '*D*'.

Algunos algoritmos como la DCT requieren una pequeña etapa auxiliar. En el resto, los datos se cargan de memoria global a registros.

El primer paso es siempre un operador opcional de escalado, seguido de una etapa mixed-radix.

A continuación, un bucle reordena los datos y computa las etapas radix-R restantes.

```
1:  template<int N, int D, int R, int S> __global__ void
2:  BPLG_DCT(float* src, int size) {
3:      // Define some thread/group identifiers and their memory offsets
4:      // Register/ShMem static allocation based on template parameters
5:
6:      // DCT pre-processing, loads data from GlbMem into registers
7:      if(D >= 0) {
8:          copy<R>(shm + threadXY, S / R,
9:                  src + dctPos, S / R);
10:         __syncthreads();
11:         packDCT<N, R, D, S>(reg, shm, thrId, bchId);
12:     } else ...
13:
14:     // The first mixed-radix stage is always computed
15:     if(D < 0) scale<R, N/2>(reg); // Inverse transform scaling
16:     radix<R, MixR, D>(reg, R / MixR);
17:
18:     // This loop computes the remaining radix stages
19:     for(int accRad = MixR; accRad < N; accRad *= R) {
20:         // Reordering stage in shared memory, Reg->Shm->Reg
21:         __syncthreads();
22:         copy<R>(shm + shmPos, N / R, reg);
23:         int stride = ... , readPos = ... ;
24:         __syncthreads();
25:         copy<R>(reg, shm + readPos, stride);
26:         // Computation with data in registers
```

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Ejemplo de algoritmo: ***DCT (II)***

Cada iteración está compuesta por:

1. Etapa de reordenamiento

- Se realiza en memoria compartida con dos operadores *copy* variando los offsets y strides.

- Se requieren sincronizaciones para mantener el orden de ejecución.

2. Etapa de computación

- Primero se obtiene el *twiddle* en función de la etapa y del identificador de thread.

- A continuación se llama el operador radix-R sobre los datos en registros.

Dependiendo del algoritmo deseado, la etapa de postprocesado puede ser necesaria.

Finalmente, los resultados se copian a memoria global.

```
13:
14:     // The first mixed-radix stage is always computed
15:     if(D < 0) scale<R, N/2>(reg); // Inverse transform scaling
16:     radix<R, MixR, D>(reg, R / MixR);
17:
18:     // This loop computes the remaining radix stages
19:     for(int accRad = MixR; accRad < N; accRad *= R) {
20:         // Reordering stage in shared memory, Reg->Shm->Reg
21:         __syncthreads();
22:         copy<R>(shm + shmPos, N / R, reg);
23:         int stride = ... , readPos = ... ;
24:         __syncthreads();
25:         copy<R>(reg, shm + readPos, stride);
26:         // Computation with data in registers
27:         float ang = getAngle<D, R>(accRad, thrId>>cont);
28:         radix<R, D>(reg, ang);
29:     }
30:
31:     // DCT post-processing, stores the result from registers to ShMem
32:     __syncthreads();
33:     if(D >= 0) {
34:         radixDCT<N, R, D, S>(reg, thrId);
35:         copy<R>(shm + shmPos, N / R, reg);
36:     } else ...
37:
38:     // Stores the final result from shred memory to global memory
39:     __syncthreads();
40:     copy<R>(src + dctPos, S / R,
41:             shm + threadXY, S / R);
42: }
```

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Tabla de parámetros (para una FFT compleja en Kepler)

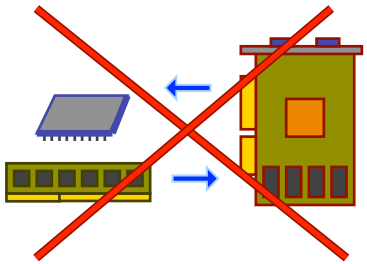
N	Tamaño de problema	**L_1**	Threads por problema	**S_B**	Mem. Shared / bloque
R	Tamaño de radix	**L_2**	Threads en batch	**B_{SM}**	Bloques por SM
R_t	Registros por thread	**L_{1x2}**	Threads por bloque	**T_{SM}**	Threads por SM

N	R	R_t	L_1	L_2	L_{1x2}	S_B	B_{SM}	T_{SM}	Ocup.
4	2	14	2	64	*128*	2048	16	2048	*1.00*
8	2	18	4	32	*128*	2048	16	2048	*1.00*
16	2	18	8	16	*128*	2048	16	2048	*1.00*
32	2	18	16	8	*128*	2048	16	2048	*1.00*
64	4	28	16	8	*128*	4096	12	1536	*0.75*
128	4	28	32	4	*128*	4096	12	1536	*0.75*
256	4	27	64	2	*128*	4096	12	1536	*0.75*
512	4	28	128	1	*128*	4096	12	1536	*0.75*
1024	4	27	256	1	*256*	8192	6	1536	*0.75*
2048	8	37	256	1	*256*	16384	3	768	*0.38*
4096	8	36	512	1	*512*	32768	1	512	*0.25*

BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

***Análisis de Rendimiento**: metodología de pruebas*

- Los datos residen en la memoria de la GPU, no se realizan transferencias durante las pruebas.



- El rendimiento de **señales** se mide en GFLOPS usando las expresiones:

  `Datos complejos` ($\mathbb{C}$):

  "$5N \log_2(N) \cdot b \cdot 10^{-9} / t$"

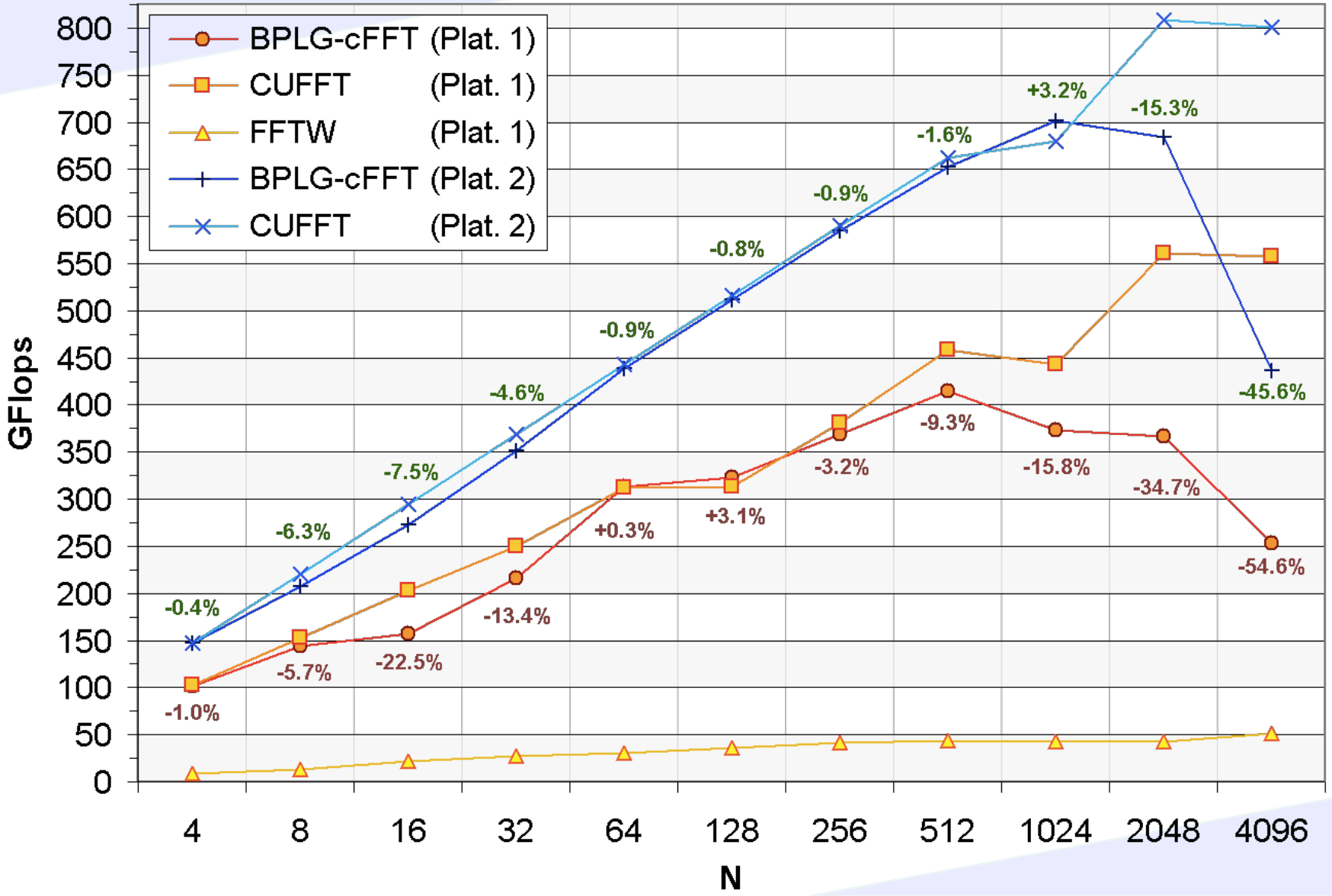
  `Datos reales` ($\mathbb{R}$):

  "$2.5N \log_2(N) \cdot b \cdot 10^{-9} / t$"

- El rendimiento de **sistemas tridiagonales** se mide en MRows/s.
- El rendimiento de la **primitiva scan** se mide en MData/s.

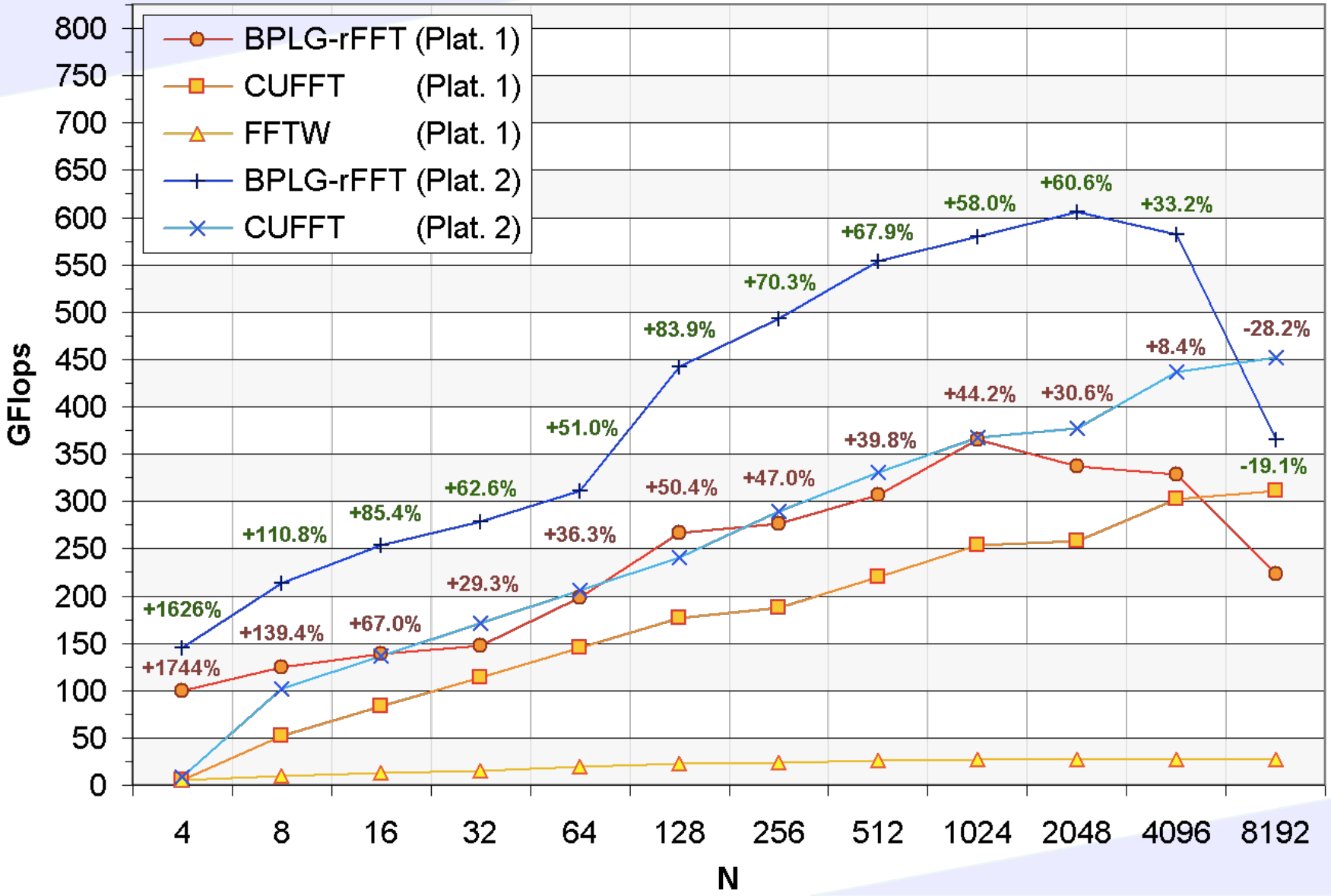
BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Análisis de rendimiento: BPLG-cFFT



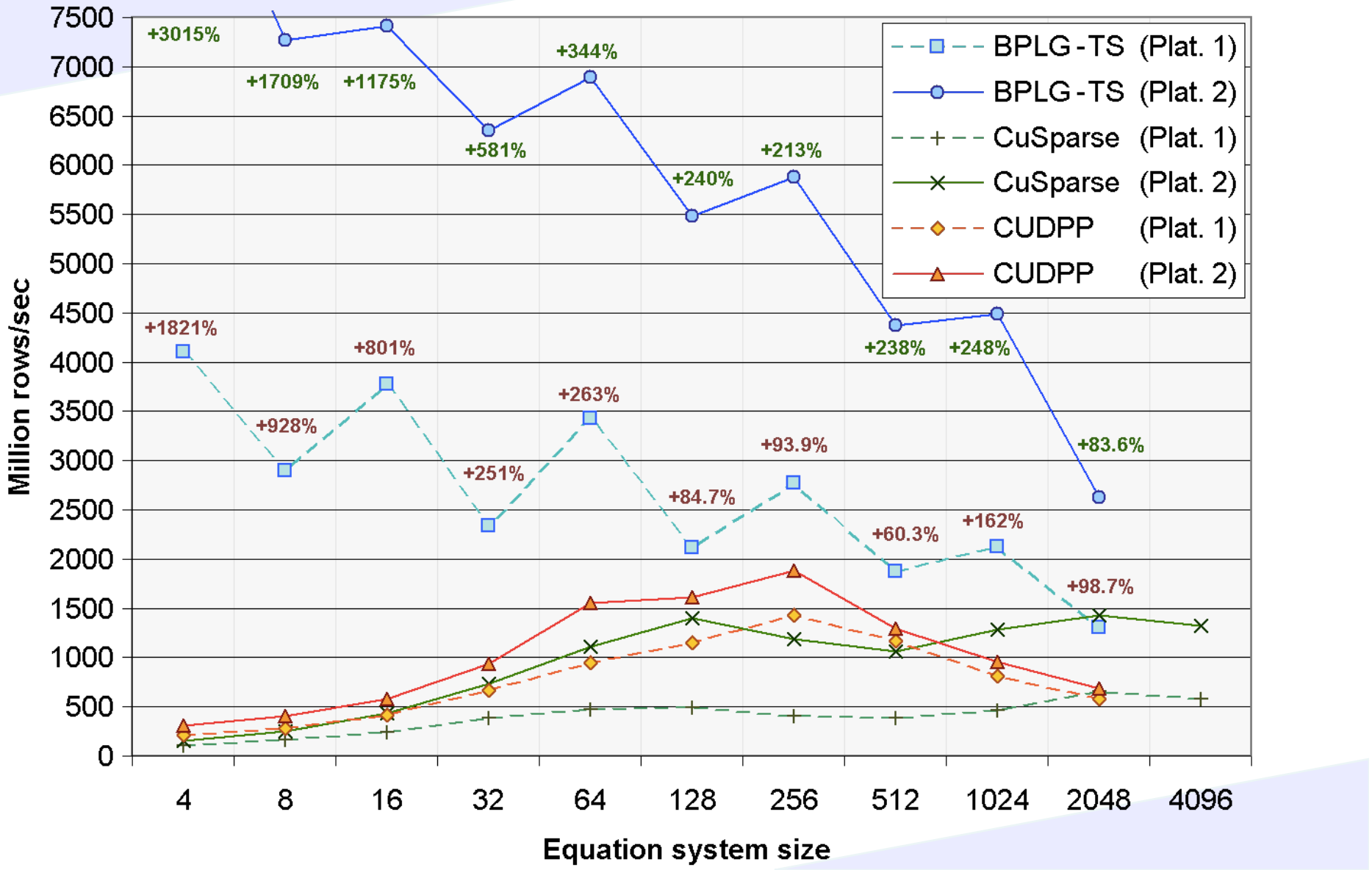
BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Análisis de rendimiento: BPLG-rFFT



BPLG: Una Librería Configurable para Algoritmos tipo Mariposa

Análisis de rendimiento: BPLG-TS



Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- **Metodología para Algoritmos Índice-Dígito**
  - **Caso 1: SM>=N**
  - Caso 2: SM<=N<=GM
- Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

Metodología para Algoritmos Índice-Dígito

*Metodología en **2 fases***

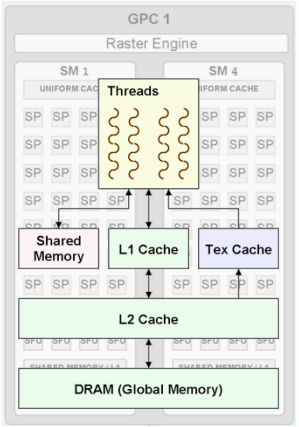
Fase 1 - Análisis de recursos GPU: obtener *resource factors*

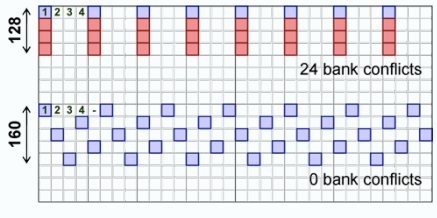
Recursos compartidos

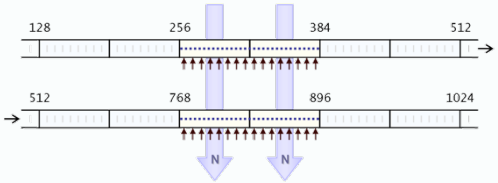
Nivel de paralelismo

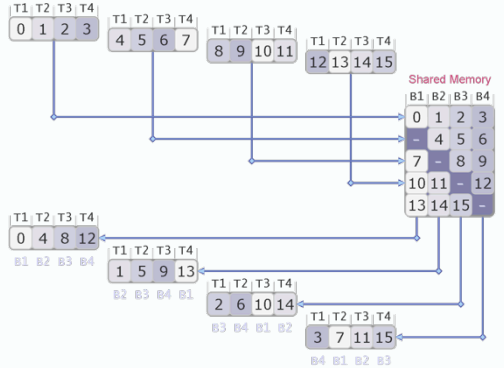
Coalescencia

Conflictos de banco









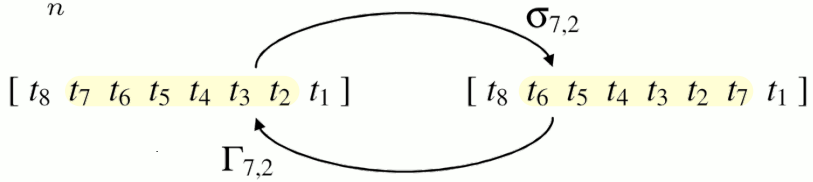
Fase 2 - *Operator string manipulation*

Tuning mapping vector

Cadenas de operadores

$$[\ \cdots t_{11}\ t_{10}\ \overbrace{t_9\ t_8\ t_7\ t_6\ t_5}^{l}\ \overbrace{t_4\ t_3\ t_2\ t_1}^{p}\ ]$$

b s n



$$\prod_{i=1}^{\lfloor n/r \rfloor} \Gamma^r_{n,n-i\cdot r+1}\ \rho_{n,n-r+1}\ B^r_{n-r+1}$$

Metodología para Algoritmos Índice-Dígito

Parametrización para asignación de datos a recursos GPU

Parameter	*Definition*
$N = r^n$	Problem size.
r^{batch}	Number of problems being simultaneously solved.
$P = r^p$	Number of elements stored in registers per thread.
$B = r^b$	Number of thread blocks per stage, where $B = B_x \cdot B_y$
$L = r^l$	Number of threads per thread block, where $L = L_x \cdot L_y$
$S = r^s$	Number of shared-memory elements per thread block, where $s = n - b_x$ and $s = p + l$

Metodología para Algoritmos Índice-Dígito

- *GPU resources utilization analysis* phase

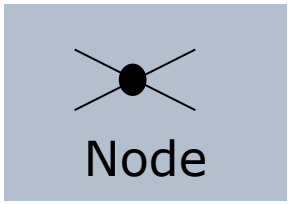
Problem size $N = r^n$ where *r* is the radix and *n* the number of problem steps

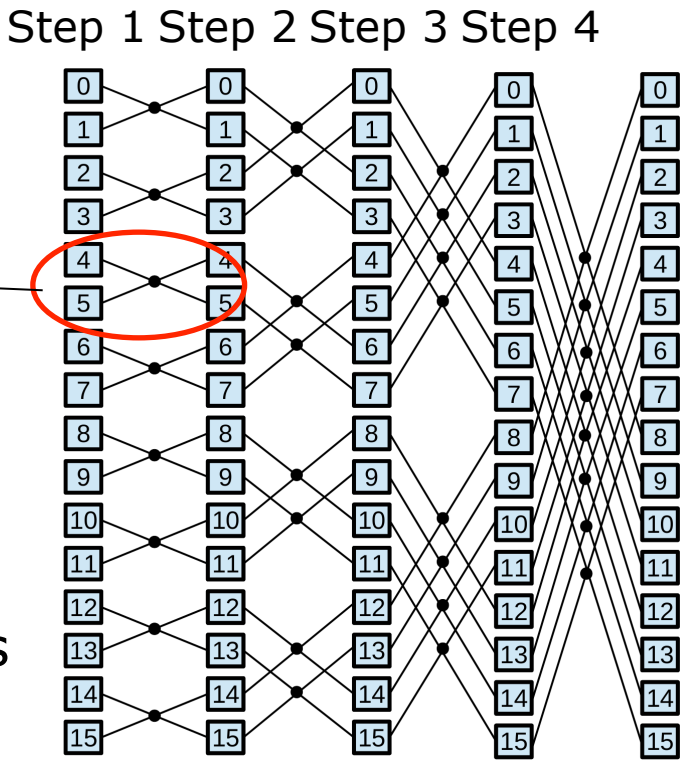
N = 16 =2^4

16 elements

radix – 2

steps - 4





Acyclic graph -> prefix circuit

Data dependences among steps

Metodología para Algoritmos Índice-Dígito

- *GPU resources utilization analysis* phase

Index-Digit representation:

Data item x(t) being t,

$$t = t_n \cdot 2^{n-1} + \ldots + t_2 \cdot 2 + t_1$$

is represented as: $\left[t_n \ldots t_2 t_1 \right]$

For instance: *x(6)* is written as [0110]

Metodología para Algoritmos Índice-Dígito

- *GPU resources utilization analysis* phase

Kernel data are divided among:

- $B = r^b$ blocks, each one executes $L = r^l$ threads.
- A thread performs $P = r^p$ data stored in private registers.
- Threads within a block have access to $S = r^s$ data in shared memory
- Mapping of data can be represented with a 5-tuple of the form *(n,p,s,l,b)* in our methodology
- Furthermore, $G = r^{batches}$ problems are simultaneously executed in a batch mode.

Metodología para Algoritmos Índice-Dígito

- *GPU resources utilization analysis* phase

  - CUDA programming -> $b = (b_x, b_y)$
  - Also with threads $l = (l_x, l_y)$
  - In our methodology, all threads in a block within the same *threadIdx.y* work in the same problem
  - Equations:

$$s = p + l \qquad l_y = s - n \qquad b = batches - l_y$$

Metodología para Algoritmos Índice-Dígito

- *GPU resources utilization analysis* phase

Example: { p=2, l=(6,1), b=(8,0) }

- 4 elements per thread stored in registers *p=2*
- 128 threads per block. 64 threads in *dimX* and 2 threads in *dimY*
- *s=p+l* so 512 elements in shared memory
- $n = s - l_y$ therefore 256 elements per problem
- 256 blocks
- $b = batches - l_y$ thus 512 problems simultaneously.

Metodología para Algoritmos Índice-Dígito

- *GPU resources utilization analysis* phase

We define the following premises:

Premise 1: *Balance warp and block parallelism*.

- Warp parallelism: As many actives warps per SM as possible.
- Block parallelism: As many actives blocks per SM as possible. Even good performance at low warp occupancy (high ILP). Better usage of shared memory. Less synchronization penalty.

Premise 2: *Increase computational workload per thread.*

- Increasing radix r implies reducing steps, that means, synchronization barriers and loop iterations
- Be careful with registers, high register usage implies less warp parallelism and/or register spilling.

Metodología para Algoritmos Índice-Dígito

Mapping Vector (I)

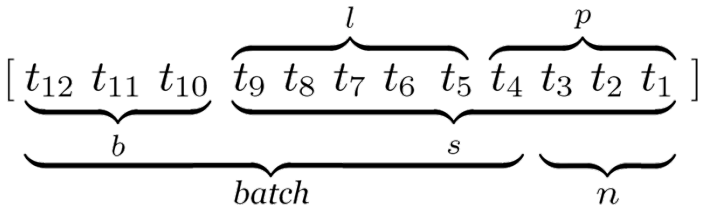
Mapping Vector Parameters

- Cada kernel procesará simultáneamente r^{batch} señales con $N = r^n$ datos en una sola invocación, donde r depende del radix elegido.
- La ejecución se expresa con una 5-tupla:

$$(n, p, s, l, b)$$

- Si se considera $s = p + l$ para simplificar los intercambios y $b = batch - (s - n)$
  => 3 parámetros independientes: (n, p, s)

Ejemplo:



n:	signal size
p:	private space
s:	shared space
l:	local space
b:	block space

Se representaría como la 5-tupla: (3, 4, 9, 5, 3)

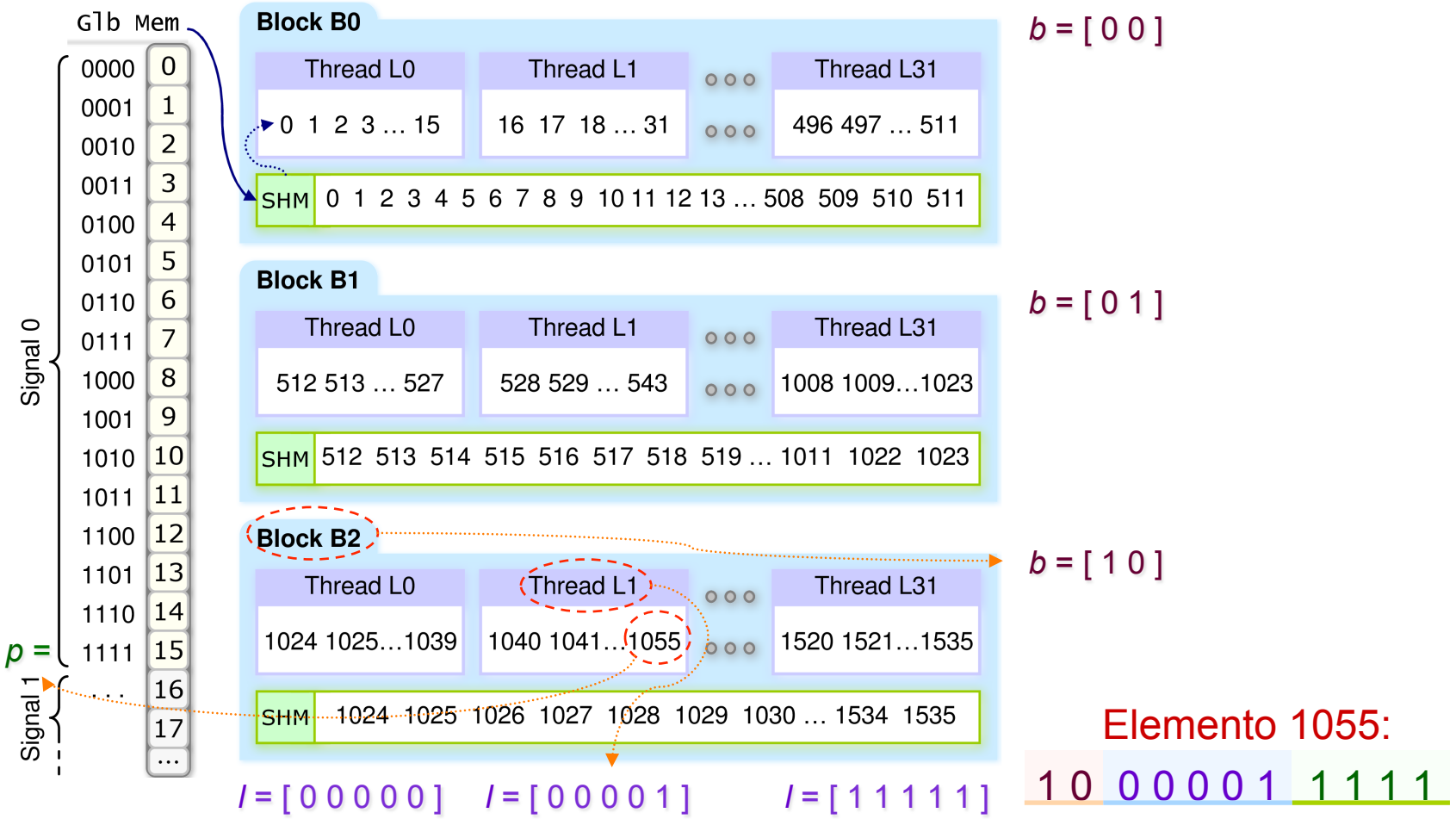
La notación se puede simplificar a: (3, 4, 9)

Metodología para Algoritmos Índice-Dígito

Mapping Vector (II)

$$[\ \underbrace{t_{11}\ t_{10}}_{b}\ \underbrace{\overbrace{t_9\ t_8\ t_7\ t_6\ t_5}^{l}\ \overbrace{t_4\ t_3\ t_2\ t_1}^{p}}_{s}\ ]$$

$batch$ $\qquad$ n



Metodología para Algoritmos Índice-Dígito

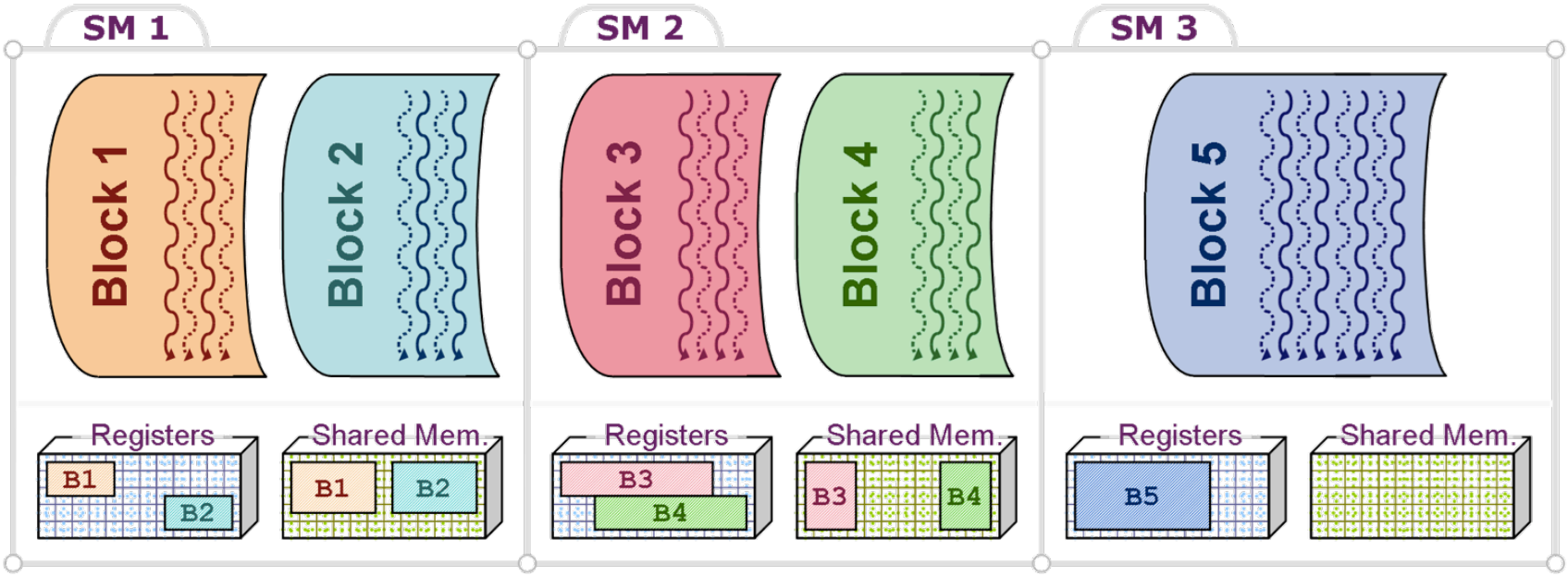
***Fase 1**: Análisis de recursos. Obtención de Resource Factors (I)*

Obtención Resource Factors

El rendimiento depende en el balance correcto entre el gran número de threads y el uso eficiente de los recursos compartidos en el hardware.

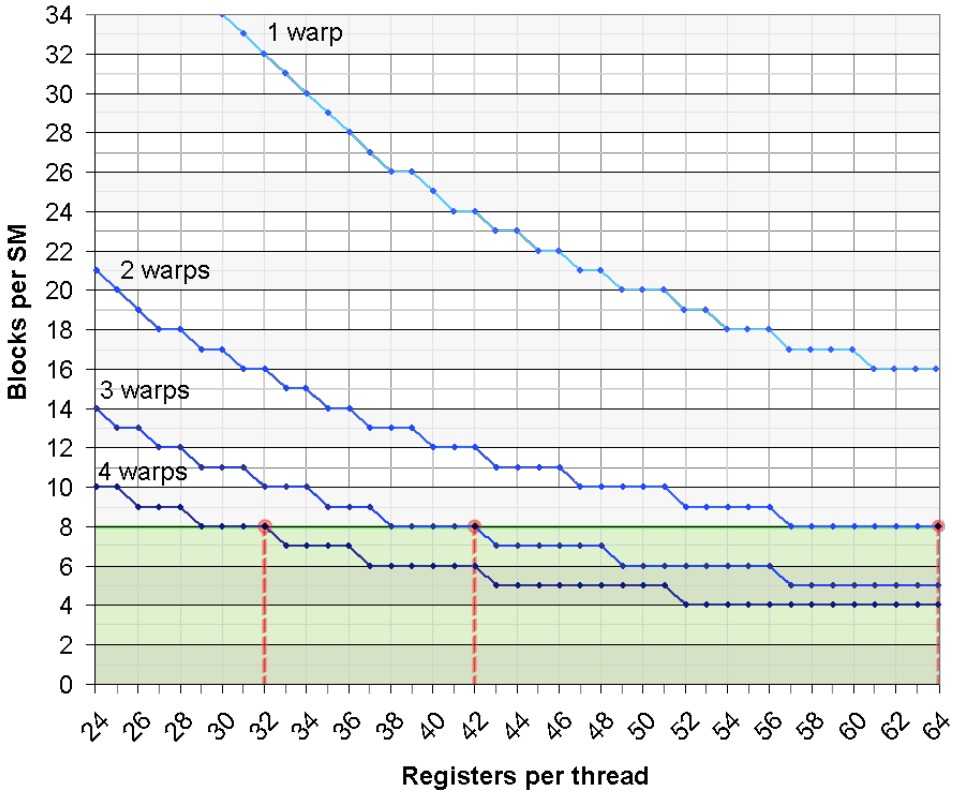
Dos principales niveles de paralelismo:

- SM warp parallelism
- SM block parallelism



Metodología para Algoritmos Índice-Dígito

***Fase 1**: Análisis de recursos. Obtención de Resource Factors (II)*



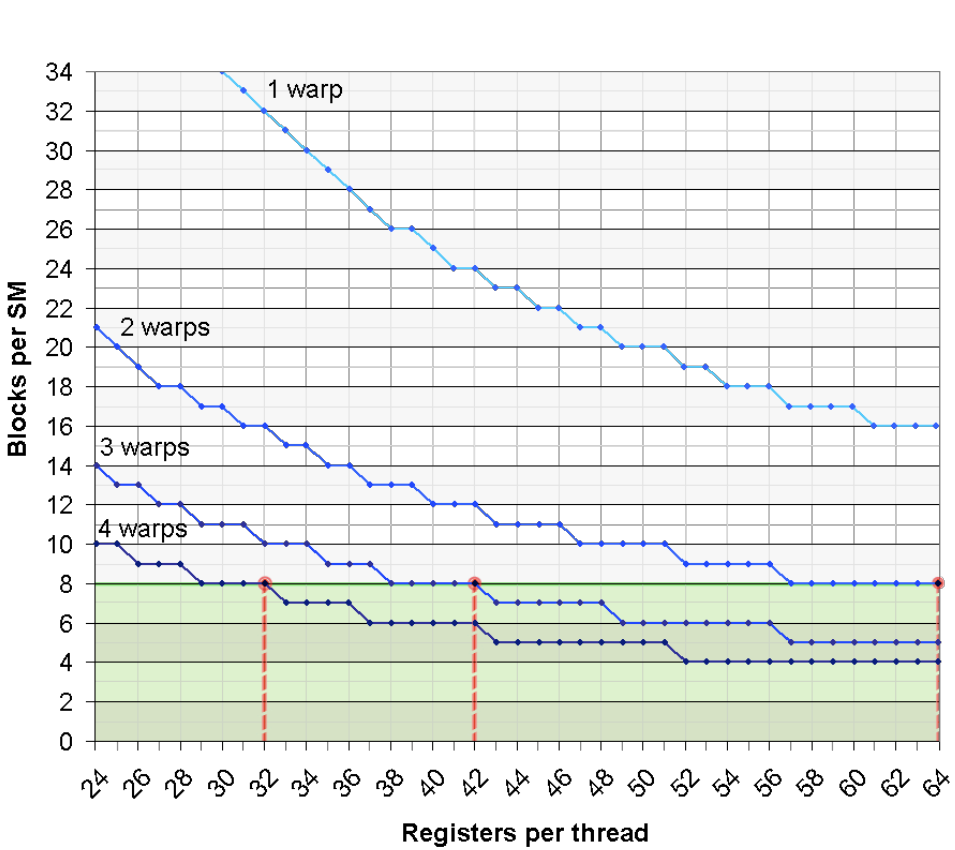
Recursos compartidos

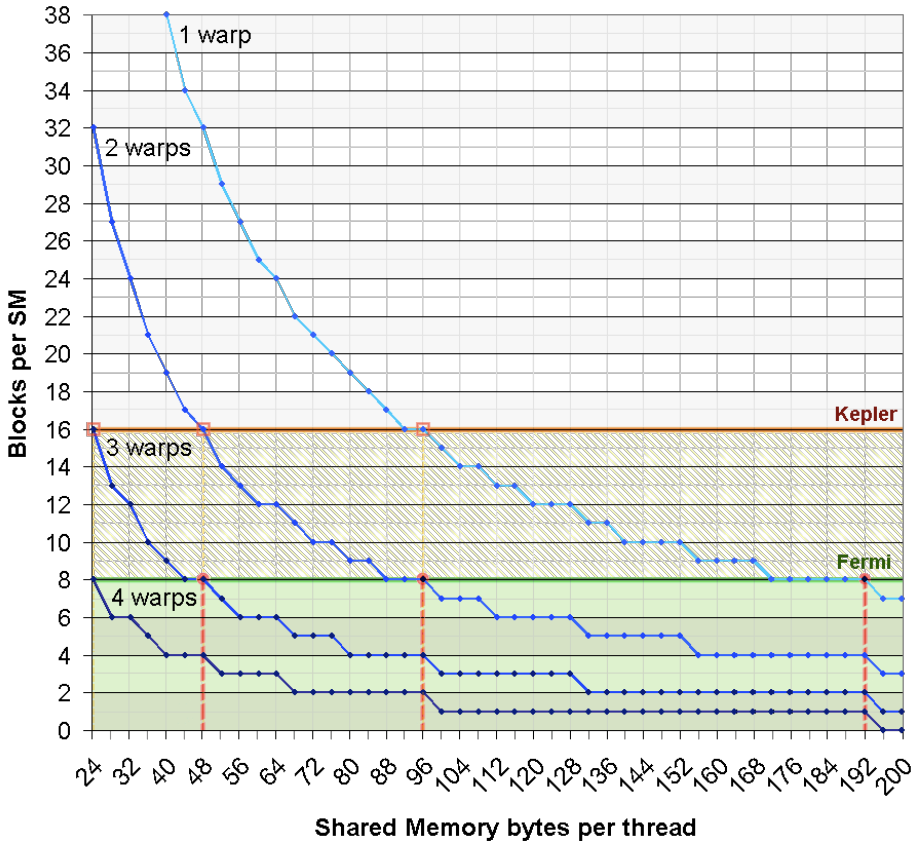
Los registros y la memoria compartida reservada pueden limitar el nivel de paralelismo.

En lo posible, intentaremos favorecer el máximo número de bloques por SM.

Metodología para Algoritmos Índice-Dígito

***Fase 1**: Análisis de recursos. Obtención de Resource Factors (II)*





Recursos compartidos

Los registros y la memoria compartida reservada pueden limitar el nivel de paralelismo.

En lo posible, intentaremos favorecer el máximo número de bloques por SM.

Metodología para Algoritmos Índice-Dígito

***Fase 1**: Análisis de recursos. Obtención de Resource Factors (III)*

Ajuste de parámetros

Los parámetros p, s, l pueden ser ajustados en base a las características del hardware.

El tamaño del radix será determinado por p, siendo a lo sumo $radix\text{-}2^p$.

El objetivo es elegir los parámetros p, s, l óptimos para maximizar el rendimiento.

Las cadenas de operadores se diseñan en base a los parámetros elegidos.

	Warps per block	Registers per thread	Shared mem. bytes/thread	SM warp occupancy
Fermi	1	63	192	16.6%
	2	63	96	33.3%
	3	42	48	50.0%
	4	32	24	66.6%
Kepler	1	63 / 128	96	25.0%
	2	63	48	50.0%
	3	42	24	75.0%
	4	32	12	100.0%

Registers per thread → p

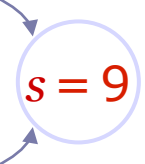
Shared mem. bytes/thread → s

Warps per block → l

❺ Diseño Eficiente de Algoritmos Índice-Dígito para GPUs

Fase 1: Análisis de recursos. Obtención de Resource Factors (IV)

	Warps per block	Registers per thread	Shared mem. bytes/thread	SM warp occupancy
Fermi	1	63	192	16.6%
	2	63	96	33.3%
	3	42	48	50.0%
	4	32	24	66.6%
Kepler	1	63 / 128	96	25.0%
	2	63	48	50.0%
	3	42	24	75.0%
	4	32	12	100.0%



Ajuste de parámetros

Dos configuraciones posibles:

$l = 5 \Rightarrow p = s - l = 9 - 5 = 4$

$l = 6 \Rightarrow p = s - l = 9 - 6 = 3$

Para resolver problemas hasta $N = 4096$ en un único kernel extendemos l, cambiando paralelismo de bloque por paralelismo de thread. Las configuraciones finales serían:

$(p, s, l) = \{ (4, 9, 5)\ \ (3, 9, 6)\ \ (4, 12, 8) \}$

Metodología para Algoritmos Índice-Dígito

***Fase 1**: Análisis de recursos. Obtención de Resource Factors (V)*

Alg	n	p	GFLOPS	Threads /block	Reg	Shared memory	Gl. mem. bandw.	Instruction replay (%)	Occupancy (%)
ID-FFT.V1	2	4	***104.75***	32	52	2112	162.50	4.4	∼16
	3	4	***157.09***		50		162.42	3.2	
	4	4	***209.64***		49		162.53	2.2	
ID-FFT.V2	5	4	***261.82***	32	42	2112	162.56	1.7	∼16
	6	4	***312.78***		45		162.11	0.9	
	7	4	363.29		47		161.40	0.5	
	8	4	411.26		44		159.95	0.3	
ID-FFT.V3	7	3	***365.30***	64	28	2304	162.45	1.2	∼33
	8	3	***418.59***		29		162.45	0.8	
	9	3	***471.97***		30		163.05	0.4	
ID-FFT.V4	9	4	413.12	256	46	17408	143.01	5.4	∼33
	10	4	***448.35***		45		139.19	5.0	
	11	4	***475.01***		46		134.18	4.6	
	12	4	***492.43***		45		126.26	1.7	

Ajuste de parámetros

Para optimizar todos los casos creamos cuatro versiones del algoritmo.

Para algunos tamaños existen dos configuraciones posibles.

En igualdad de condiciones, se elige la que tenga el mayor paralelismo a nivel de warp.

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

> *Using the mapping vector and the computation/permutation operators is possible to model any ID-algorithm*

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

Mapping Vector: A compact representation of data distribution on the system memory hierarchy.

At the beginning and at the end, data reside in global memory. During execution, data are moved among different GPU resources.

$$\lfloor \underbrace{t_{n+batch} \cdots t_{s+1}}_{b} \overbrace{\underbrace{t_s \cdots t_{p+1}}_{l} \underbrace{t_p \cdots t_1}_{p}}^{s} \rfloor$$

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

Example: { p=2, l=(6,1), b=(8,0) }

4 elements per thread, 128 threads per block,
512 elements in shared memory,
256 elements per problem, 256 blocks,
2 problems per block, 512 problems simultaneously.

$$\left[\underbrace{t_{17}\cdots t_{10}}_{b}\overbrace{\underbrace{t_{9}\cdots t_{3}}_{l}\underbrace{t_{2}\cdots t_{1}}_{p}}^{s}\right]$$

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

Example: { *p=2, l=(6,1), b=(8,0)* }

6th element within third problem is located at:

$$\left[\underbrace{00000001}_{b}\overbrace{\underbrace{00000010}_{l}\underbrace{1}_{p}}^{s}\right]$$

TreadIdx=(1,0) ; register[1] of its thread ;
6th element in shared memory ; BlockIdx=(1,0)

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

Two types of operators: computation and permutation.

- *Node operator (computation):* Ψ_i^r *with* $1 \le i \le n$ *reads sets of r data whose position differs in their i-digit and writes r results.*

0: 00000*0*
1: 00001*1*
2: 00010
3: 00011

Ψ_1^2

Each algorithm defines its own node operator behavior: Addition, Gauss reduction, sorting, ...

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

  - *Perfect unshuffle operator (permutation):* $\Gamma_{i,j}$ $i \geq j$ *performs a cyclic shift to the right between the i and j-th digits*

$$\Gamma_{i,j}\left[t_n \cdots t_1\right] = \left[t_n \cdots t_{i+1} \underline{t_j t_i \cdots t_{j+1}} t_{j-1} \cdots t_1\right]$$

We also define a generalization $\Gamma^m{}_{i,j}$ performing *m* consecutive shift operations. For example:

$$\Gamma^2{}_{7,2}\left[t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1\right] = \left[t_8 \underline{t_3 t_2 t_7 t_6 t_5 t_4} t_1\right]$$

Metodología para Algoritmos Índice-Dígito

- *Operators string manipulation* phase

  - *General unshuffle operator (permutation):* $\Gamma_{i,j,k,l}$ *performs a cyclic shift to the right between two digits subfields {i..j} and {k..l}*

$$\Gamma_{i,j,k,l}\left[t_n \cdots t_1\right] = \left[t_n \cdots t_{i+1} \underline{t_l t_i \cdots t_{j+1}} t_{j-1} \cdots t_{k+1} \underline{t_j t_k \cdots t_{l+1}} t_{l-1} \cdots t_1\right]$$

For example:

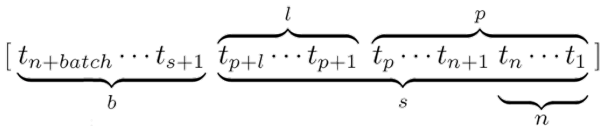
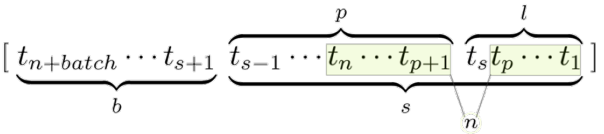
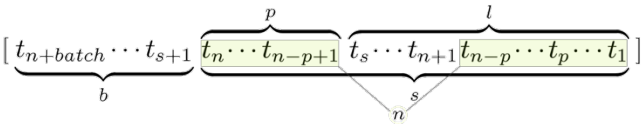
$$\Gamma_{8,6,2,1}\left[t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1\right] = \left[\underline{t_1 t_8} t_7 t_5 t_4 t_3 \underline{t_6 t_2}\right]$$

Metodología para Algoritmos Índice-Dígito

***Fase 2**: Cadenas optimizadas, ID-FFT*

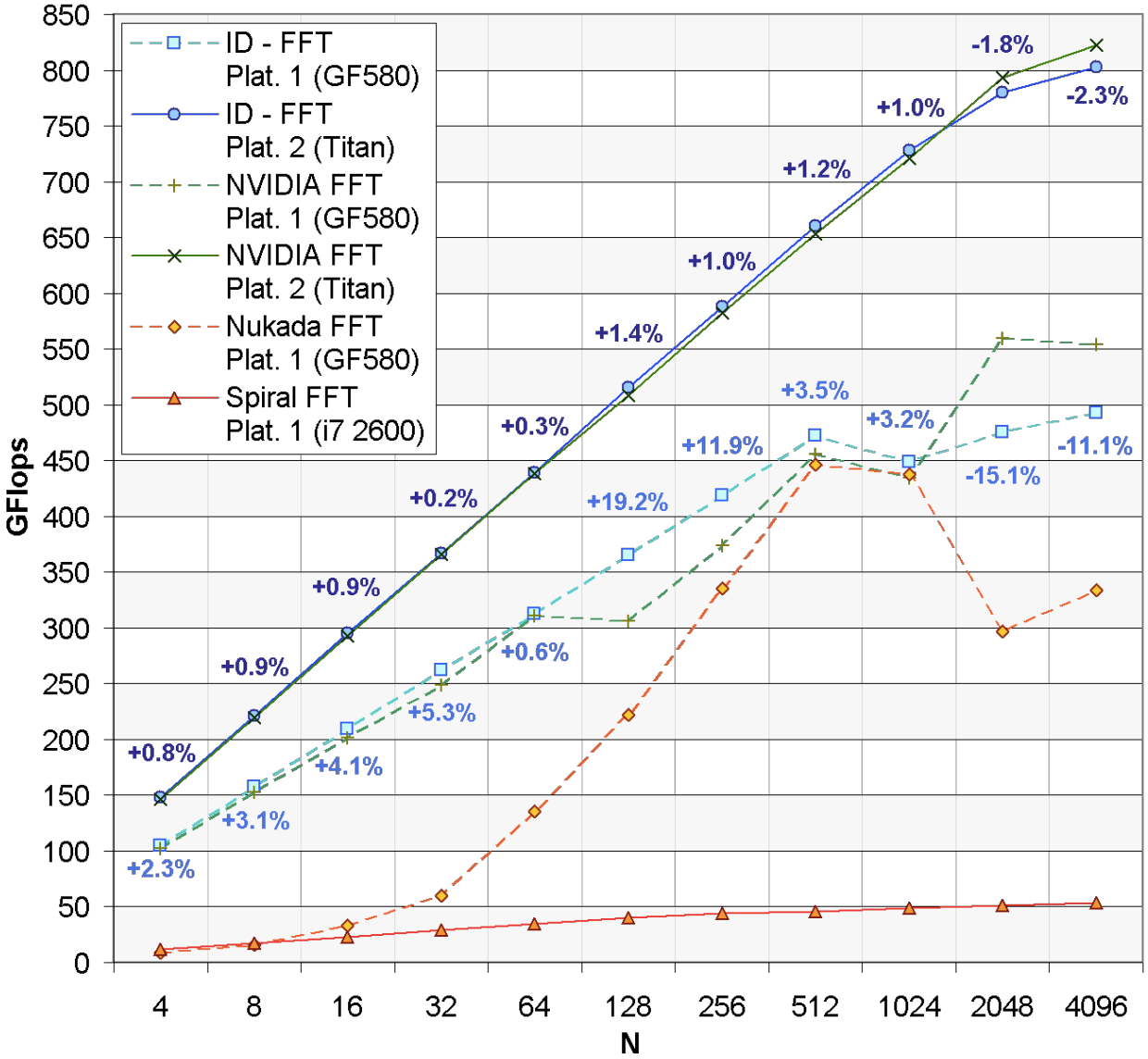
Transformada de Fourier

- Expresión general: $\prod_{i=1}^{\lfloor n/r \rfloor} \Gamma^r_{n,n-i\cdot r+1}\ \rho_{n,n-r+1}\ B^r_{n-r+1}$

Rango	Parámetros (p, s, l)	Expresión particular	Mapping vector
$n \le p$	(4, 9, 5)	$\rho_{n,1}\ B^n_1$	
$p < n \le 2p$	(4, 9, 5)	$\left(\rho_{s-p+m,s-p+1}\ B^m_{s-p+1}\right)\Gamma^p_{s,s-p+1,p,1}$ $\left(\rho_{s,s-p+1}\ B^p_{s-p+1}\right)$	
$2p < n \le 3p$	(3, 9, 6)	$\left(\rho_{s,s-p+1}\ B^p_{s-p+1}\right)\Gamma^p_{s,s-2\cdot p+1}$ $\left(\rho_{s-p+m,s-p+1}\ B^m_{s-p+1}\right)\Gamma^p_{s,s-3\cdot p+1}$ $\left(\rho_{s,s-p+1}\ B^p_{s-p+1}\right)$	
$2p < n \le 3p$	(4, 12, 8)		

Metodología para Algoritmos Índice-Dígito

***Análisis de Rendimiento**: ID-cFFT*



GeForce Titan

Max: 803 GFlops

Promedio: +0.8%

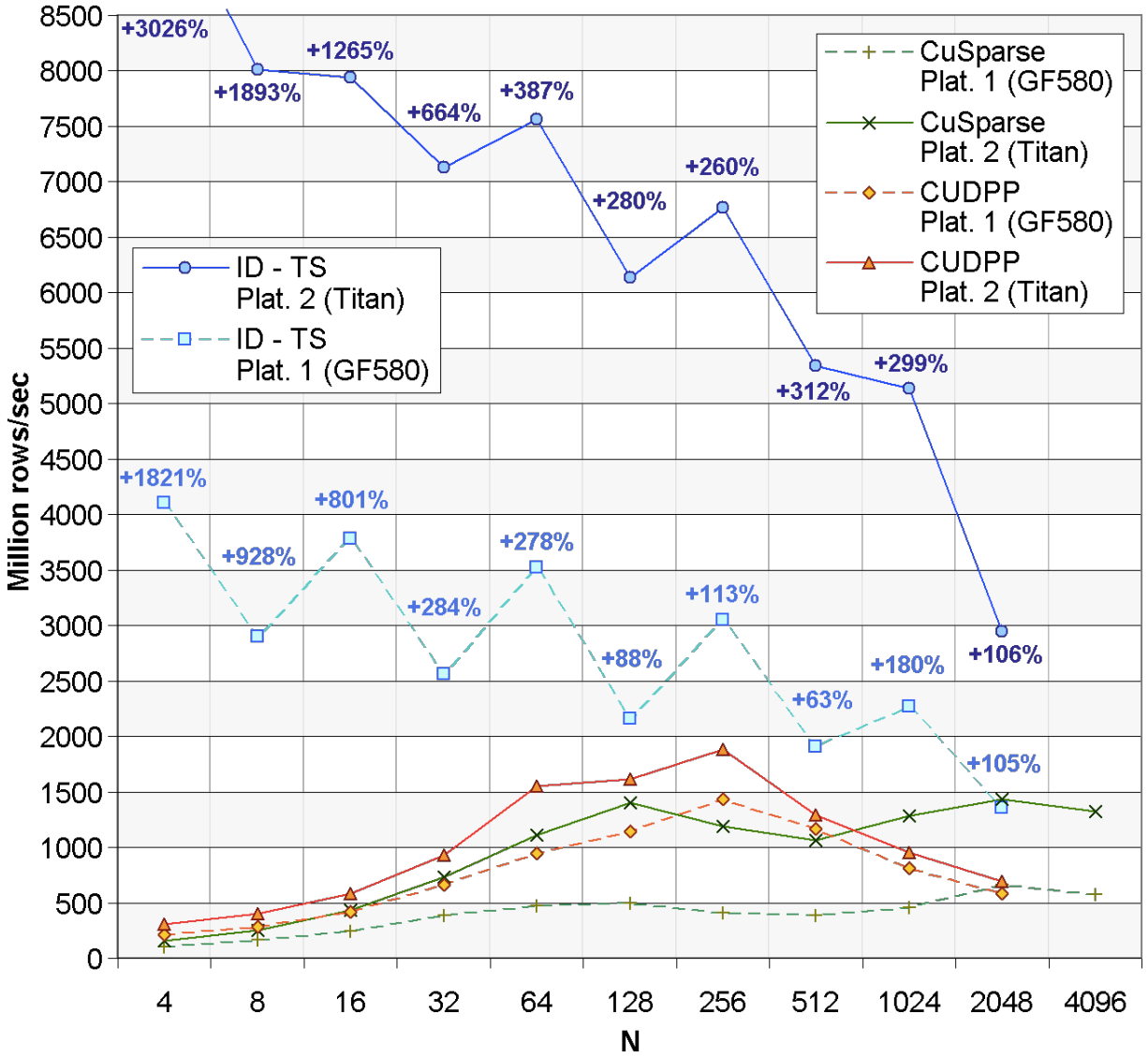
GeForce 580

Max: 492 GFlops

Promedio: +5.9%

Metodología para Algoritmos Índice-Dígito

***Análisis de Rendimiento**: ID-TS*



GeForce Titan

Max: 9643 MRows

Promedio: +447%

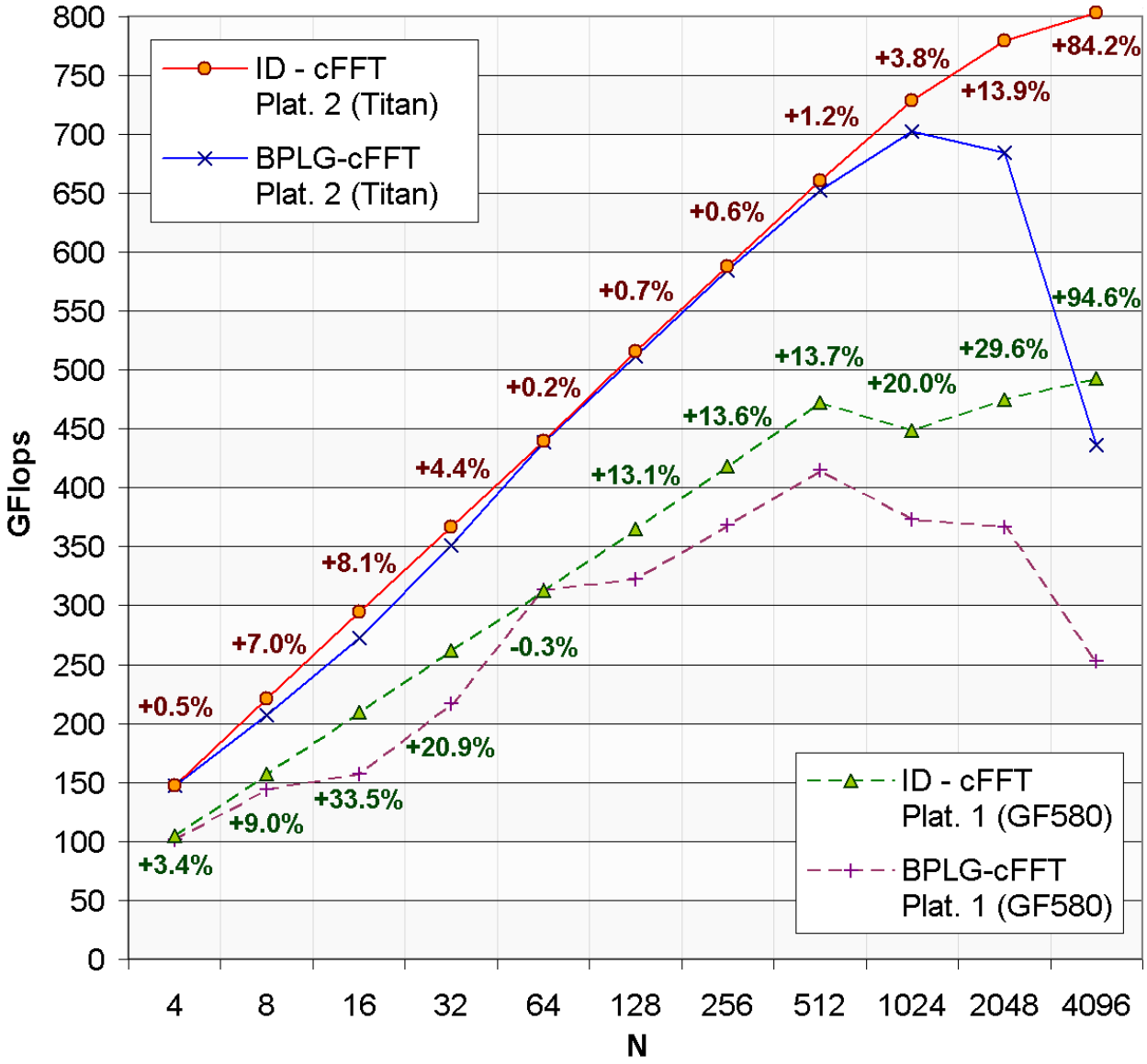
GeForce 580

Max: 4105 MRows

Promedio: +238%

Comparación Índice-Dígito vs. BPLG

BPLG-cFFT vs ID-cFFT



GeForce Titan

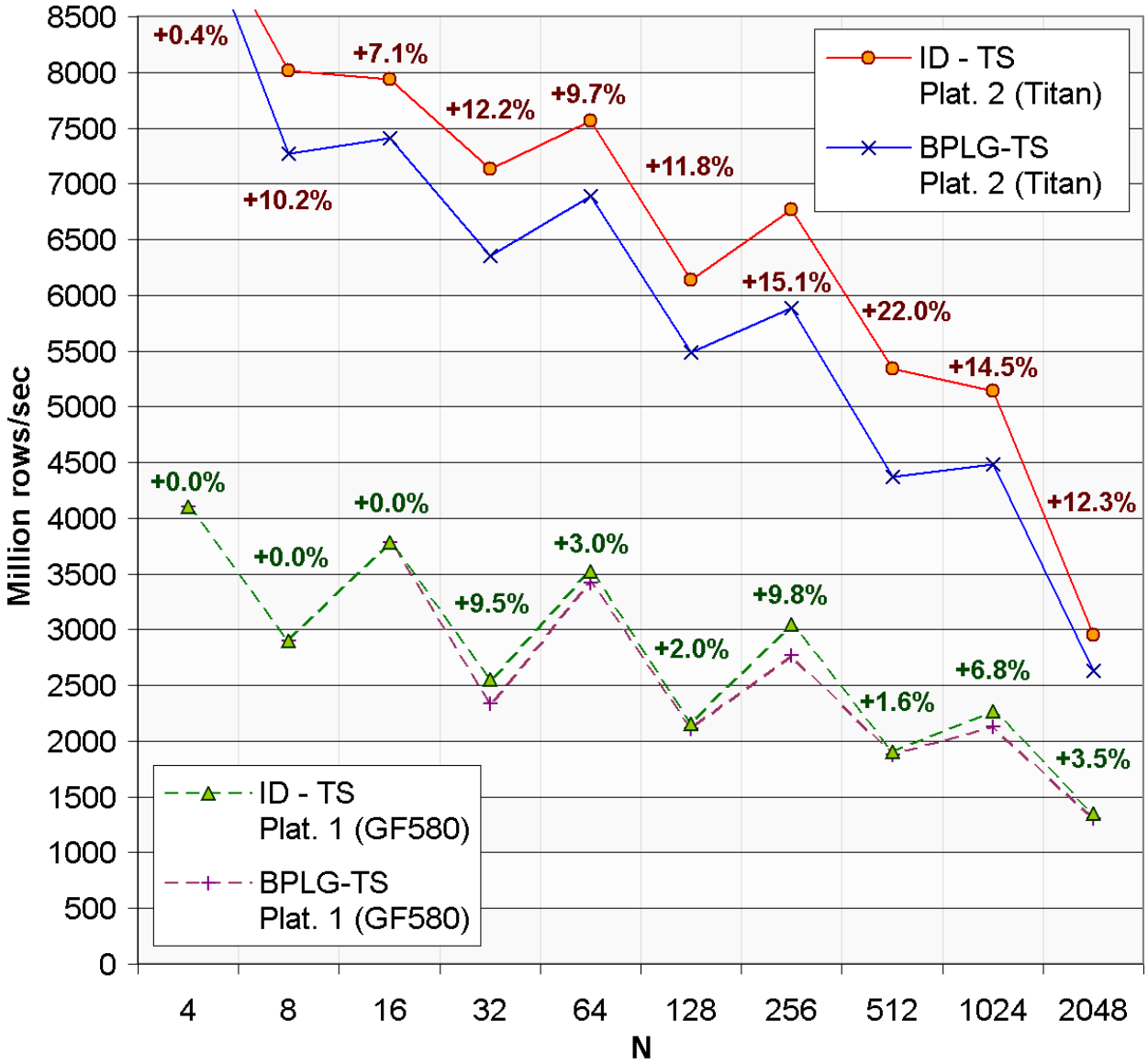
Promedio: +11.3%

GeForce 580

Promedio: +22.8%

Comparación Índice-Dígito vs. BPLG

BPLG-TS vs ID-TS



GeForce Titan

Promedio: +11.5%

GeForce 580

Promedio: +3.6%

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

Metodología para Algoritmos Índice-Dígito:

Caso 1: SM>=N

Caso 2: SM<=N<=GM

Nueva estrategia para Algoritmos Parallel-Prefix

Trabajo Futuro

Metodología para Algoritmos Índice-Dígito (caso 2)

The problem is larger than shared memory but still can be stored in global memory

Data distributed among several blocks. Each block computes a portion of the solution.

It is necessary a synchronizing mechanism for exchanging partial results among blocks

Nowadays, **there is not any global synchronization barrier for CUDA**

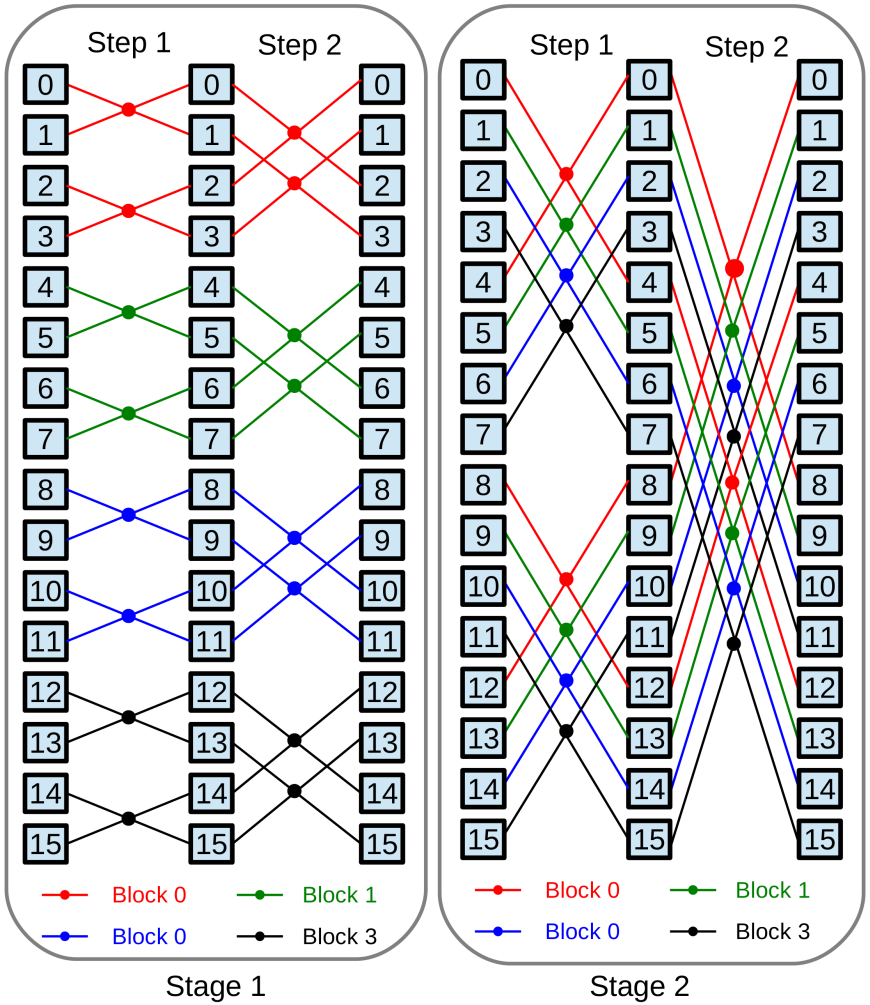
Metodología para Algoritmos Índice-Dígito (caso 2)

But there are some strategies:

- **Multi-Stage strategy**. Using several *kernels* (hence forth stages). It uses global memory as information exchanging mechanism.
- *CUDA Dynamic Parallelism*. A kernel from GPU launches other kernels. Generated code runs slower: Relocatable device code + local memory as stack.
- *Persistent threads*. As many blocks as the device can simultaneously run (otherwise deadlock). Each block sets a flag, sleeps and waits for the flag reset. A master block will wait for all blocks, doing a flag reset.

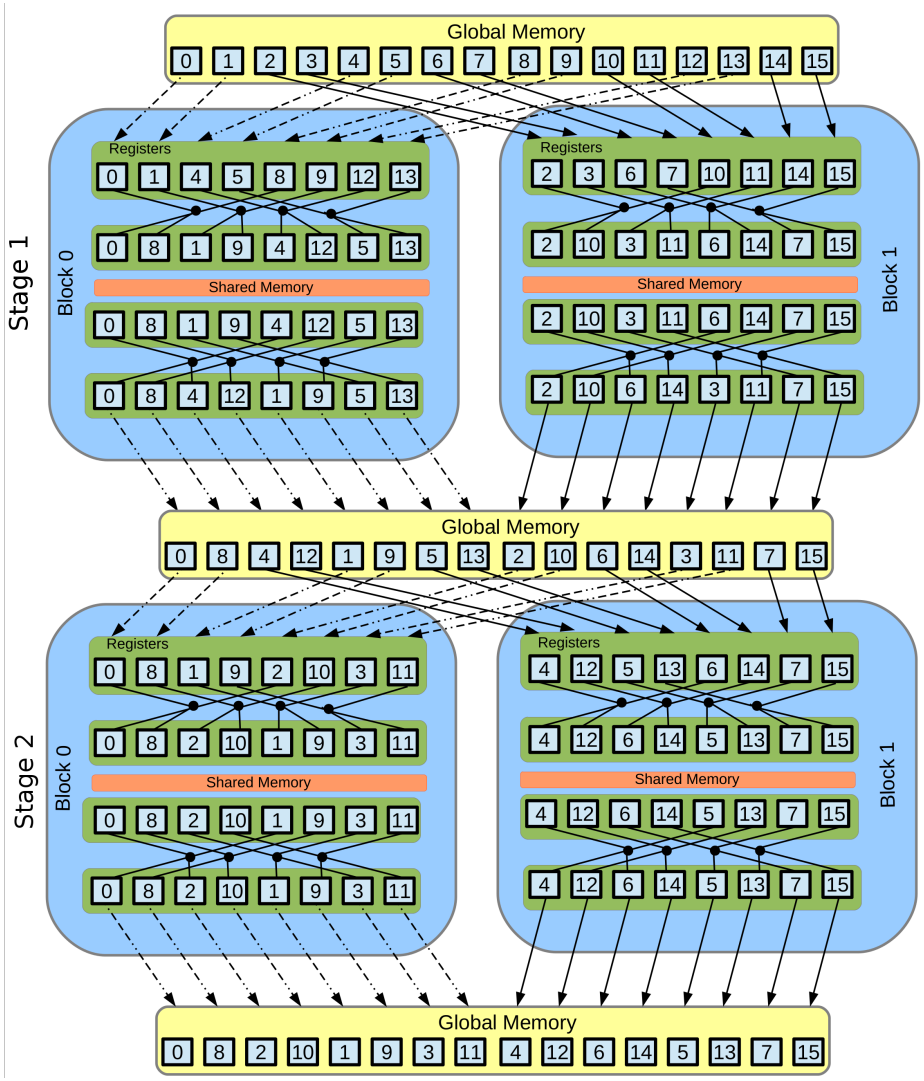
Metodología para Algoritmos Índice-Dígito (caso 2)

Multi-Stage strategy: tridiagonal solver

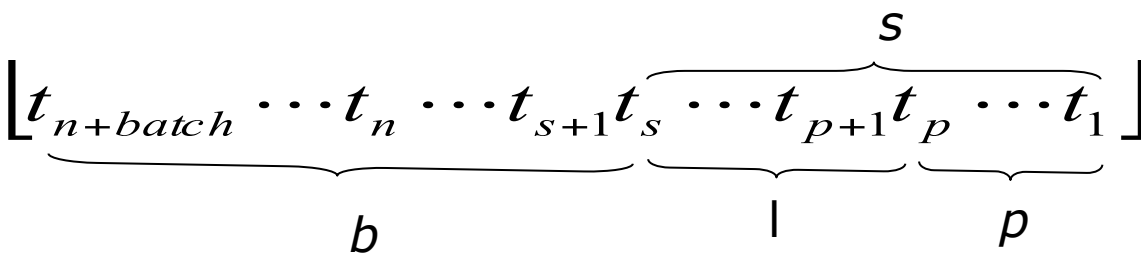


Metodología para Algoritmos Índice-Dígito (caso 2)

Multi-Stage strategy: FFT



Metodología para Algoritmos Índice-Dígito (caso 2)



- Using more than 1 stage forces to introduce a new parameter in our methodology: *m*
- This parameter represents the number of stages:

$$m = \left\lceil \frac{n}{s} \right\rceil$$

- Also introduces a new Premise:
  - *Premise 3: The minimization of m.* Global memory data exchanges are slower than using other memories. Furthermore, each kernel invocation implies an overhead.
- Thus, *s* must be as greater as possible.

Metodología para Algoritmos Índice-Dígito (caso 2)

- Combining all premises is not easy:
  - *Large s sizes implies good warp parallelism but reduces the number of active blocks, decreasing block parallelism.*
  - *Large p values reduces the number of steps per stage but also can generate register spilling and decreases the warp parallelism.*
  - *Raising s in order to minimize m implies reducing block parallelism.*

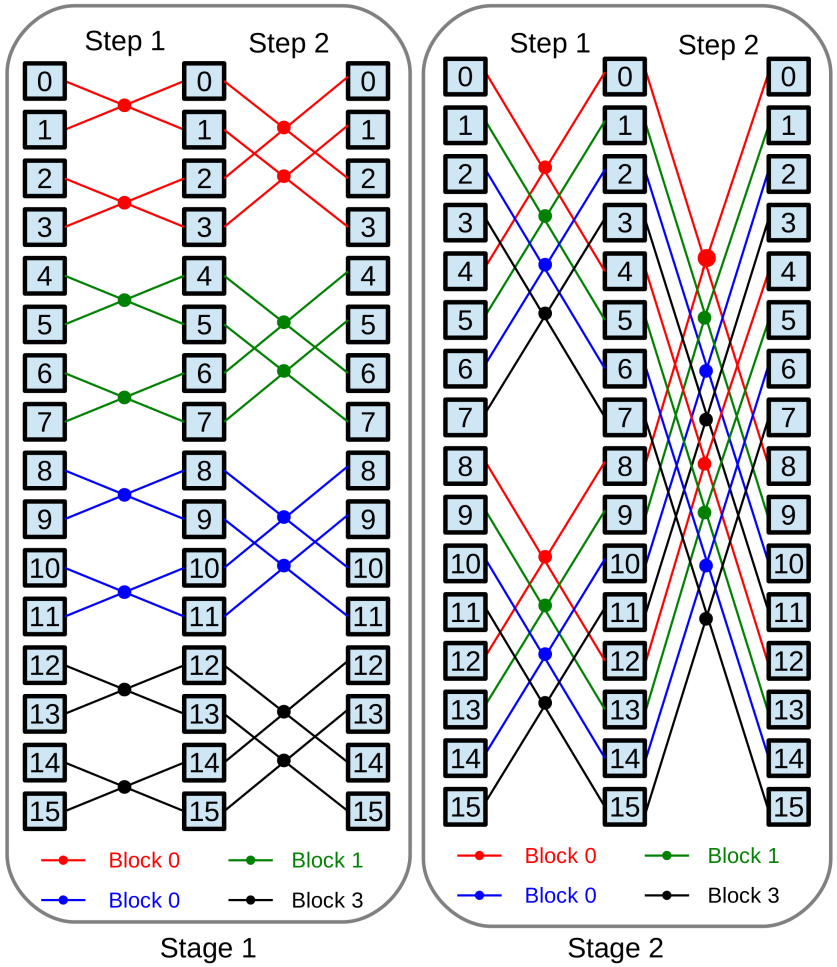
Case 2 Example: TS Solver

- Wang&Mou TS Solver.

Adjancecy property can be applied in Stage 1 but not in Stage 2.

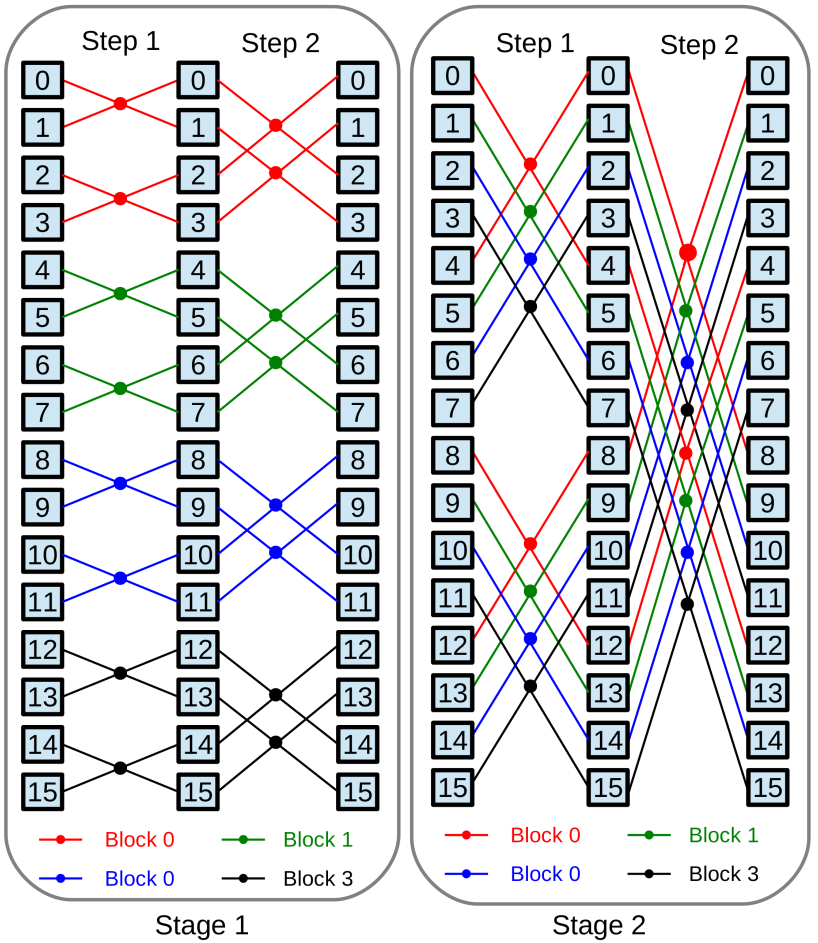
In Stage 2 is necessary to store the whole triad:

- *Stage 1 elements are equations float4 type -> 16 bytes*
- *Other Stage elements are triad of equations 3xfloat4 type -> 48 bytes*
- *n* is splitted in $s_1 + (m-1) \cdot s_2$ instead of $m \cdot s$



Case 2 Example: TS Solver

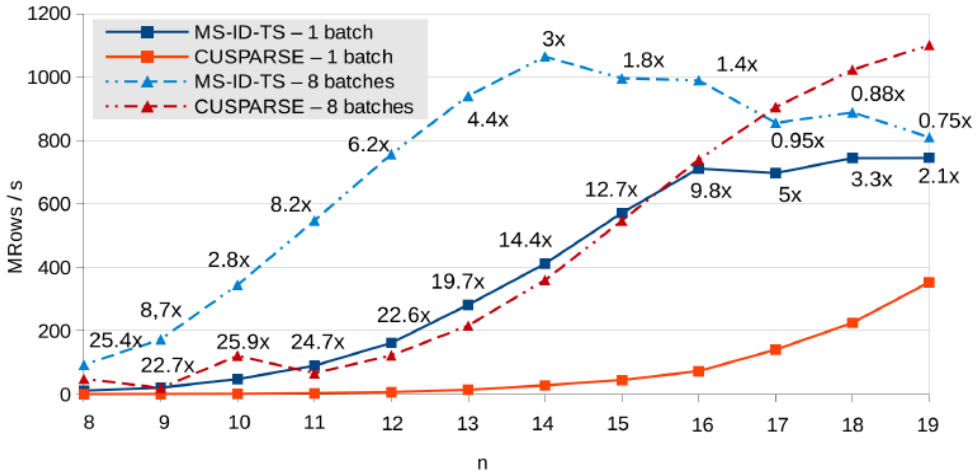
- The first kernel's shared memory can hold much more elements than the shared memory of other stages.
- It is better to process as many steps as possible in stage 1, it is cheaper.



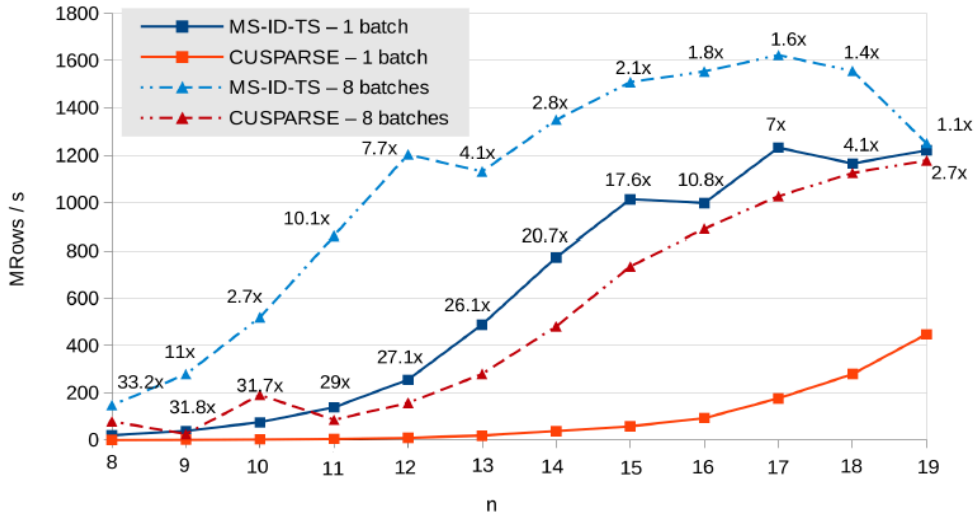
Case 2 Experimental Results

	Platform 1, Platform 2	*Platform 3*
CPU	Intel Xeon E5-2660 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	64 GB DDR3 1600	8 GB DDR3 1333
OS	CentOS 6.4	Ubuntu 12.04 LTS
Compiler	GCC 4.4.7	GCC 4.6.3
GPU	*Nvidia Tesla K20, Nvidia Tesla K40*	*Nvidia GeForce GTX980*
Driver	340.58, SDK 6.0	343.22, SDK 6.5

Case 2: TS Solver

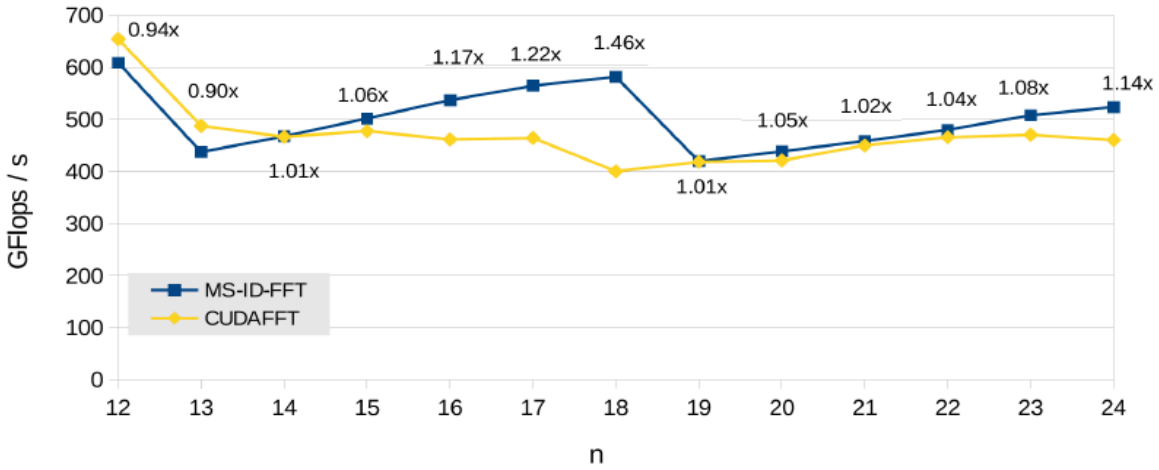


(b) Platform 2

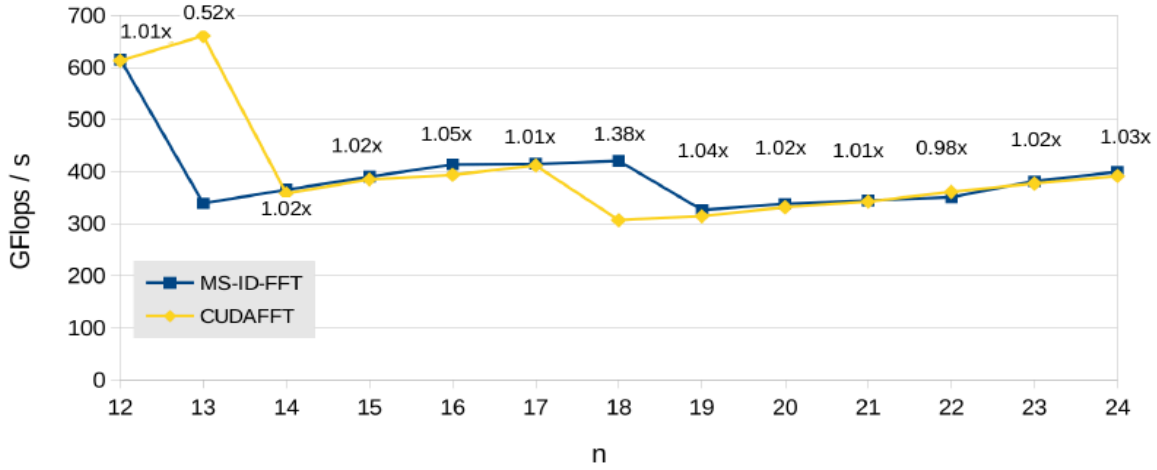


(c) Platform 3

Case 2 Example: FFT



(b) Platform 2



(c) Platform 3

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- Metodología para Algoritmos Índice-Dígito
- **Nueva estrategia (general) para Algoritmos Parallel-Prefix**

Trabajo Futuro

Estrategia (general) para Algoritmos Parallel-Prefix

- *GPU Resource Utilization Analysis*
- *CUDA Kernel Optimization*
- *Performance Parameter Tuning*

Estrategia (general) para Algoritmos Parallel-Prefix

- *Tuning Parameters*

Problem Parameters

$N = 2^n$	Problem size.
$G = 2^{batch}$	Number of problems being solved simulteneously.

GPU Performance Parameters

$S = 2^s$	Number of shared-memory elements per block.
$P = 2^p$	Number of elements stored in registers per thread.
$B = 2^b$	Number of thread blocks executed per GPU
$L = 2^l$	Number of threads that compose a block, where $s = p + l$
L_C	Number of problems being solved per block, where $B = \frac{G}{L_G}$ and $L = \frac{N}{P} \times L_C$
W_{SM}	Number of warps per SM
W^{max}	Maximum number of warps per SM
W_B	Number of warps per thread block
B_a	Number of active thread blocks simultaneously executed per SM
B^{max}	Maximum number of thread blocks per SM
B^r	Number of thread blocks per SM limited by the registers available
B^s	Number of thread blocks per SM limited by the shared memory available

Estrategia (general) para Algoritmos Parallel-Prefix

- *CUDA Kernel Optimization: uses BPLG skeletons (building blocks)*
  - *CUDA features for improving skeleton implementations:*
    - *Efficient-Index calculation avoiding non-uniform access*
    - *Hybrid communication strategy inside a block*
    - *Specialized skeletons*

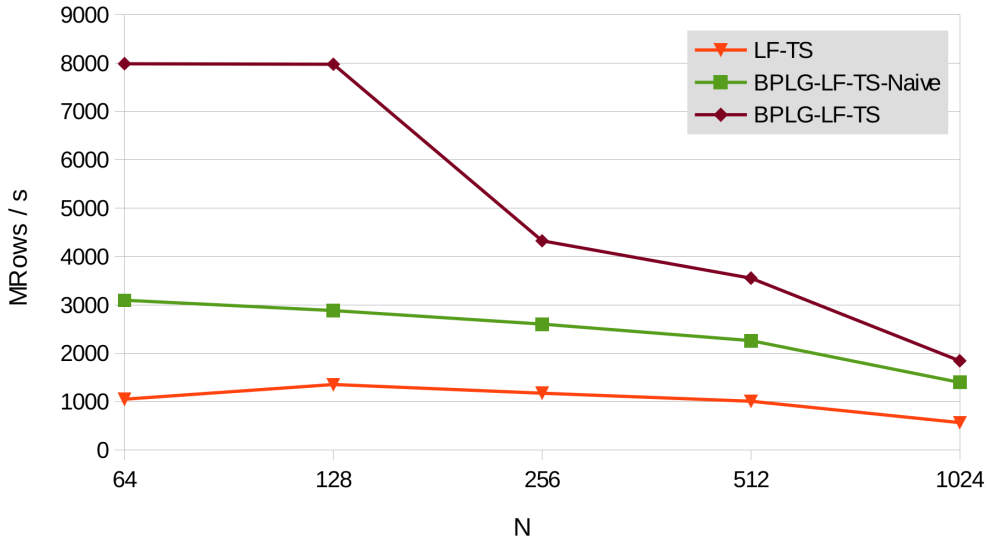
Estrategia (general) para Algoritmos Parallel-Prefix

- *Performance parameter tuning*
  - *Maximizing number of warps and blocks per SM*

Archit.	Warps per block	Regs per thread	Shared memory per block	Warp occupancy (W_{SM}/W^{max})	SM blocks (B_a)
Kepler	1	128	3072	25%	16
	2	64	3072	50%	16
	4	32	0	100%	16
	4	40	0	75%	12
	4	**32**	**3072**	**100%**	**16**
	4	32	4096	75%	12
	8	32	6143	100%	8
	8	32	8192	75%	6
	16	32	12285	100%	4
	16	32	16384	75%	3
	32	32	24576	100%	2
Maxwell	1	64	2048	50%	32
	2	32	0	100%	32
	2	40	0	75%	24
	2	**32**	**2048**	**100%**	**32**
	4	32	4096	100%	16
	8	32	8192	100%	8
	16	32	16384	100%	4
	32	32	32768	100%	2

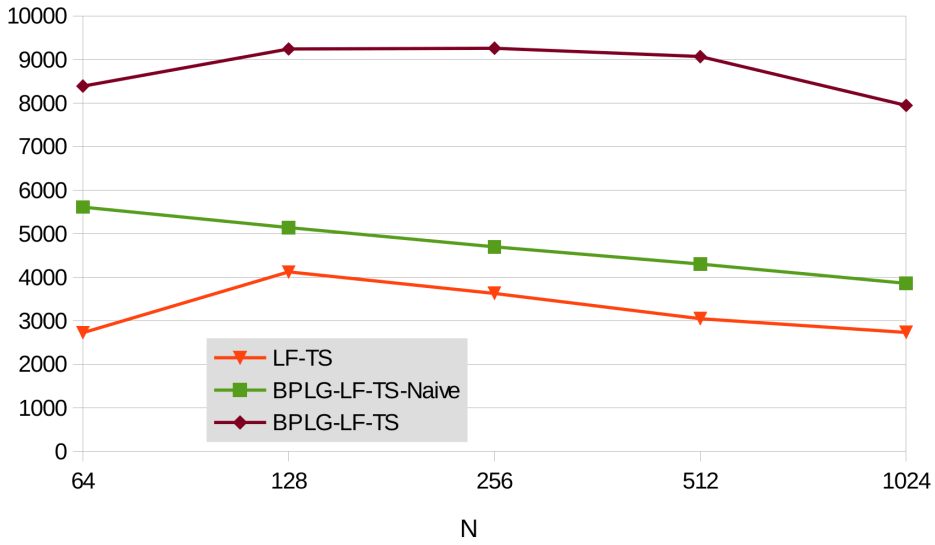
Estrategia (general) para Algoritmos Parallel-Prefix

- *Performance results:tridiagonal solver*
  - *BPLG-LF-TS: based en Ladner-Fischer pattern*



a) Platform 1

Kepler

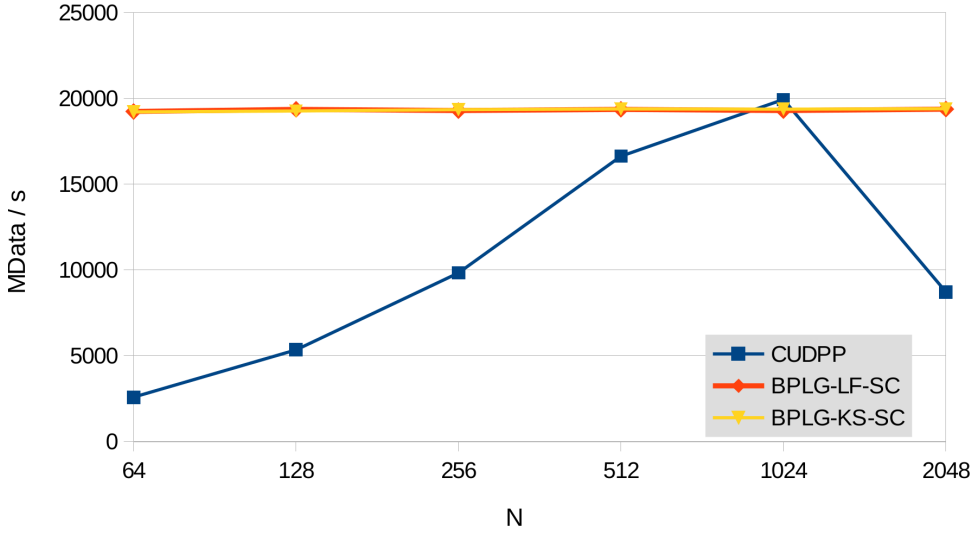


b) Platform 2

Maxwell

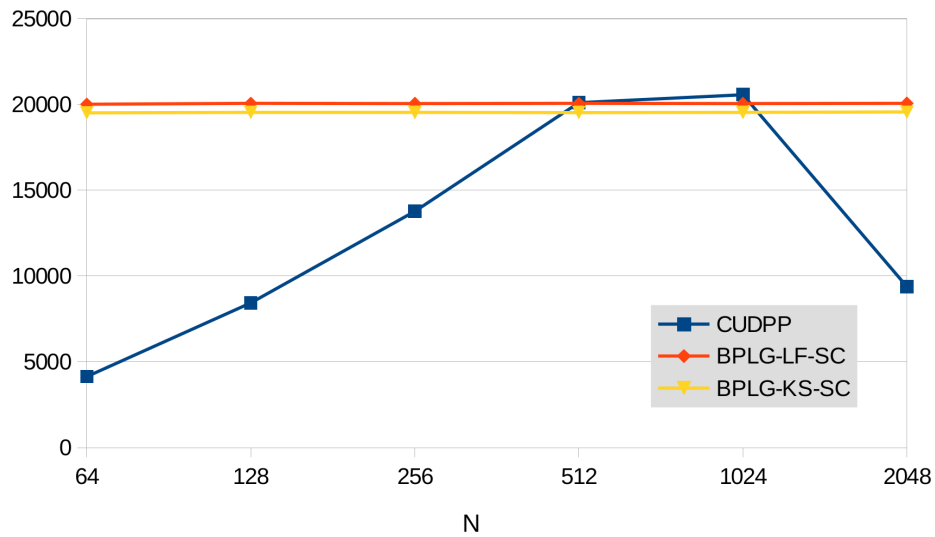
Estrategia (general) para Algoritmos Parallel-Prefix

- *Performance results: scan primitive*
  - *BPLG-LF-SC: based en Ladner-Fischer pattern*
  - *BPLG-KS-SC: based en Kogge-Stone pattern*



a) Platform 1

Kepler



b) Platform 2

Maxwell

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- Metodología para Algoritmos Índice-Dígito
- Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

- **Mobile devices**
- Aplicaciones

Mobile devices: Nvidia ARM SoCs

NVIDIA ARM SoCs			
	Xavier	**Parker**	**Erista (Tegra X1)**
CPU	8x NVIDIA Custom ARM	2x NVIDIA Denver + 4x ARM Cortex-A57	4x ARM Cortex-A57 + 4x ARM Cortex-A53
GPU	Volta, 512 CUDA Cores	Pascal, 256 CUDA Cores	Maxwell, 256 CUDA Cores
Memory	?	LPDDR4, 128-bit Bus	LPDDR3, 64-bit Bus
Video Processing	7680x4320 Encode & Decode	3840x2160p60 Decode 3840x2160p60 Encode	3840x2160p60 Decode 3840x2160p30 Encode
Transistors	7B	?	?
Manufacturing Process	TSMC 16nm FinFET+	TSMC 16nm FinFET+	TSMC 20nm Planar

Tegra X1(Erista) SoC: CPU

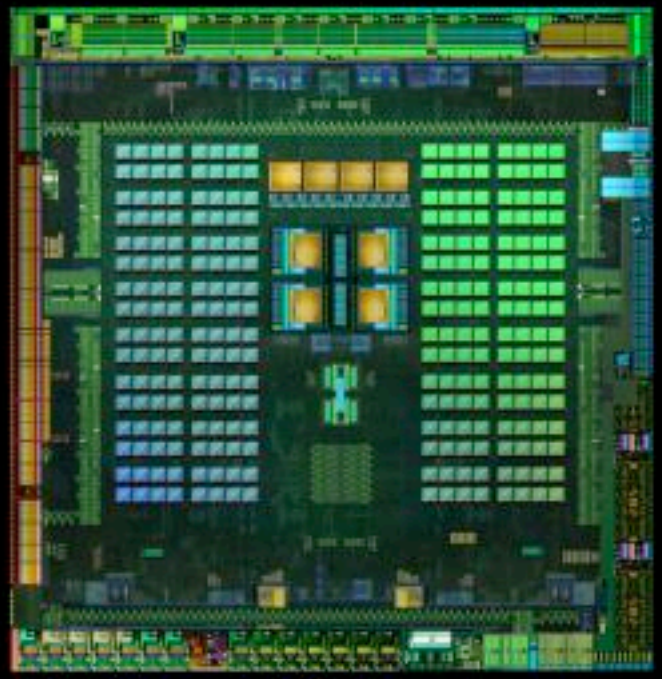
TEGRA X1
CPU CONFIGURATION

4 HIGH PERFORMANCE A57 BIG CORES

- 2MB L2 cache
- 48KB L1 instruction cache
- 32KB L1 data cache

4 HIGH EFFICIENCY A53 LITTLE CORES

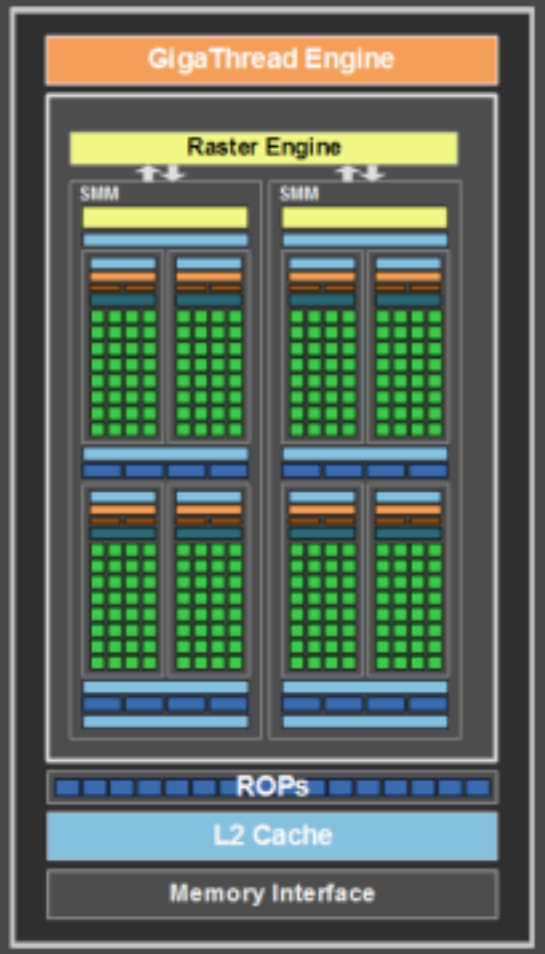
- 512KB L2 cache
- 32KB L1 instruction cache
- 32KB L1 data cache



Tegra X1(Erista) SoC: GPU

TEGRA X1 MAXWELL GPU

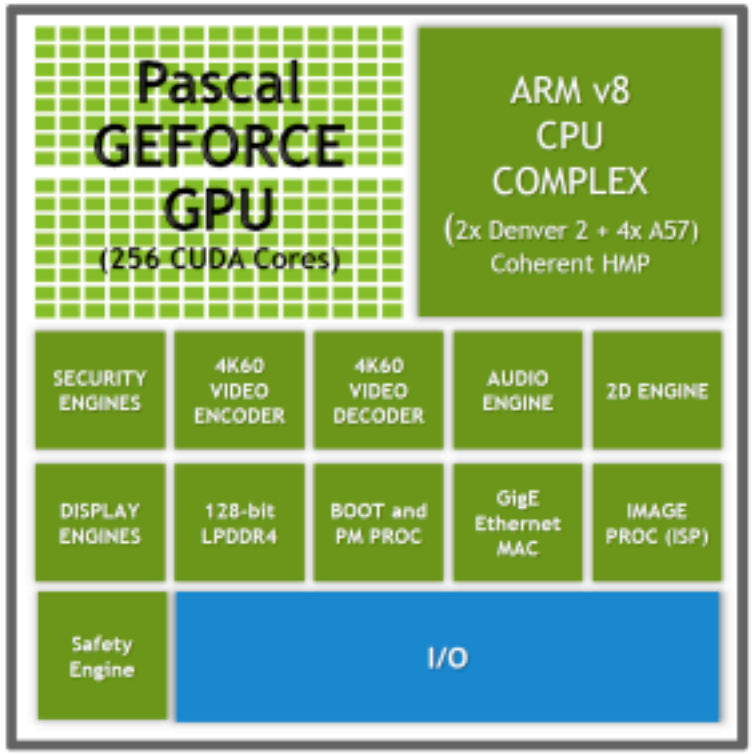
- 2x performance vs Tegra K1
- 2x perf/watt vs Tegra K1
- 2 SM
- 256 CUDA Cores
- 2 Geometry Units
- 16 Texture Units
- 16 ROP Units
- Maxwell Memory Arch
- 64-bit LPDDR4



Tegra X2(Parker) SoC

“PARKER”

NEXT-GENERATION SYSTEM-ON-CHIP

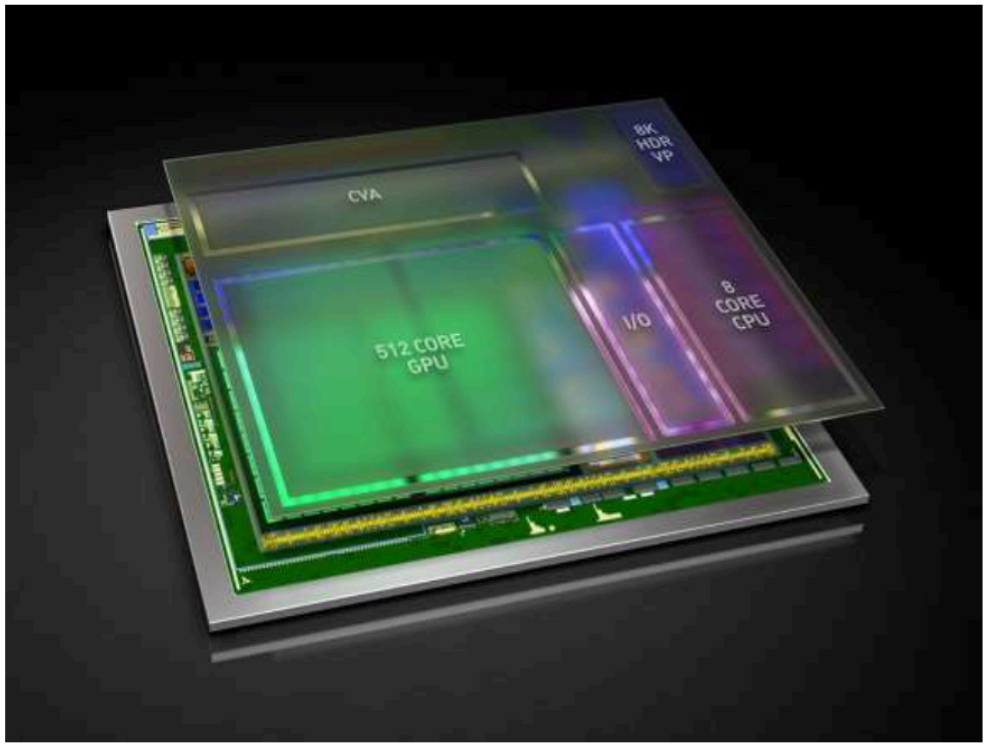


- NVIDIA’s next-generation Pascal graphics architecture
- NVIDIA’s next-generation ARM 64b Denver 2 CPU
- Functional safety for automotive applications
- Hardware-enabled virtualization architecture
- Improvements to SoC architecture to enable modularity and ASIC development efficiency
- Industry-leading 16nm FF process

Xavier SoC

- 7 Billion Transistors 16nm FF
- 8 Core Custom ARM64 CPU
- 512 Core Volta GPU
- New Computer Vision Accelerator CVA
- Dual 8K HDR Video Processors
- Designed for

  ASIL C Functional Safety



Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

Introducción

Diferentes estrategias de diseño para algoritmos Parallel-Prefix sobre GPUs

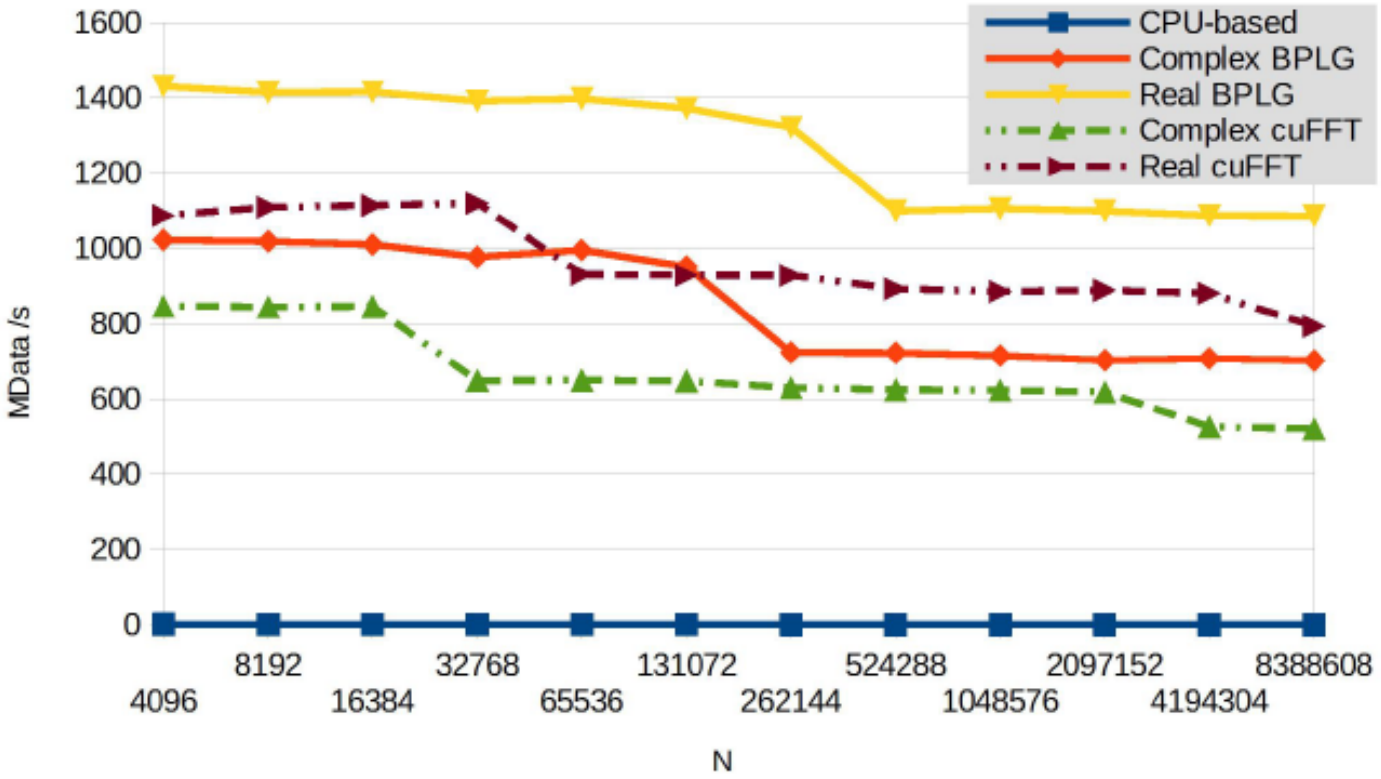
- BPLG: Una Librería Configurable para Algoritmos tipo Mariposa
- Metodología para Algoritmos Índice-Dígito
- Nueva estrategia (general) para Algoritmos Parallel-Prefix

Trabajo Futuro

- Mobile devices
- **Aplicaciones:**
  - **Multiplicación grandes números, Face detection**

Aplicaciones: Multiplicación grandes enteros

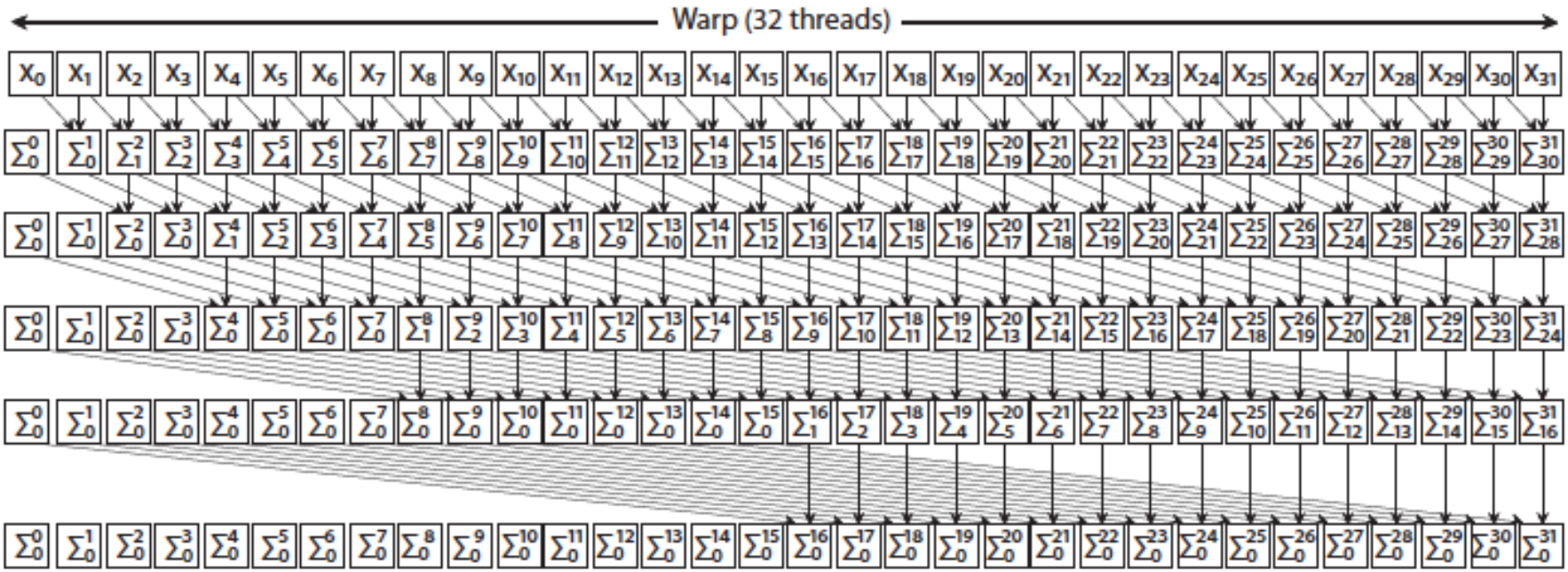
- **Multiplicación Polinómica** basada en FFT: algoritmo de Strassen
  - Máximo interés en **criptografía clave pública**
  - Algunos resultados preliminares utilizando FFT-ID (BPLG en figura):



Aplicaciones: Detección de caras

• Utilizan método **Integral Image Generation => scan** en dos dimensiones (filas y columnas)

  • Algunas implementaciones (poco eficientes) sobre GPUs en bibliografía basadas en filtros Haar



Primitiva scan paralela a nivel de warp (vector de entrada x de 32 elementos)

Ciclo Conferencia Grupo ArTeCS, UCM

Madrid, 22 Febrero 2018

Diseño eficiente de algoritmos Parallel Prefix sobre GPUs

GRACIAS!!

COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES

UNIVERSIDADE DA CORUÑA