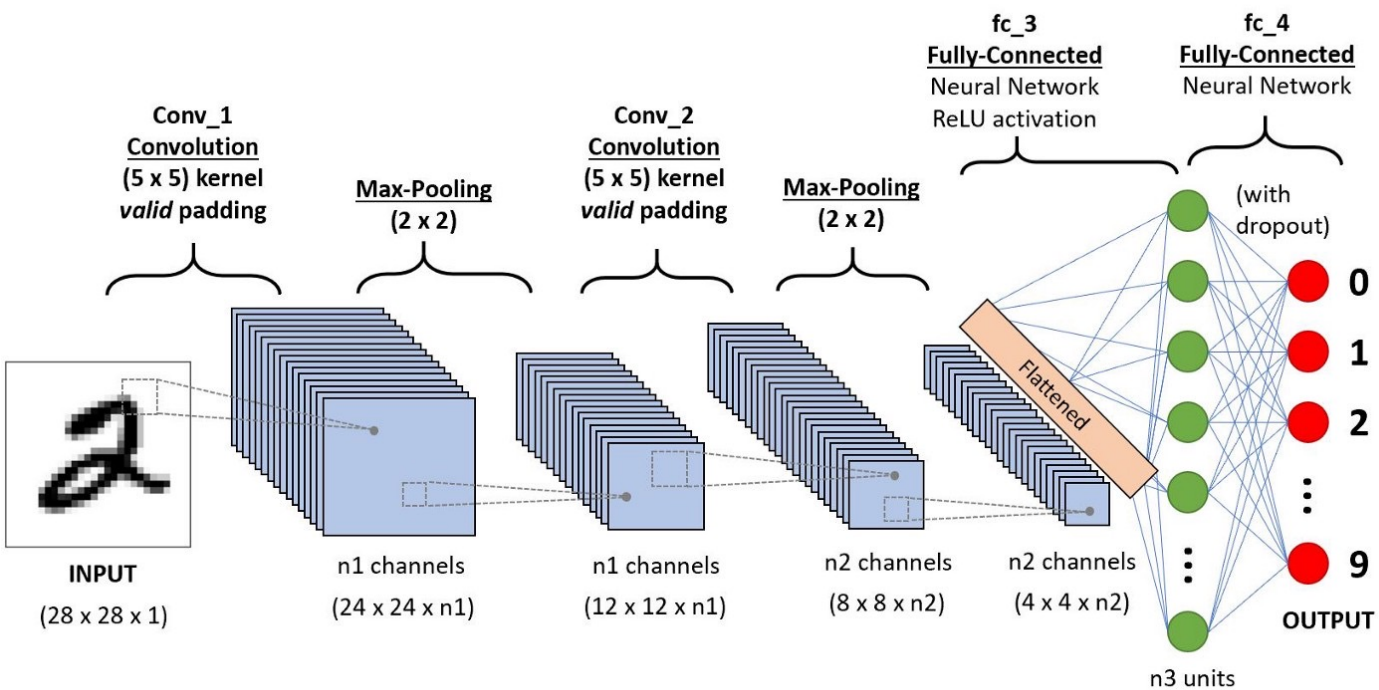
Una Visión Computacional de las Redes Neuronales

Enrique S. Quintana Ortí

Introducción a DL

Deep learning (DL): Aprendizaje automático mediante redes neuronales profundas (RNPs)

RNP: Algoritmo genérico que aprende automáticamente (es decir, se auto-adapta) para resolver un problema específico



Introducción a DL

Deep learning (DL): Aprendizaje automático mediante redes neuronales profundas (RNPs)

RNP: Algoritmo genérico que aprende automáticamente (es decir, se auto-adapta) para resolver un problema específico

Introducción a DL

¿Cómo aprende?

Entrenamiento supervisado: observar/ajustar

Dadas I y salidas asociadas correctas O (conocidas)

$$\rightarrow O^* = f(W, I)$$

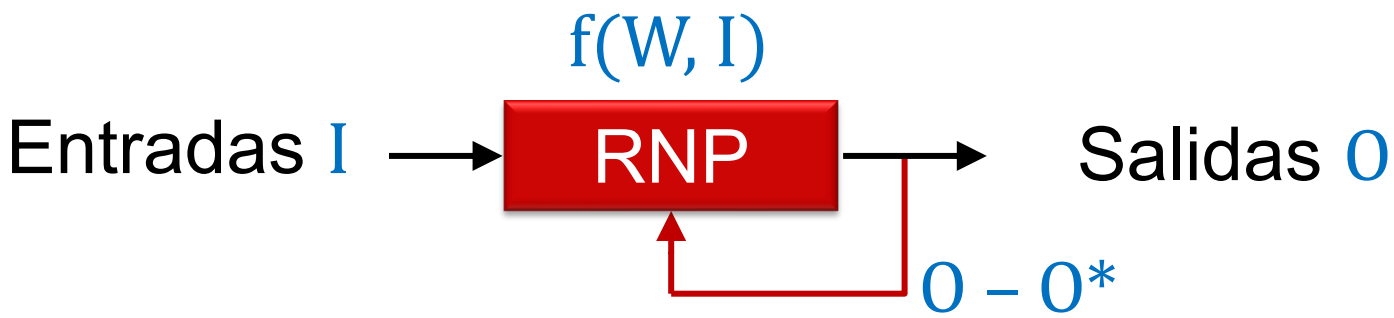
Calcular (ajustar) W para minimizar el error $O - O^*$

Introducción a DL

¿Cómo aprende?

Introducción a DL

¿Cómo aprende?



For counter $= 1$ upto Niter
- Choose an integer k uniformly at random from $\{1, 2, 3, \ldots, N\}$
- $x^{\{k\}}$ is current training data point
- $a^{[1]} = x^{\{k\}}$
- For $l = 2$ upto L
  - $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
  - $a^{[l]} = \sigma\left(z^{[l]}\right)$
  - $D^{[l]} = \mathrm{diag}\left(\sigma'(z^{[l]})\right)$
- end
- $\delta^{[L]} = D^{[L]}\left(a^{[L]} - y(x^{\{k\}})\right)$
- For $l = L - 1$ downto 2
  - $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$
- end
- For $l = L$ downto 2
  - $W^{[l]} \to W^{[l]} - \eta\, \delta^{[l]} a^{[l-1]^T}$
  - $b^{[l]} \to b^{[l]} - \eta\, \delta^{[l]}$
- end

end

- Procedimiento iterativo de optimización
- Fundamentos matemáticos (propagación de “ajustes” mediante derivación y la regla de la cadena)

Introducción a DL

Una vez ha aprendido…

31 de diciembre (1 de enero, 2 de enero…)

Introducción a DL

Una vez ha aprendido…

Inferencia: poner la RNP a trabajar (despliegue)

$$O^* = f(W, I) \quad (\approx O \text{ tras el entrenamiento})$$

Dados $I, W \rightarrow$ calcular O^*

Introducción a DL

Una vez ha aprendido…

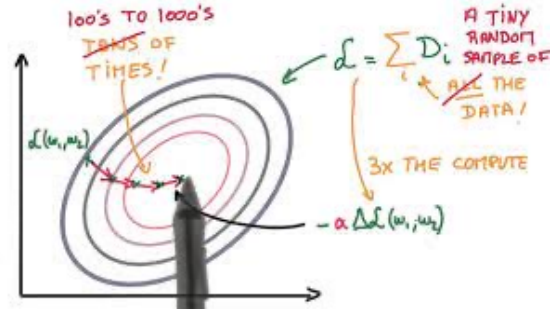
For counter = 1 upto Niter
  Choose an integer k uniformly at random from $\{1, 2, 3, \ldots, N\}$
  $x^{\{k\}}$ is current training data point
  $a^{[1]} = x^{\{k\}}$
  For $l = 2$ upto L
    $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
    $a^{[l]} = \sigma\left(z^{[l]}\right)$
    $D^{[l]} = \mathrm{diag}\left(\sigma'(z^{[l]})\right)$
  end
  $\delta^{[L]} = D^{[L]}\left(a^{[L]} - y(x^{\{k\}})\right)$
  For $l = L - 1$ downto 2
    $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$
  end
  For $l = L$ downto 2
    $W^{[l]} \to W^{[l]} - \eta\, \delta^{[l]} a^{[l-1]^T}$
    $b^{[l]} \to b^{[l]} - \eta\, \delta^{[l]}$
  end
end

- Procedimiento directo

Introducción a DL

¿Por qué ahora (y no antes)?

1. Cantidades ingentes de datos
2. “Nuevos” métodos
3. Arquitecturas paralelas potentes
4. Entornos (paquetes software)

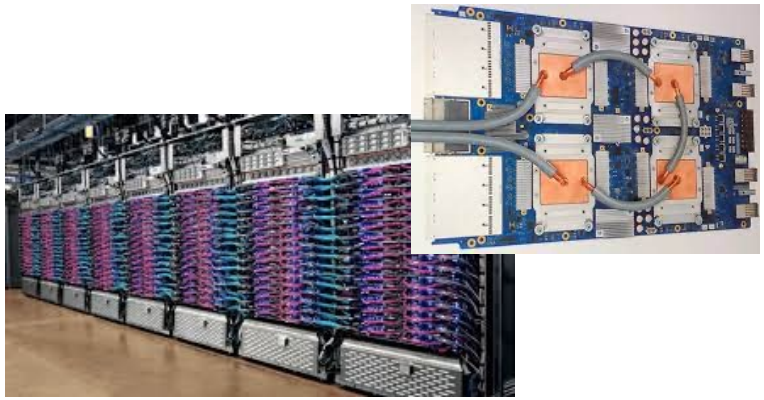


PyTorch

TensorRT

TensorFlow

Keras



Introducción a DL

El algoritmo se adapta al problema de manera (semi-)automática

- SGD (Stochastic Gradient Descent) o alguna variante

- Muy costoso…

For counter $= 1$ upto Niter
    Choose an integer k uniformly at random from $\{1, 2, 3, \ldots, N\}$
    $x^{\{k\}}$ is current training data point
    $a^{[1]} = x^{\{k\}}$
    For $l = 2$ upto L
        $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
        $a^{[l]} = \sigma\left(z^{[l]}\right)$
        $D^{[l]} = \mathrm{diag}\left(\sigma'(z^{[l]})\right)$
    end
    $\delta^{[L]} = D^{[L]}\left(a^{[L]} - y(x^{\{k\}})\right)$
    For $l = L - 1$ downto 2
        $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$
    end
    For $l = L$ downto 2
        $W^{[l]} \to W^{[l]} - \eta\, \delta^{[l]} {a^{[l-1]}}^T$
        $b^{[l]} \to b^{[l]} - \eta\, \delta^{[l]}$
    end
end

Introducción a DL

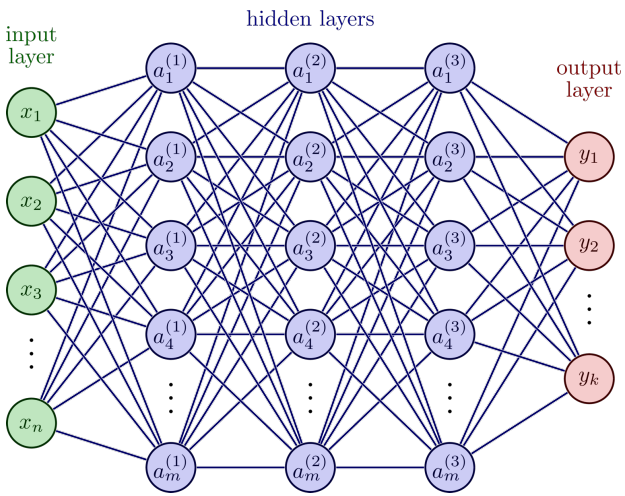
AlexNet	GPT-3	GPT-4
• 2 millones de parámetros • 27E+6 FLOPS • MNIST: 10000 B • 1-2 GPUs: 1-2 horas	• 175 “billones” de parámetros • 3,14E+23 FLOPS • Dataset: 45 TB • 1.024 NVIDIA A100 GPUs: 34 días	• 1.8 ”trillones” de parámetros • 2,15E+25 FLOPS • Dataset: 1 PB • 25.000 NVIDIA A100 GPUs: 90-100 días

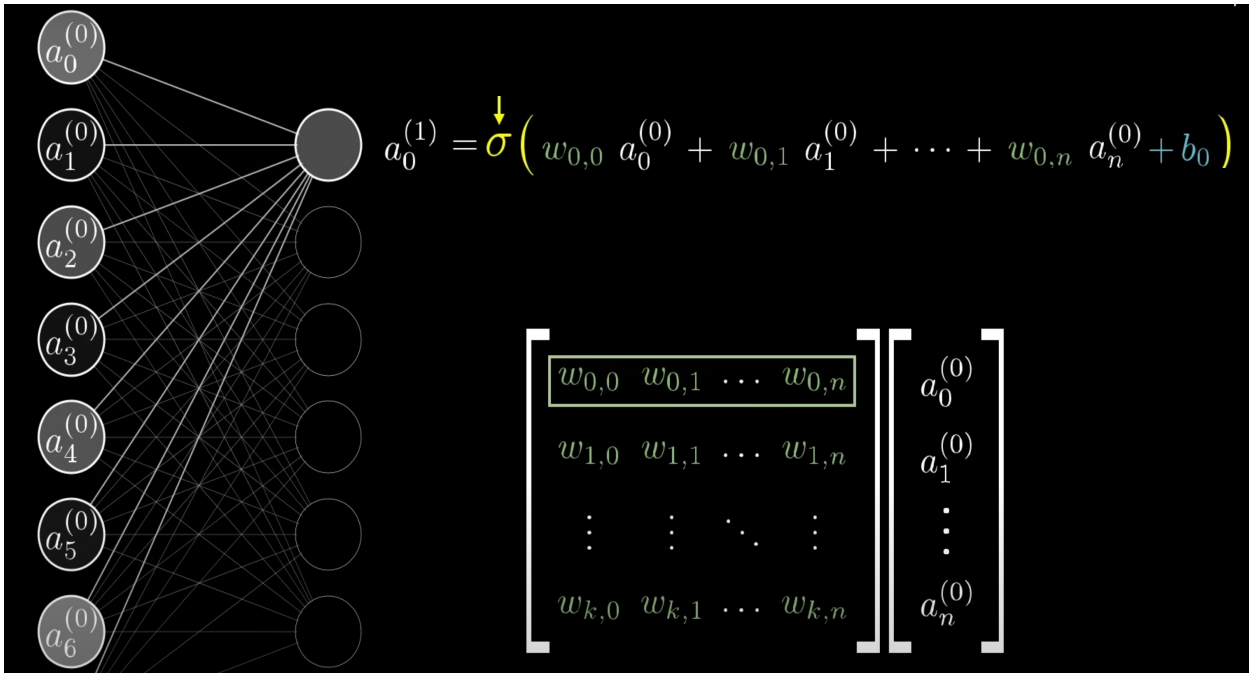
https://carboncredits.com/

Emitter	Equivalent number to training GPT-3 once
Plane Ride	345 flights across the US
Car	40 cars driven for one year
Person	13 American’s annual emissions or 50 non-American’s annual emissions

¿Qué hay tras una RNP?

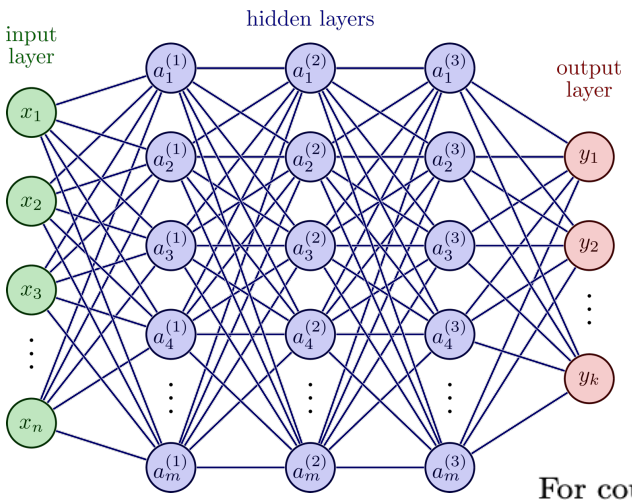
Perceptrón Multicapa: Fully-Connected (FC)





¿Qué hay tras una RNP?

Perceptrón Multicapa: Fully-Connected (FC)



For counter = 1 upto Niter

Choose an integer k uniformly at random from $\{1, 2, 3, \ldots, N\}$

$x^{\{k\}}$ is current training data point

$a^{[1]} = x^{\{k\}}$

For $l = 2$ upto L

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

$a^{[l]} = \sigma\left(z^{[l]}\right)$

$D^{[l]} = \text{diag}\left(\sigma'(z^{[l]})\right)$

end

$\delta^{[L]} = D^{[L]}\left(a^{[L]} - y(x^{\{k\}})\right)$

For $l = L - 1$ downto 2

$\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$

end

For $l = L$ downto 2

$W^{[l]} \rightarrow W^{[l]} - \eta\, \delta^{[l]} a^{[l-1]^T}$

$b^{[l]} \rightarrow b^{[l]} - \eta\, \delta^{[l]}$

end

end

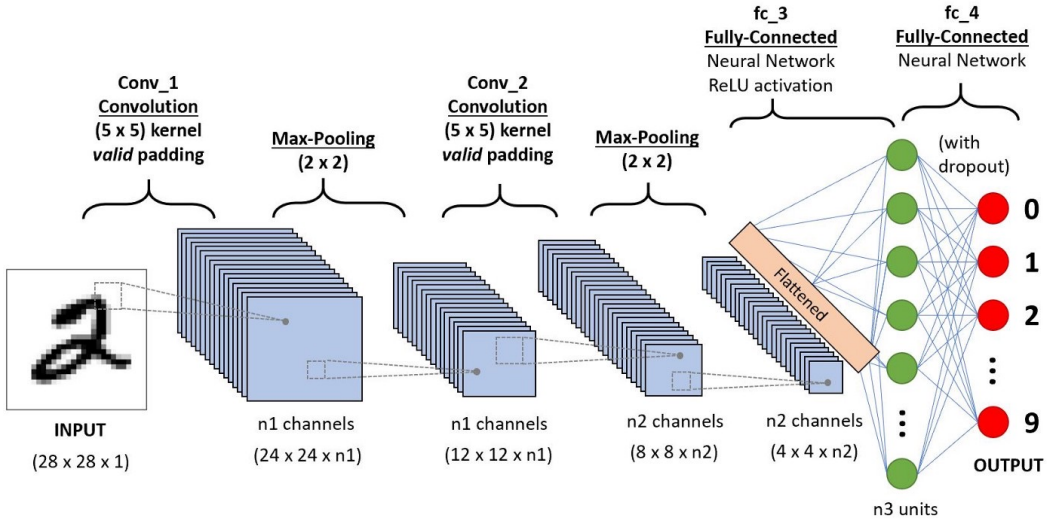
$A^{(l)} = W\, A^{(l-1)}$

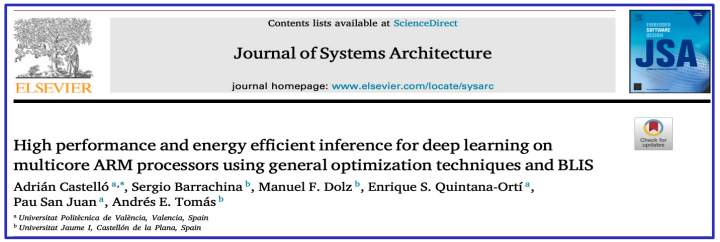
$D^{(l)} = (W^{(l+1)})^T\, A^{(l+1)}$

$W^{(l)} = W^{(l)} - D^{(l)}\, (A^{(l-1)})^T$

¿Qué hay tras una RNP?

- RNP convolucionales, utilizadas en visión por computador y procesamiento de señal
- Capas de regularización, softmax, dropout, pooling, FC y convolucionales
- Alrededor del 90% del tiempo en capas convolucionales y fully-connected (FC) para ResNet-50 v1.5




Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

High performance and energy efficient inference for deep learning on multicore ARM processors using general optimization techniques and BLIS

Adrián Castelló [a,*], Sergio Barrachina [b], Manuel F. Dolz [b], Enrique S. Quintana-Ortí [a], Pau San Juan [a], Andrés E. Tomás [b]

[a] Universitat Politècnica de València, Valencia, Spain
[b] Universitat Jaume I, Castellón de la Plana, Spain

¿Qué hay tras una RNP?

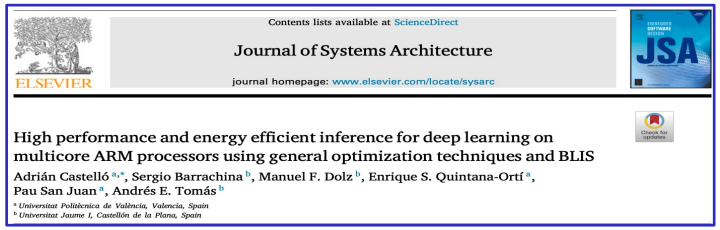
Convolución directa

```
1  void ConvDirect_Blocked( I[N][H_i][W_i][C_i],
2                           F[H_f][W_f][C_i][C_o],
3                           O[N][H_o][W_o][C_o] )
4  {
5    for ( h=0; h < N; h++ )
6     for ( j'=0; j' < C_o/C_o,b; j'++ )
7      for ( i'=0; i' < C_i/C_i,b; i'++ )
8       for ( l=0; l < H_o; l++ )
9        for ( k'=0; k' < W_o/W_o,b; k'++ )
10        for ( n=0; n < H_f; n++ )
11         for ( m=0; m < W_f; m++ )
12          // Micro-kernel for C+=A*B
13          for ( ii=0; ii < C_i,b; ii++ )
14           for ( kk=0; kk < W_o,b; kk++ )
15            for ( jj=0; jj < C_o,b; jj++ )
16             O[h][l][k'·W_o,b + kk][j'·C_o,b + jj]
17                  += I[h][l+n][k'·W_o,b + kk + m][i'·C_i,b + ii]
18                  *  F[n][m][j'·C_o,b + jj][i'·C_i,b + ii]
19 }
```

c — vi gemm_base.c — 43×12

~/c — vi gemm_base.c

```
void gemm_base( int m, int n, int k,
                float *A, int ldA,
                float *B, int ldB,
                float *C, int ldC ){

  for ( int j=0; j<n; j++ )
    for ( int i=0; i<m; i++ )
      for ( int p=0; p<k; p++ )
        C[j*Clda+i] += A[p*Alda+i]
                     * B[j*Blda+p];
}
```


Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

ELSEVIER

JSA

High performance and energy efficient inference for deep learning on multicore ARM processors using general optimization techniques and BLIS

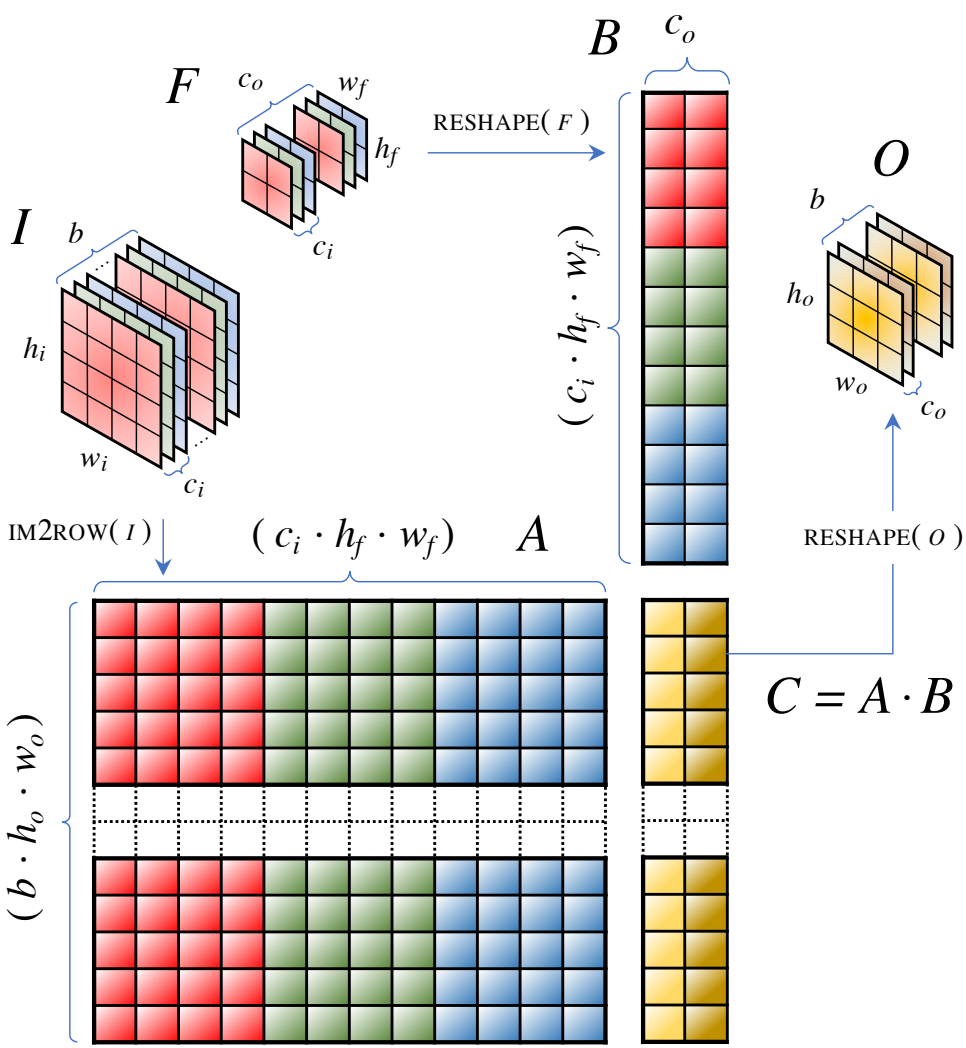
Adrián Castelló [a,*], Sergio Barrachina [b], Manuel F. Dolz [b], Enrique S. Quintana-Ortí [a], Pau San Juan [a], Andrés E. Tomás [b]

[a] Universitat Politècnica de València, Valencia, Spain

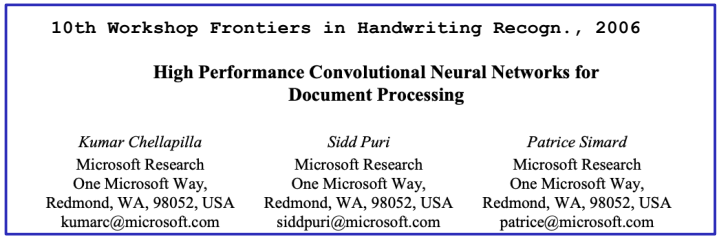
[b] Universitat Jaume I, Castellón de la Plana, Spain

¿Qué hay tras una RNP?

Convolución vía lowering




```
1  void Im2Row( A[bh_o w_o][c_i h_f w_f], I[b][h_i][w_i][c_i] ) {
2    for ( h = 0; h < b; h++ )
3      for ( i = 0; i < c_i; i++ )
4        for ( k = 0; k < w_o; k++ )
5          for ( l = 0; l < h_o; l++ ) {
6            r = l + kh_i + hw_i h_i;
7            for ( m = 0; m < w_f; m++ )
8              for ( n = 0; n < h_f; n++ ) {
9                c = n + mh_f + iw_f h_f;
10               A[r][c] = I[h][l + n][k + m][i];
11 }}}
```


10th Workshop Frontiers in Handwriting Recogn., 2006

High Performance Convolutional Neural Networks for Document Processing

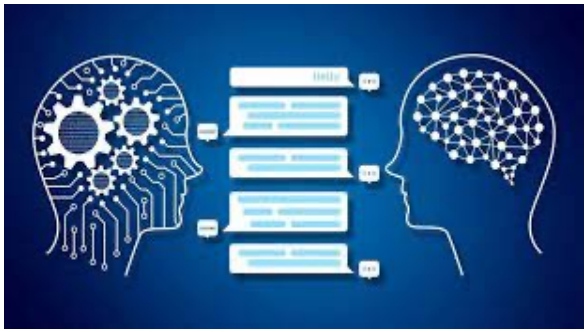
Kumar Chellapilla — Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA, kumarc@microsoft.com

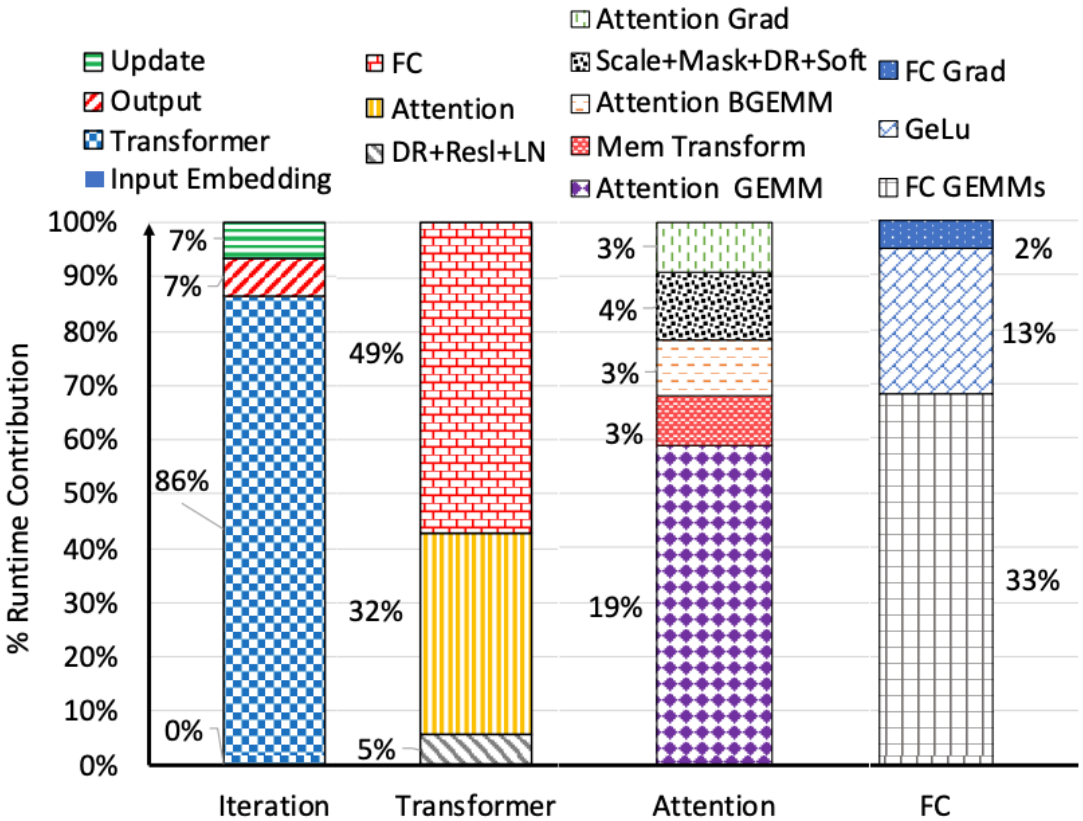
Sidd Puri — Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA, siddpuri@microsoft.com

Patrice Simard — Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA, patrice@microsoft.com

¿Qué hay tras una RNP?

- Transformers dominan en tareas de procesamiento del lenguaje natural y la IA generativa
- GEMM consume sobre el 55% del tiempo de ejecución para BERT-Large





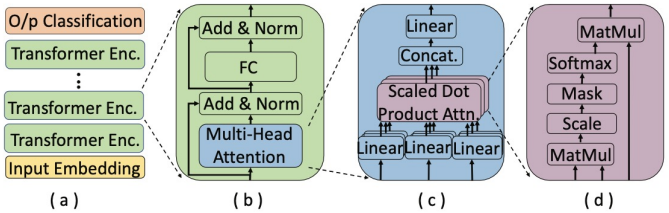


Figure 2: BERT hierarchical model breakdown.

2022 IEEE International Symposium on Workload Characterization (IISWC)

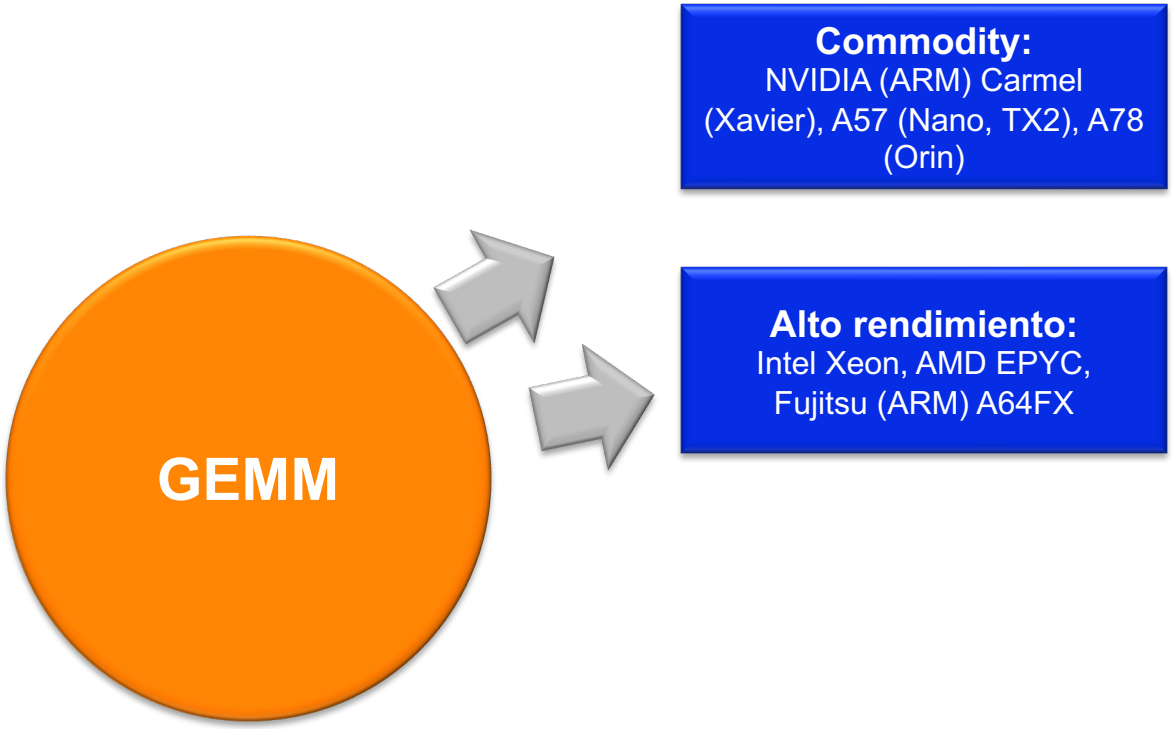
Demystifying BERT: System Design Implications

Suchita Pati[1,2] Shaizeen Aga[2] Nuwan Jayasena[2] Matthew D. Sinclair [1,2]

[1] *University of Wisconsin-Madison*
{spati, sinclair}@cs.wisc.edu
[2] *Advanced Micro Devices Inc.*
{shaizeen.aga, nuwan.jayasena}@amd.com

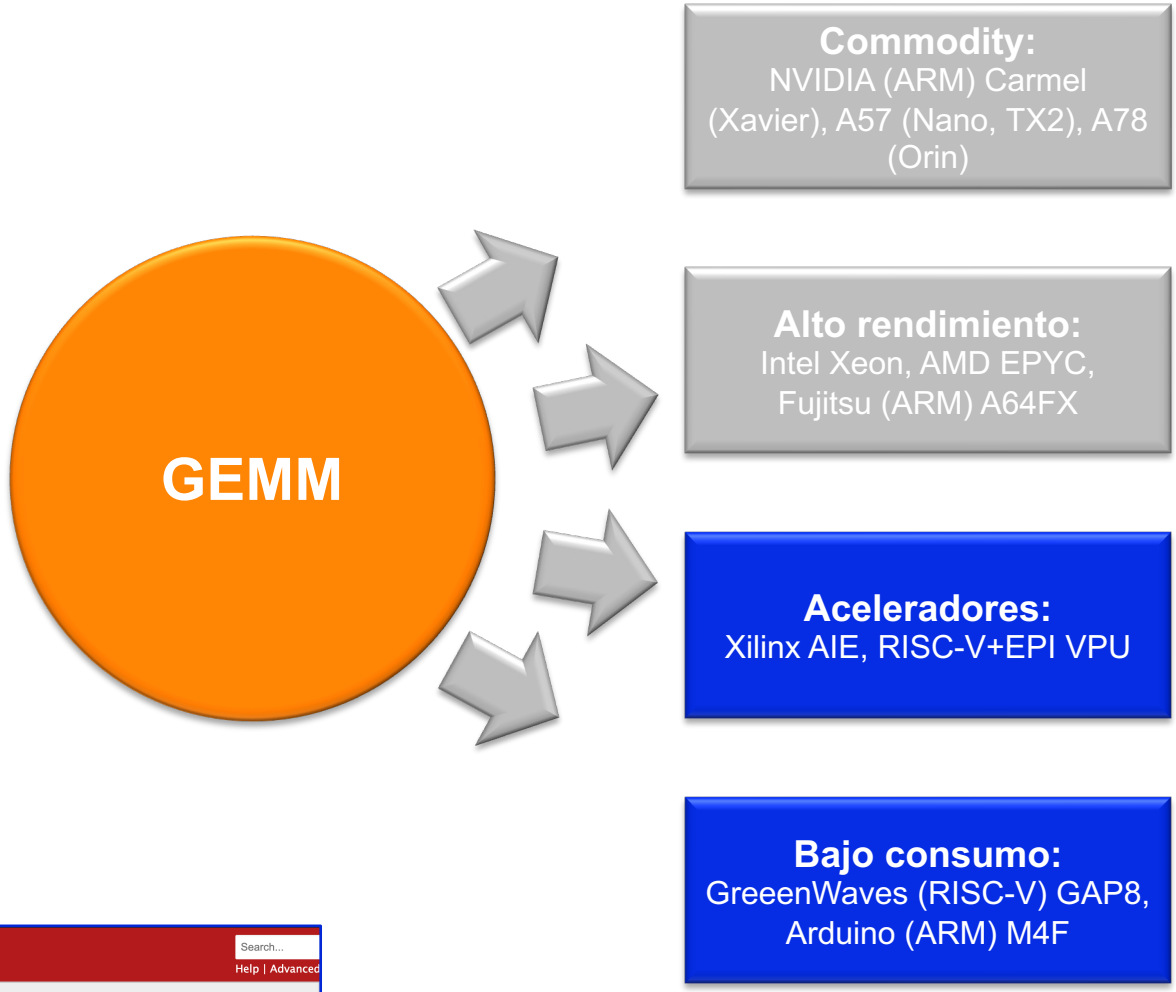
¿Qué hay tras una RNP?

GEMM en arquitecturas convencionales, optimizada para computación científica



¿Qué hay tras una RNP?

Deep learning y nuevas arquitecturas

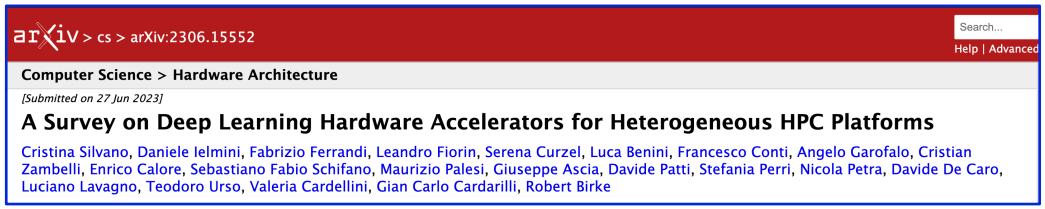


DOI:10.1145/3282307

Innovations like domain-specific hardware, enhanced security, open instruction sets, and agile chip development will lead the way.

BY JOHN L. HENNESSY AND DAVID A. PATTERSON

A New Golden Age for Computer Architecture


arXiv > cs > arXiv:2306.15552

Search...

Help | Advanced

Computer Science > Hardware Architecture

[Submitted on 27 Jun 2023]

A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms

Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, Sebastiano Fabio Schifano, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Stefania Perri, Nicola Petra, Davide De Caro, Luciano Lavagno, Teodoro Urso, Valeria Cardellini, Gian Carlo Cardarilli, Robert Birke

Índice

- Introducción
- Optimización de GEMM
- GEMM para DL
- Generación automática
- Portabilidad

Optimización de GEMM

GotoBLAS2

ACM Trans. Math. Software, 2008

Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

2020 SIAM Activity Group on Supercomputing Best Paper Prize

- Open source: BLIS*, OpenBLAS
- Vendedores: ARMPL, AMD AOCL-BLAS, posiblemente también Intel MKL (oneAPI)

1. Jerarquía del sistema de cache
2. Unidades aritméticas SIMD
3. (Procesadores multinúcleo con niveles de caché privados/compartidos)

*2023 J. H. Wilkinson Prize for Numerical Software

Optimización de GEMM

GotoBLAS2

ACM Trans. Math. Software, 2008

Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

$$C \mathrel{+}= A * B$$
$$C \rightarrow m \; x \; n,$$
$$A \rightarrow m \; x \; k, \quad B \rightarrow k \; x \; n$$

```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){    // L2


  for (ic=0; ic<m; ic+=mc){   // L3


   for (jr=0; jr<nc; jr+=nr)  // L4
     for(ir=0; ir<mc; ir+=mr) // L5




  }}
```

- 5 bucles

Optimización de GEMM

GotoBLAS2

ACM Trans. Math. Software, 2008

Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

$$C \mathrel{+}= A * B$$
$$C \rightarrow m \; x \; n,$$
$$A \rightarrow m \; x \; k, \quad B \rightarrow k \; x \; n$$

```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){     // L2
  // Pack B
  Bc:=B(pc:pc+kc-1, jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc){    // L3
   // Pack A
   Ac:=A(ic:ic+mc-1, pc:pc+kc-1);
   for (jr=0; jr<nc; jr+=nr)   // L4
     for(ir=0; ir<mc; ir+=mr) // L5




  }}
```

- 5 bucles
- 2 rutinas de empaquetamiento

Optimización de GEMM

GotoBLAS2

ACM Trans. Math. Software, 2008

Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

$$C \mathrel{+}= A * B$$
$$C \rightarrow m \; x \; n,$$
$$A \rightarrow m \; x \; k, \quad B \rightarrow k \; x \; n$$

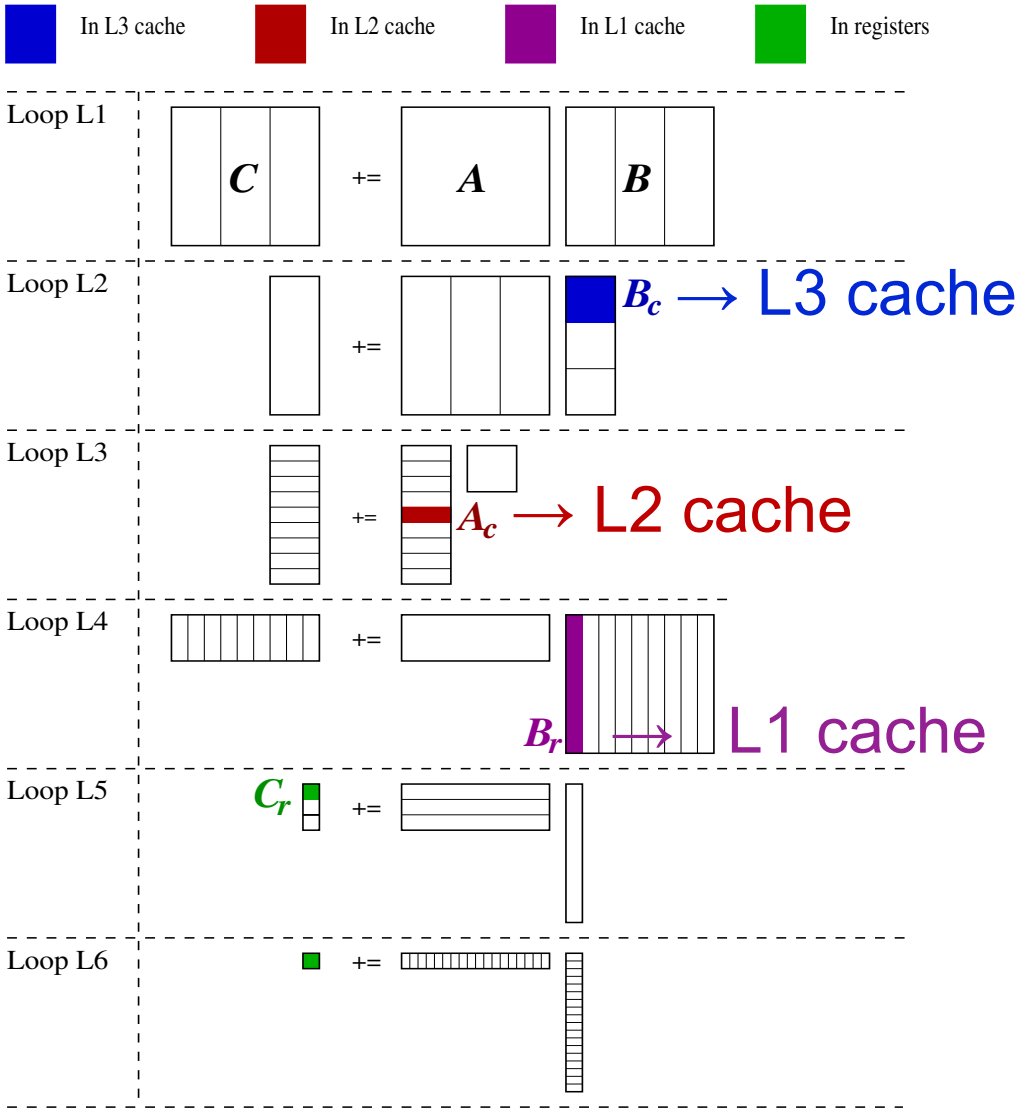
```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){    // L2
  // Pack B
  Bc:=B(pc:pc+kc-1, jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc){   // L3
   // Pack A
   Ac:=A(ic:ic+mc-1, pc:pc+kc-1);
   for (jr=0; jr<nc; jr+=nr)  // L4
     for(ir=0; ir<mc; ir+=mr) // L5
      // Micro-kernel
      C(ic+ir:ic+ir+mr-1,
        jc+jr:jc+jr+nr-1)
        += Ac(ir:ir+mr-1, 0:kc-1)
        *  Bc(0:kc-1, jr:jr+nr-1);
  }}
```

- 5 bucles
- 2 rutinas de empaquetamiento
- 1 micro-kernel (+1 bucle)

Optimización de GEMM

1) Tiling (blocking):

- Particionar $m, n, k \rightarrow mc, nc, kc$




```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){    // L2


  for (ic=0; ic<m; ic+=mc){   // L3




  }}
```

Optimización de GEMM

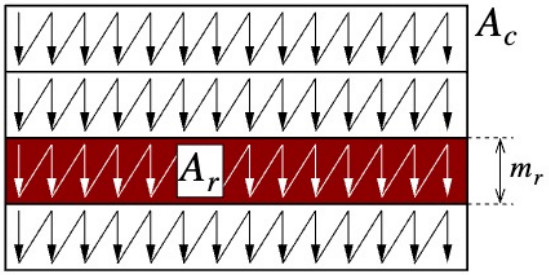
2) Empaquetado:

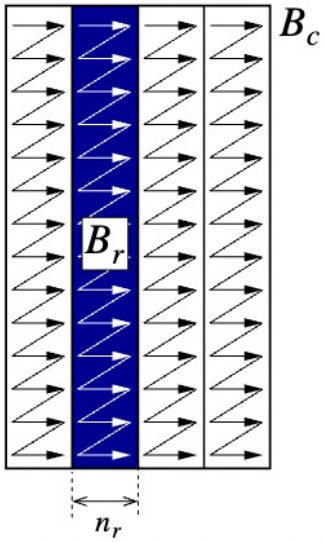
- Buffer $A_c \rightarrow mc\ x\ kc$, buffer $B_c \rightarrow kc\ x\ nc$

Accesos consecutivos en memoria desde

el micro-kernel

- Reducir fallos de cachés
- Cargas SIMD de datos en los buffers





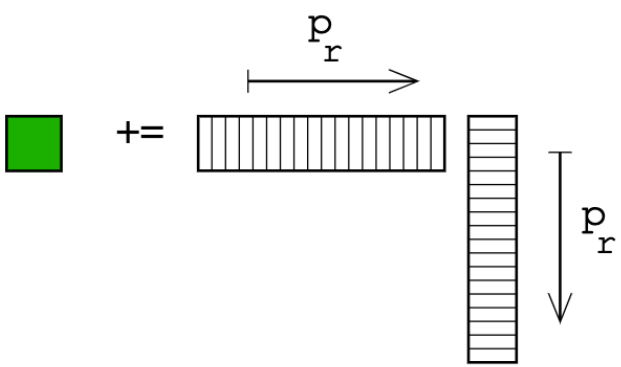

```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){    // L2
  // Pack B
  Bc:=B(pc:pc+kc-1, jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc){   // L3
   // Pack A
   Ac:=A(ic:ic+mc-1, pc:pc+kc-1);




  }}
```

Optimización de GEMM

3) Micro-kernel:

- $C \rightarrow mr\ x\ nr$ residente en registros vectoriales
- Cargas SIMD de C en registros vectoriales antes del bucle
- Dentro del bucle **L6**:
  - Aritmética SIMD
  - Cargas de A_c, B_r desde caché L2, L1, resp.
- Almacenamiento SIMD en C después del bucle

```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){    // L2
  // Pack B
  Bc:=B(pc:pc+kc-1, jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc){   // L3
   // Pack A
   Ac:=A(ic:ic+mc-1, pc:pc+kc-1);
   for (jr=0; jr<nc; jr+=nr)  // L4
     for(ir=0; ir<mc; ir+=mr) // L5
       // Micro-kernel
       C(ic+ir:ic+ir+mr-1,
         jc+jr:jc+jr+nr-1)
         += Ac(ir:ir+mr-1, 0:kc-1)
         *  Bc(0:kc-1, jr:jr+nr-1);
  }}
```




```
for (pr=0; pr<kc; pr++)  // Loop L6
  C(ic+ir:ic+ir+mr-1,
    jc+jr:jc+jr+nr-1)
    += Ac(ir:ir+mr-1,pr)
    *  Bc(pr,jr:jr+nr-1);
```

Micro-kernel (+1 loop)

Índice

- …
- Optimización de GEMM
- GEMM para DL
- Generación automática
- Portabilidad

GEMM para DL

Implementaciones de BLAS:

- Open source: BLIS, OpenBLAS
- Vendedor: ARMPL, AMD AOCL-BLAS, Intel MKL (oneAPI)?

Problemas:

- Consumo de memoria
- Funcionalidad no presente: INT8/INT16/INT32, BF16/TF32, precisión mixta
- Rendimiento subóptimo para DL

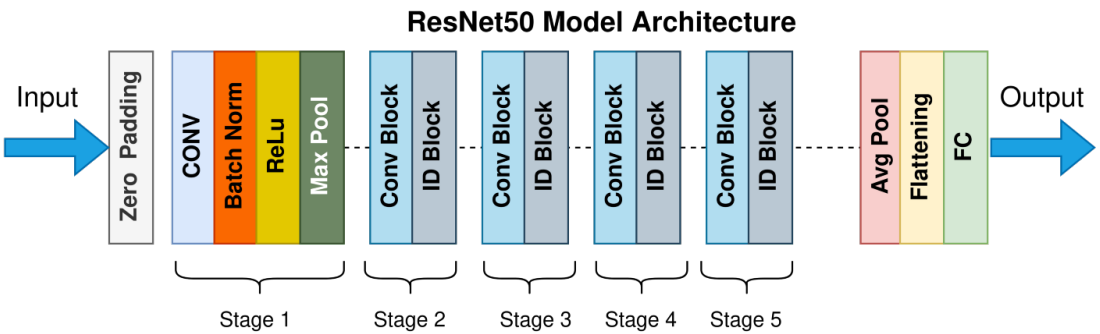
GEMM para DL

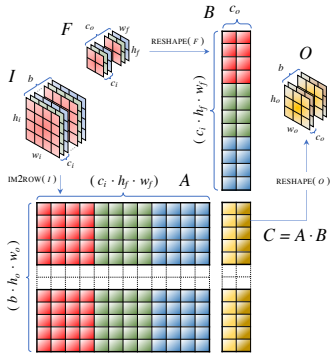
NVIDIA Carmel (ARM v8.2), 2.3 GHz, IEEE FP32

- GEMM, $m=n=k=2{,}000$
  - BLIS (v0.8.1): 28 GFLOPS (76% del pico)
  - ARMPL (v21.1): 26.5 GFLOPS
  - OpenBLAS (v0.3.19): 23 GFLOPS

GEMM para DL

Capas convolucionales en ResNet-50 v1.5, después de IM2COL





Layer id.	Layer numbers in ResNet50 v1.5	m	n	k
1	001	1,605,632	64	147
2	006	401,408	64	64
3	009/021/031	401,408	64	576
4	012/014/024/034	401,408	256	64
5	018/028	401,408	64	256
6	038	401,408	128	256
7	041/053/063/073	100,352	128	1,152
8	044/056/066/076	100,352	512	128
9	046	100,352	512	256
10	050/060/070	100,352	128	512

Layer id.	Layer numbers in ResNet50 v1.5	m	n	k
11	080	100,352	256	512
12	083/095/105/115/125/135	25,088	256	2,304
13	086/098/108/118/128/138	25,088	1,024	256
14	088	25,088	1,024	512
15	092/102/112/122/132	25,088	256	1,024
16	142	25,088	512	1,024
17	145/157/167	6,272	512	4,608
18	148/160/170	6,272	2,048	512
19	150	6,272	2,048	1,024
20	154/164	6,272	512	2,048

GEMM para DL

NVIDIA Carmel (ARM v8.2), 2.3 GHz, IEEE FP32

- GEMM, m= 401,408, n= 64, k= 147 (layer 006, ResNet-50 v1.5)
  - BLIS (v0.8.1): 17 GFLOPS (46% del pico)
  - ARMPL (v21.1): 13 GFLOPS
  - OpenBLAS (v0.3.19): 12 GFLOPS

¿Por qué?

GEMM para DL

Parámetros de GEMM que determinan el rendimiento:

- Parámetros de configuración de cache: mc, nc, kc
  - Adaptar tamaños de bloques de *A, B, C* a las capacidades de las cache, para reducir fallos
- Parámetros del micro-kernel: $mr\ x\ nr$
  - Maximizar el uso de registros vectoriales
  - Maximizar los flops por acceso a memoria

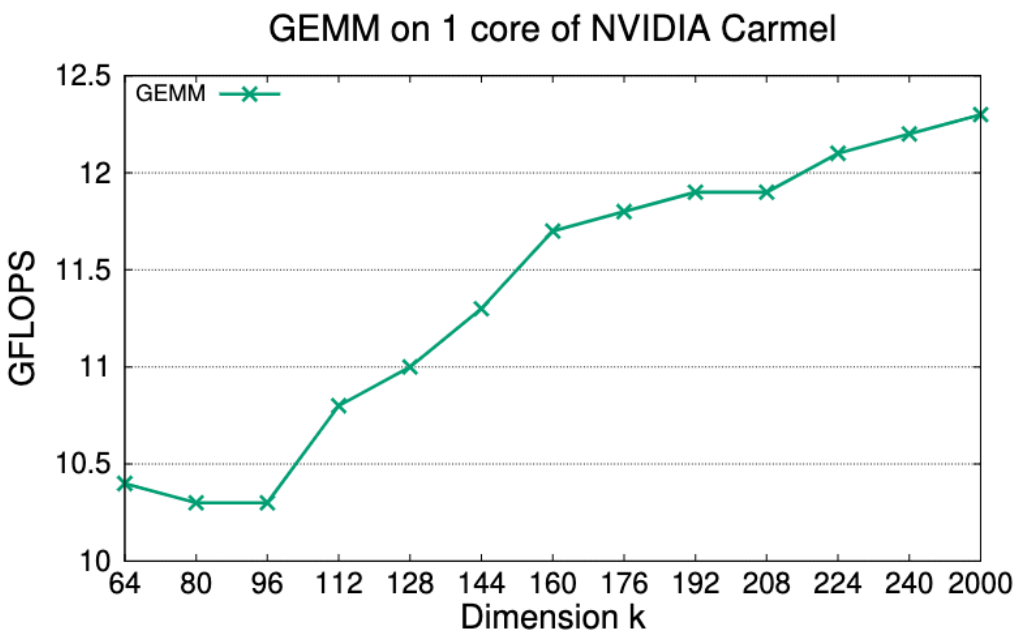
```
1  for (jc=0; jc<n; jc+=nc) // Loop L1
2   for (pc=0; pc<k; pc+=kc){    // L2
3    // Pack B
4    Bc:=B(pc:pc+kc-1, jc:jc+nc-1);
5    for (ic=0; ic<m; ic+=mc){   // L3
6     // Pack A
7     Ac:=A(ic:ic+mc-1, pc:pc+kc-1);
8     for (jr=0; jr<nc; jr+=nr)  // L4
9       for(ir=0; ir<mc; ir+=mr) // L5
10       // Micro-kernel
11       C(ic+ir:ic+ir+mr-1,
12         jc+jr:jc+jr+nr-1)
13         += Ac(ir:ir+mr-1, 0:kc-1)
14         *  Bc(0:kc-1, jr:jr+nr-1);
15   }}
```

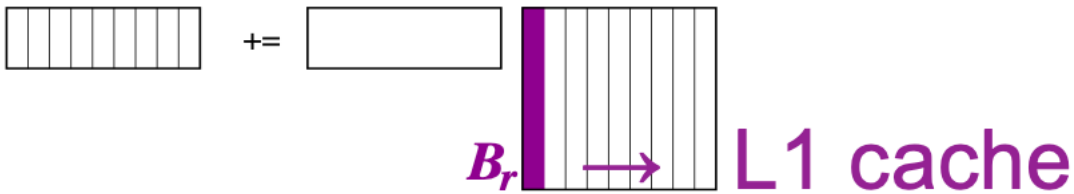
GEMM para DL

Implementaciones de BLAS utilizan valores fijos para los parámetros de configuración de caché: mc, nc, kc, pero $kc \leq k \rightarrow$

Micro-panel B_r, de dimensión $kc\ x\ nr$, en cache L1 resulta en una baja utilización de la cache L1

k	L1 KB	(%)
64	4.0	6.2
96	6.0	9.4
128	8.0	12.5
160	10.0	15.6
192	12.0	18.7
224	14.0	21.9
240	15.0	23.4
2000	15.0	23.4





GEMM para DL

Implementaciones de BLAS solo ofrecen un micro-kernel por arquitectura

Layer id.	m_c	n_c	k_c	m_r	n_r	L1 (%)	Max (%)
8	3,584	256	128	4	4	3.12	25
8	3,584	256	128	4	8	6.25	50
8	3,328	275	128	4	12	9.38	50
8	3,328	275	128	4	16	12.50	50
8	3,072	298	128	4	20	15.60	50
8	3,072	298	128	4	24	18.80	50
8	3,072	298	128	4	28	21.90	50
8	3,584	256	128	8	4	3.12	25
8	3,584	256	128	8	8	6.25	25
8	3,328	275	128	8	12	9.38	50
8	3,584	256	128	12	4	3.12	25
8	3,584	256	128	12	8	6.25	25
8	3,584	256	128	16	4	3.12	25
8	3,584	256	128	20	4	3.12	25
8	3,584	256	128	24	4	3.12	25
8	3,584	256	128	28	4	3.12	25

BEST

BLIS*

¿Por qué?

Micro-panel B_r, de dimensión $kc\ x\ nr$, en cache L1 → Mejor utilizar micro-kernels con nr más grande

* FP32

GEMM para DL

Mejorar la utilización de la cache L1: Variedad de micro-kernels

ARMv8a, 4x4

```
.LBB1_5:
  add  x11, x9, x10
  add  x12, x8, x10
  ldr  q4, [x11, #15952]
  ldr  q5, [x12, #15968]
  ldr  q6, [x11, #15968]
  ldr  q7, [x12, #15984]
  ldr  q16, [x11, #15984]
  ldr  q17, [x12, #16000]
  fmla v3.4s, v5.4s, v4.s[0]
  fmla v2.4s, v5.4s, v4.s[1]
  fmla v1.4s, v5.4s, v4.s[2]
  fmla v0.4s, v5.4s, v4.s[3]
  ldr  q4, [x11, #16000]
  ldr  q5, [x12, #16016]
  fmla v3.4s, v7.4s, v6.s[0]
  fmla v2.4s, v7.4s, v6.s[1]
  fmla v1.4s, v7.4s, v6.s[2]
  fmla v0.4s, v7.4s, v6.s[3]
  fmla v3.4s, v17.4s, v16.s[0]
  fmla v2.4s, v17.4s, v16.s[1]
  fmla v1.4s, v17.4s, v16.s[2]
  fmla v0.4s, v17.4s, v16.s[3]
  adds x10, x10, #64
  fmla v3.4s, v5.4s, v4.s[0]
  fmla v2.4s, v5.4s, v4.s[1]
  fmla v1.4s, v5.4s, v4.s[2]
  fmla v0.4s, v5.4s, v4.s[3]
  b.ne .LBB1_5
```

x86 AVX512, 4x16

```
.LBB5_12:
  vmovaps       -16(%rdx,%r13), %xmm4
  vmovaps       (%rdx,%r13), %xmm5
  vmovups       -64(%rsi,%r13,4), %zmm6
  vmovups       (%rsi,%r13,4), %zmm7
  vbroadcastss  %xmm4, %zmm8
  vfmadd213ps   %zmm3, %zmm6, %zmm8
  vmovshdup     %xmm4, %xmm3
  vbroadcastsd  %xmm3, %zmm9
  vfmadd213ps   %zmm2, %zmm6, %zmm9
  vpermilps     $234, %xmm4, %xmm2
  vbroadcastsd  %xmm2, %zmm10
  vfmadd213ps   %zmm1, %zmm6, %zmm10
  vpermilps     $239, %xmm4, %xmm1
  vbroadcastsd  %xmm1, %zmm4
  vfmadd213ps   %zmm0, %zmm6, %zmm4
  vbroadcastss  %xmm5, %zmm3
  vfmadd213ps   %zmm8, %zmm7, %zmm3
  vmovshdup     %xmm5, %xmm0
  vbroadcastsd  %xmm0, %zmm2
  vfmadd213ps   %zmm9, %zmm7, %zmm2
  vpermilps     $234, %xmm5, %xmm0
  vbroadcastsd  %xmm0, %zmm1
  vfmadd213ps   %zmm10, %zmm7, %zmm1
  vpermilps     $239, %xmm5, %xmm0
  vbroadcastsd  %xmm0, %zmm0
  vfmadd213ps   %zmm4, %zmm7, %zmm0
  addq          $32, %r13
  cmpq          $2048, %r13
  jne           .LBB5_12
```

GEMM para DL

¿Solución?

ARMv8a, 4x4

```
.LBB1_5:
  add   x11, x9, x10
  add   x12, x8, x10
  ldr   q4, [x11, #15952]
  ldr   q5, [x12, #15968]
  ldr   q6, [x11, #15968]
  ldr   q7, [x12, #15984]
  ldr   q16, [x11, #15984]
  ldr   q17, [x12, #16000]
  fmla v3.4s, v7.4s, v6.s[0]
  fmla v2.4s, v7.4s, v6.s[1]
  fmla v1.4s, v7.4s, v6.s[2]
  fmla v0.4s, v7.4s, v6.s[3]
  fmla v3.4s, v17.4s, v16.s[0]
  fmla v2.4s, v17.4s, v16.s[1]
  fmla v1.4s, v17.4s, v16.s[2]
  fmla v0.4s, v17.4s, v16.s[3]
  adds x10, x10, #64
  fmla v3.4s, v5.4s, v4.s[0]
  fmla v2.4s, v5.4s, v4.s[1]
  fmla v1.4s, v5.4s, v4.s[2]
  fmla v0.4s, v5.4s, v4.s[3]
  b.ne .LBB1_5
```

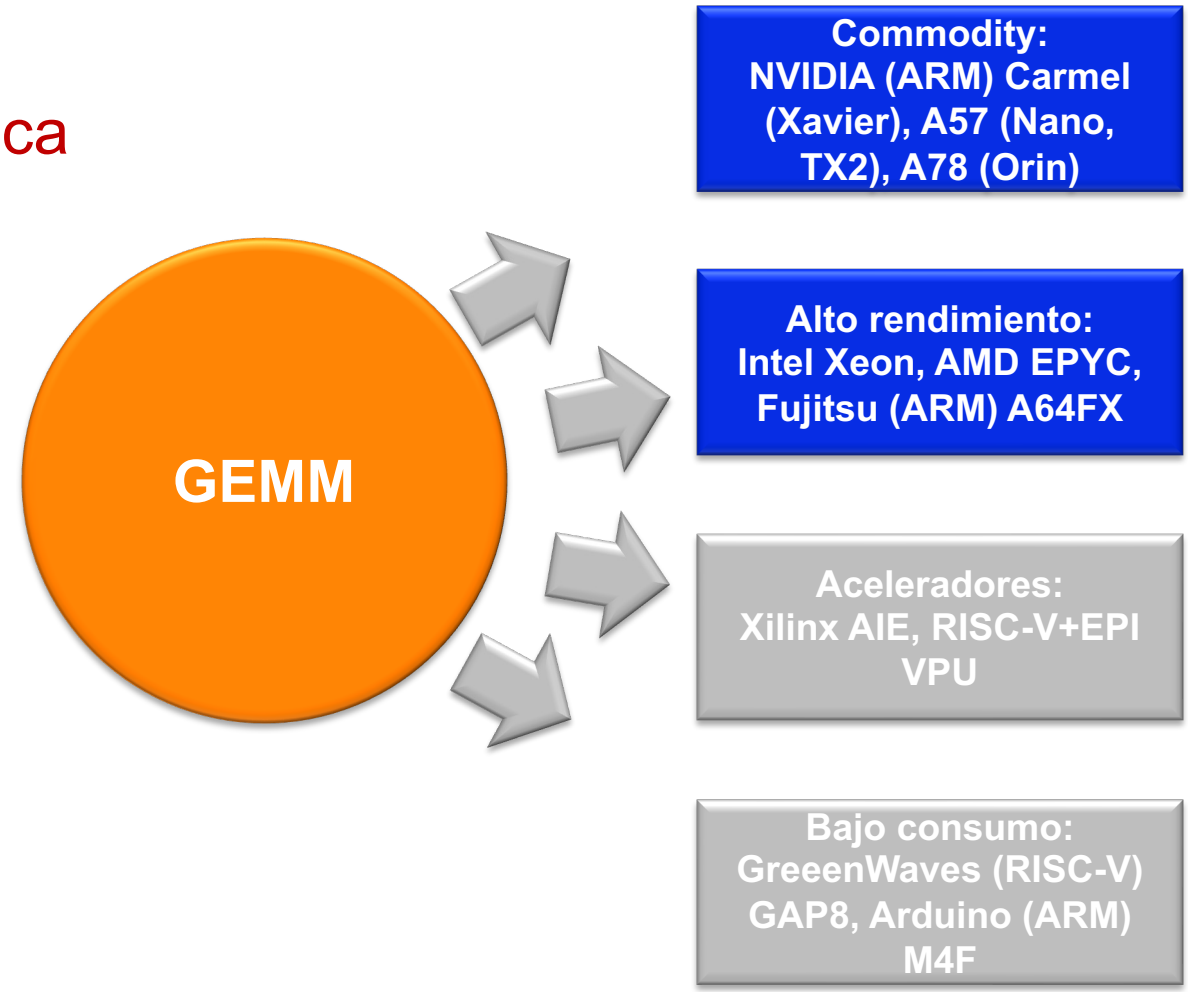
x86 AVX512, 4x16

```
.LBB5_12:
  vmovaps        -16(%rdx,%r13), %xmm4
  vmovaps        (%rdx,%r13), %xmm5
  vmovups        -64(%rsi,%r13,4), %zmm6
  vmovups        (%rsi,%r13,4), %zmm7
  vbroadcastss   %xmm4, %zmm8
  vfmadd213ps    %zmm3, %zmm6, %zmm8
  vmovshdup      %xmm4, %xmm3
  vbroadcastsd   %xmm3, %zmm9
  vfmadd213ps    %zmm0, %zmm6, %zmm4
  vbroadcastss   %xmm5, %zmm3
  vfmadd213ps    %zmm8, %zmm7, %zmm3
  vmovshdup      %xmm5, %xmm0
  vbroadcastsd   %xmm0, %zmm2
  vfmadd213ps    %zmm9, %zmm7, %zmm2
  vpermilps      $234, %xmm5, %xmm0
  vbroadcastsd   %xmm0, %zmm1
  vfmadd213ps    %zmm10, %zmm7, %zmm1
  vpermilps      $239, %xmm5, %xmm0
  vbroadcastsd   %xmm0, %zmm0
  vfmadd213ps    %zmm4, %zmm7, %zmm0
  addq           $32, %r13
  cmpq           $2048, %r13
  jne            .LBB5_12
```

¡Solo para expertos!

Índice

- …
- GEMM para DL
- Generación automática
- Portabilidad



Generación automática de Micro-kernels / GEMM

ChatGPT

EN write me a micro-kernel for matrix multiplication for ARM NEON

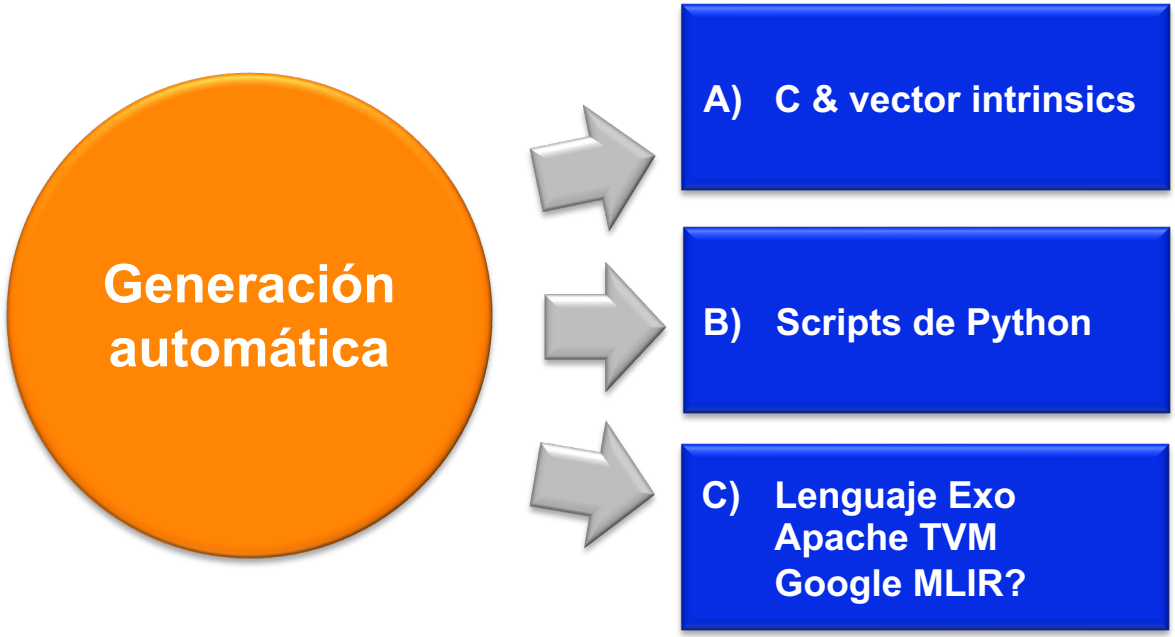
Writing a complete ARM NEON micro-kernel for matrix multiplication here might be a bit too extensive, but I can certainly provide you with a simplified version of a micro-kernel to get you started. This micro-kernel will perform matrix multiplication for a 4×4 block using ARM NEON intrinsics.

```
#include <arm_neon.h>

// Matrix multiplication micro-kernel for a 4×4 block
void matrix_multiply_neon(const float *A, const float *B, float *C, int stride)
    float32×4_t c0 = vdupq_n_f32(0.0f);
    float32×4_t c1 = vdupq_n_f32(0.0f);
    float32×4_t c2 = vdupq_n_f32(0.0f);
    float32×4_t c3 = vdupq_n_f32(0.0f);

    for (int k = 0; k < 4; ++k) {
        float32×4_t b = vld1q_f32(B + k * 4);
        float32×4_t a0 = vdupq_n_f32(A[k * stride]);
        float32×4_t a1 = vdupq_n_f32(A[k * stride + 1]);
        float32×4_t a2 = vdupq_n_f32(A[k * stride + 2]);
        float32×4_t a3 = vdupq_n_f32(A[k * stride + 3]);

        c0 = vmlaq_f32(c0, a0, b);
        c1 = vmlaq_f32(c1, a1, b);
        c2 = vmlaq_f32(c2, a2, b);
        c3 = vmlaq_f32(c3, a3, b);
    }

    vst1q_f32(C, c0);
    vst1q_f32(C + stride, c1);
    vst1q_f32(C + 2 * stride, c2);
    vst1q_f32(C + 3 * stride, c3);
}
```

Reger

Generación automática de Micro-kernels / GEMM

Necesitamos micro-kernels de diferentes tamaños, para diferentes arquitecturas y dimensiones de problemas, pero ¡el desarrollo manual es una tarea solo para expertos!



Proc. 43rd ACM SIGPLAN PLDI, 2022

Exocompilation for Productive Programming of Hardware Accelerators

Yuka Ikarashi* MIT CSAIL, USA · Gilbert Louis Bernstein* UC Berkeley, USA · Alex Rein UC Berkeley · Hasan Genc UC Berkeley, USA · Jonathan Ragan-Kelley MIT CSAIL, USA

arXiv > cs > arXiv:1802.04799

Computer Science > Machine Learning

[Submitted on 12 Feb 2018 (v1), last revised 5 Oct 2018 (this version, v3)]

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

Generación automática de Micro-kernels

A) C & vector instrinsics*:

+ Independente de las dimensiones del micro-kernel

+ Independiente (casi) del tipo de datos

```
1  #define mv     (mr/vl_fp32)
2  #define nv     (nr/vl_fp32)
3
4  gemm_ukernel_Cresident_SIMD_mrxnr( int kc, float *Ar, float *Br,
5                                     float *C,  int Clda ) {
6  // mr x nr micro-kernel with C resident in regs.
7  // Inputs:
8  //   - C  stored in column-major order, with leading dimension Clda
9  //   - Ar packed by columns, with leading dimension mr
10 //   - Br packed by rows, with leading dimension nr
11
12   int       iv, j, jv, pr, baseA = 0, baseB = 0;
13   vregister Cr[mv][nr] // Micro-tile of C
14             ar[mv], br[nv]; // Single column/row of Ar/Br
15
16   vinit();
17
18   // Load the micro-tile of C into vector registers,
19   // each with vl_fp32 FP32 elements
20   for ( j=0; j<nr; j++ )
21     for ( iv=0; iv<mv; iv++ )
22       vload(Cr[iv][j], &C[j*Clda+iv*vl_fp32]);
23
24   for ( pr=0; pr<kc; pr++ ) {
25     // Load the pr-th column/row of Ar/Br into vector registers
26     for ( iv=0; iv<mv; iv++ )
27       vload(ar[iv], &Ar[baseA+iv*vl_fp32]);
28     for ( jv=0; jv<nv; jv++ )
29       vload(br[jv], &Br[baseB+jv*vl_fp32]);
30
31     // Update the micro-tile with axpy
32     for ( iv=0; iv<mv; iv++ )
33       for ( jv=0; jv<nv; jv++ )
34         for ( j=0; j<vl_fp32; j++ )
35           vupdate(Cr[iv][jv*vl_fp32+j], ar[iv], br[jv], j);
36     baseA += mr; baseB += nr;
37   }
38
39   // Store the micro-tile in memory
40   for ( j=0; j<nr; j++ )
41     for ( iv=0; iv<mv; iv++ )
42       vstore(&C[j*Clda+iv*vl_fp32], Cr[iv][j]);
43 }
```

*Requiere soporte para arrays de registros vectoriales

Generación automática de Micro-kernels

```c
// Macros for ARM Neon
#define vl_fp32 4

#define vregister float32x4_t
#define vinit()
#define vload(vreg, mem)  vreg = vld1q_f32(mem)
#define vstore(mem, vreg) vst1q_f32(mem, vreg)
#define vupdate(vreg1, vreg2, vreg3, j) \
                vreg1 = vfmaq_laneq_f32(vreg1, vreg2, vreg3, j)
```

```c
// Macros for ARM SVE
#define vl_fp32 16

#define vregister svfloat32_t
#define vinit() svbool_t pred = svwhilelt_b32_u32(0, vl_fp32)
#define vload(vreg, mem)  vreg = svld1_f32(pred, mem)
#define vstore(mem, vreg) svst1_f32(pred, mem, vreg)
#define vupdate(vreg1, vreg2, mem, j) \
                vreg1 = svmla_n_f32_z(pred, vreg1, vreg2, mem)
```

```c
// Macros for Intel AVX-512
#define vl_fp32 16

#define vregister __m512
#define vinit()
#define vload(vreg, mem)  vreg = _mm512_loadu_ps(mem)
#define vstore(mem, vreg) _mm512_storeu_ps(mem, vreg)
#define vupdate(vreg1, vreg2, vreg3, j) \
                vreg1 = _mm512_fmadd_ps(vreg2, \
                        _mm512_set1_ps(vreg3[j]), \
                        vreg1)
```

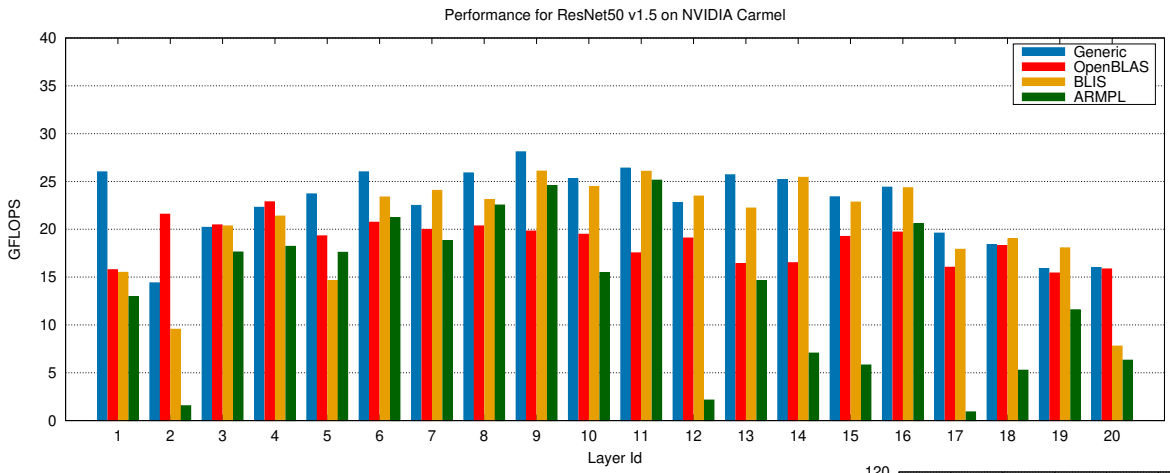
```c
// Macros for RISC-V V
#define vl_fp32 16

#define vregister vfloat32m4_t
#define vinit() \
            float init[vl_fp32]; \
            memset(&init, 0.0, vl_fp32); \
            vle32_v_f32m4(&init, vl_fp32))
#define vload(vreg, mem)  vreg = vle32_v_f32m4(mem, vl_fp32)
#define vstore(mem, vreg) vse32_v_f32m4(mem, vreg, vl_fp32)
#define vupdate(vreg1, vreg2, vreg3, j) \
          vreg1 = vfmacc_vf_f32m4(vreg1, vreg3[j], vreg2, \
                                  vl_fp32)
```

```c
#define mv     (mr/vl_fp32)
#define nv     (nr/vl_fp32)

gemm_ukernel_Cresident_SIMD_mrxnr( int kc, float *Ar, float *Br,
                                   float *C,  int Clda ) {
// mr x nr micro-kernel with C resident in regs.
// Inputs:
//   - C  stored in column-major order, with leading dimension Clda
//   - Ar packed by columns, with leading dimension mr
//   - Br packed by rows, with leading dimension nr
```

+ !Independente de la arquitectura destino!

```c
  for ( pr=0; pr<kc; pr++ ) {
    // Load the pr-th column/row of Ar/Br into vector registers
    for ( iv=0; iv<mv; iv++ )
      vload(ar[iv], &Ar[baseA+iv*vl_fp32]);
    for ( jv=0; jv<nv; jv++ )
      vload(br[jv], &Br[baseB+jv*vl_fp32]);

    // Update the micro-tile with axpy
    for ( iv=0; iv<mv; iv++ )
      for ( jv=0; jv<nv; jv++ )
        for ( j=0; j<vl_fp32; j++ )
          vupdate(Cr[iv][jv*vl_fp32+j], ar[iv], br[jv], j);
    baseA += mr; baseB += nr;
  }

  // Store the micro-tile in memory
  for ( j=0; j<nr; j++ )
    for ( iv=0; iv<mv; iv++ )
      vstore(&C[j*Clda+iv*vl_fp32], Cr[iv][j]);
}
```

Generación automática de Micro-kernels

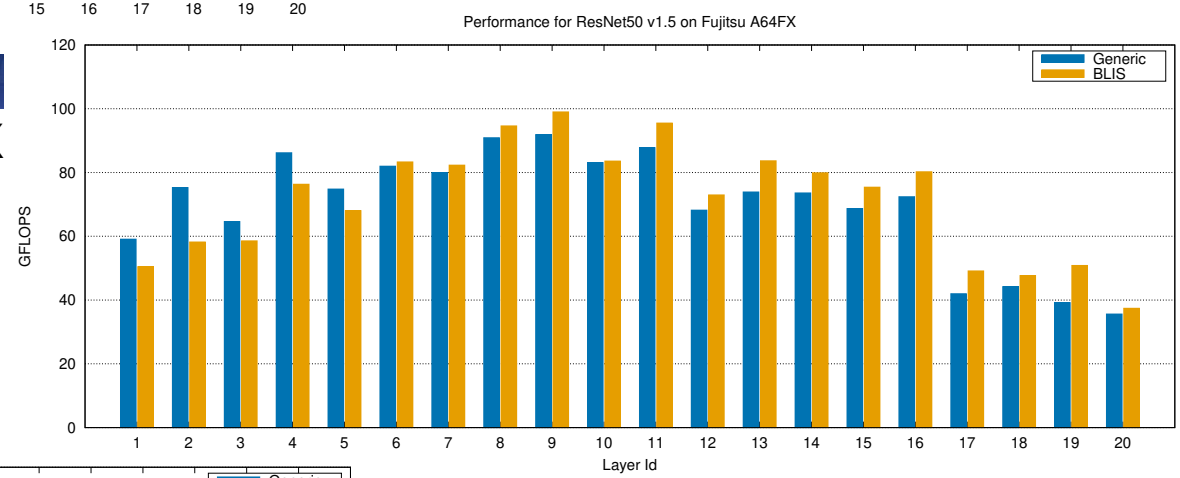


arm NEON AARCH 64

NVIDIA Carmel

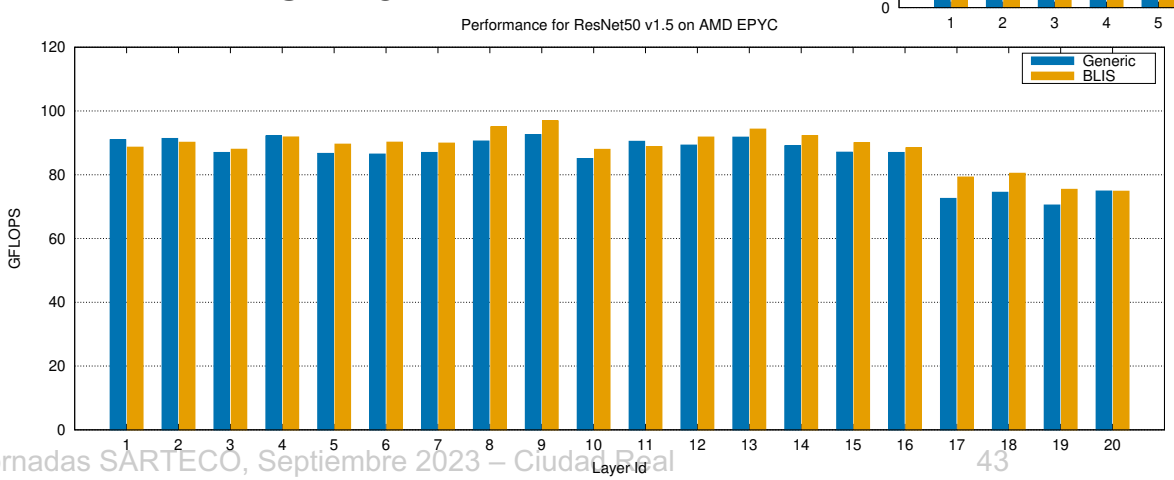
Arm SVE

Fujitsu A64FX



AMD AVX2

AMD EPYC 7282



The Journal of Supercomputing (2023) 79:8124–8147
https://doi.org/10.1007/s11227-022-05003-3

Check for updates

Micro-kernels for portable and efficient matrix multiplication in deep learning

Guillermo Alaejos[1] · Adrián Castelló[1] · Héctor Martínez[2] · Pedro Alonso-Jordá[1] · Francisco D. Igual[3] · Enrique S. Quintana-Ortí[1]

Generación automática de Micro-kernels

B) Scripts de Python

```
12  for c in range(1, nr): #Pointers to columns of C, column stride=ldC
13    fout.write(f"add C0{c}, C0{c-1}, ldC\n")
14  #
15  for c in range(0, nr):                          # Load the micro-tile
16    fout.write(f"ldr Crq0{c}, [C0{c}]\n")  # of C into vector regs.
17    addr = shift
18    for r in range(1, mr // vl):
19      fout.write(f"ldr Crq{r}{c}, [C0{c}, #{addr}]\n")
20      addr += shift
21  #
22  fout.write(f".LOOP_{mr}x{nr}:\n")
23  addr = shift
24  for r in range(0, mr // vl):
25    fout.write(f"ldr arq{r},  [Ar]\n")
26    fout.write(f"add Ar, Ar, #{addr}\n")
27  #
28  for c in range(0, nr // vl):
29    fout.write(f"ldr brq{c},  [Br]\n")
30    fout.write(f"add Br, Br, #{addr}\n")
31  #
32  for r in range(0, mr // vl):
33    for c in range(0, nr // vl):
34      for v in range(0, vl):
35        Cp = c*vl+v
36        fout.write(f"fmla Crv{r}{Cp}, arv{r}, brv{c}[{v}]\n")
37  #
38  fout.write(f"sub kc, kc, 1\n")                  # Prepare for the
39  fout.write(f"cmp kc, 0\n")                      # next iteration
40  fout.write(f"b.ne .LOOP_{mr}x{nr}\n")
41  #
```

+ Direct generation of assembly code: low-level optimizations (prefetching)
- Dependent of processor architecture, data type

Generación automática de Micro-kernels

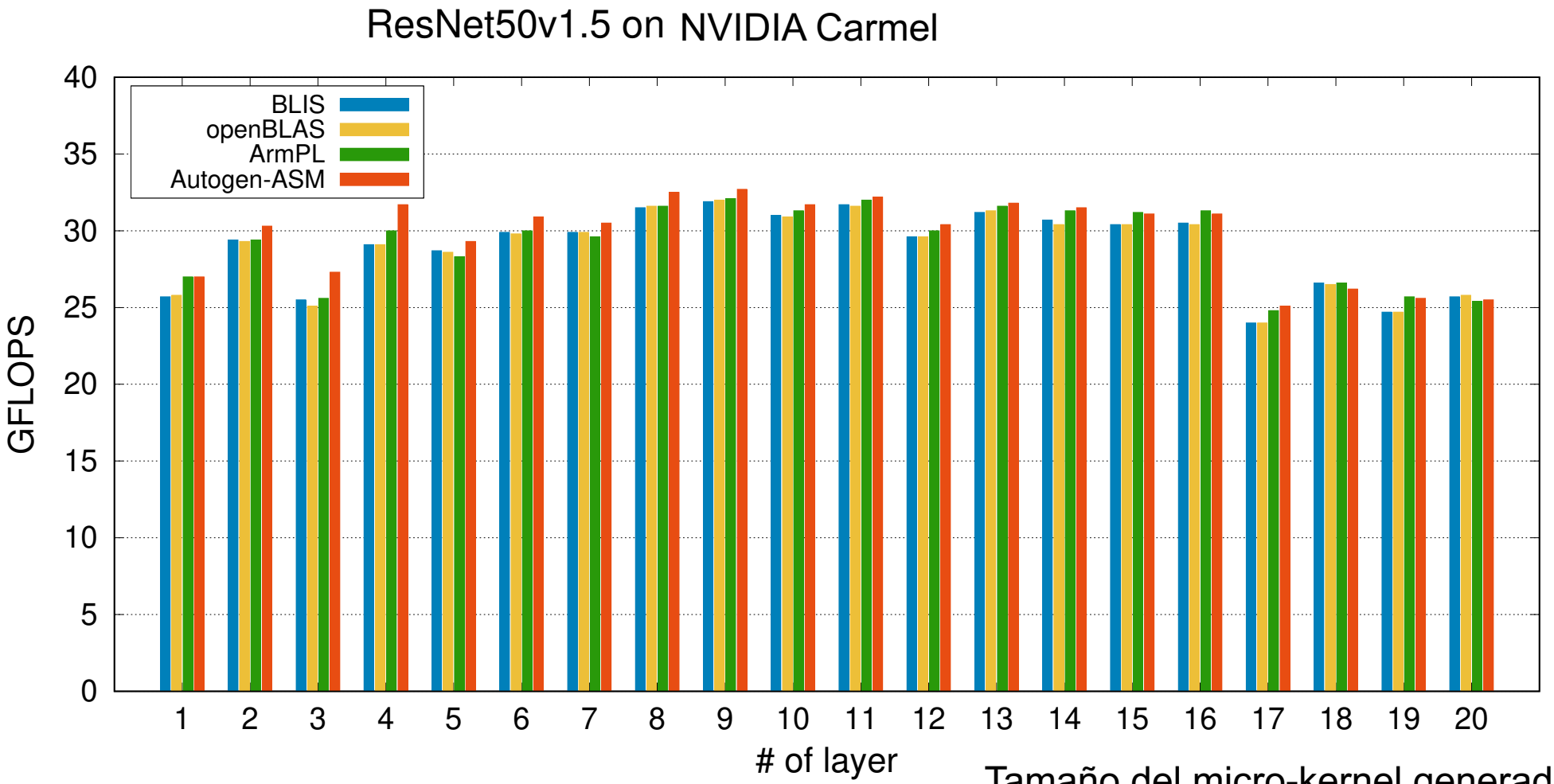
B) Scripts de Python

```
.text
.align      5
.global     gemm_ukernel_NEON_asm_4x4

gemm_ukernel_NEON_asm_4x4:
// Save variables to stack
// Omitted for brevity

add C01, C00, ldC           // Pointers
add C02, C01, ldC           // to columns of C,
add C03, C02, ldC           // column stride=ldC

ldr Crq00, [C00]            // Load the
ldr Crq01, [C01]            // micro-tile of C
ldr Crq02, [C02]            // into vector
ldr Crq03, [C03]            // registers

.LOOP_4x4:                  // Main loop
ldr arq0,   [Ar]            // Load the pr-th
add Ar, Ar, #shift          // column/row
ldr brq0,   [Br]            // of Ar,Br
add Br, Br, #shift

fmla Crv00, arv0, brv0[0]   // Update the
fmla Crv01, arv0, brv0[1]   // micro-tile
fmla Crv02, arv0, brv0[2]   // with AXPYs
fmla Crv03, arv0, brv0[3]

sub kc, kc, 1               // Prepare for
cmp kc, 0                   // the next
b.ne .LOOP_4x4              // iteration

str Crq00, [C00]           // Store the
str Crq01, [C01]           // micro-tile
str Crq02, [C02]           // in memory
str Crq03, [C03]

END:
// Restore variables from stack
// Omitted for brevity
ret
```

```
// Inputs
#define kc     x0
#define Ar     x1
#define Br     x2
#define C      x3
#define ldC    x4

// ARM NEON
// vl * sizeof
// datatype (FP32)
#define shift 16

// Addresses for
// remaining
// columns of C
#define C00    C
#define C01    x5
#define C02    x6
#define C03    x7

// Vector registers
// for micro-tile
// of C
#define Crq00 q0
#define Crv00 v0.4s
#define Crq01 q1
#define Crv01 v1.4s
#define Crq02 q2
#define Crv02 v2.4s
#define Crq03 q3
#define Crv03 v3.4s

// Vector registers
// for column/row
// of Ar,Br
#define arq0   q4
#define arv0   v4.4s
#define brq0   q5
#define brv0   v5.s
```

Generación automática de Micro-kernels

B) Scripts de Python



Tamaño del micro-kernel generado automáticamente igual al utilizado por BLIS. Objetivo: mostrar rendimiento similar

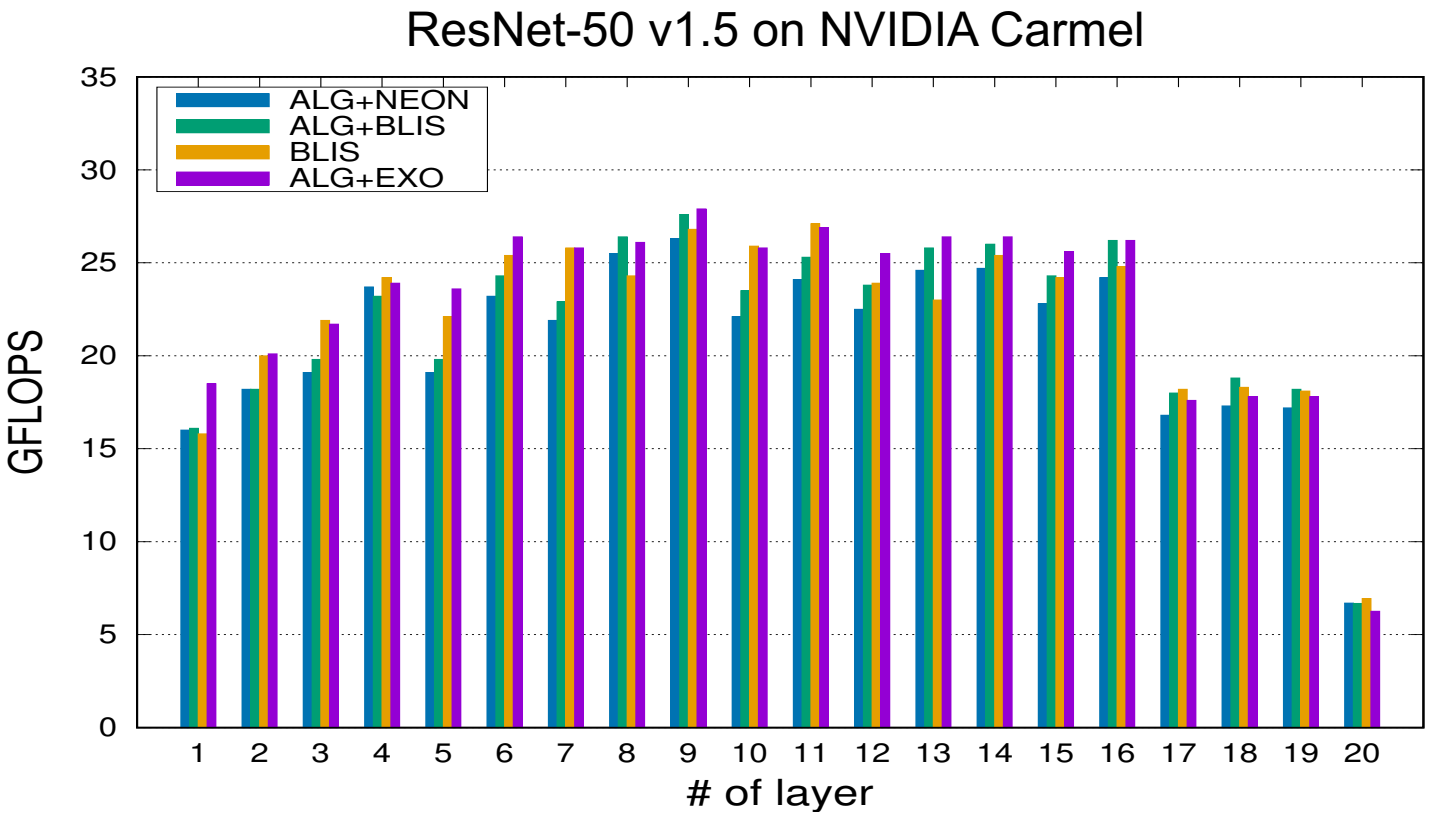
Generación automática de Micro-kernels

C) Lenguaje Exo

```
# USER CODE
p = unroll_loop(p,'it')
p = unroll_loop(p,'jt')

# RESULTING EXO GENERATED CODE
def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
  Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
  beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
  ):

  # Registers for C and load C as in
  # previous figures.  Omitted for brevity

  # Registers for Ac and Bc
  A_reg: R[2, 4] @ Neon
  B_reg: R[3, 4] @ Neon

  # C += Ac * Bc
  for k in seq(0, KC):
    # Unrolled loads from Ac and Bc to registers
    neon_vld_4xf32(A_reg[0, 0:4], Ac[k, 0:4 + 0])
    neon_vld_4xf32(A_reg[1, 0:4], Ac[k, 4:4 + 4])
    neon_vld_4xf32(B_reg[0, 0:4], Bc[k, 0:4 + 0])
    neon_vld_4xf32(B_reg[1, 0:4], Bc[k, 4:4 + 4])
    neon_vld_4xf32(B_reg[2, 0:4], Bc[k, 8:4 + 8])
    # Computation with registers
    for jt in seq(0, 3):
      for it in seq(0, 2):
        for jtt in seq(0, 4):
          neon_vfmla_4xf32_4xf32(
            C_reg[jtt + 4 * jt, it, 0:4],
            A_reg[it, 0:4], B_reg[0:4, jt], jtt)

  ;
  ;
```

Generación automática de Micro-kernels

C) Lenguaje Exo



Generación automática de GEMM

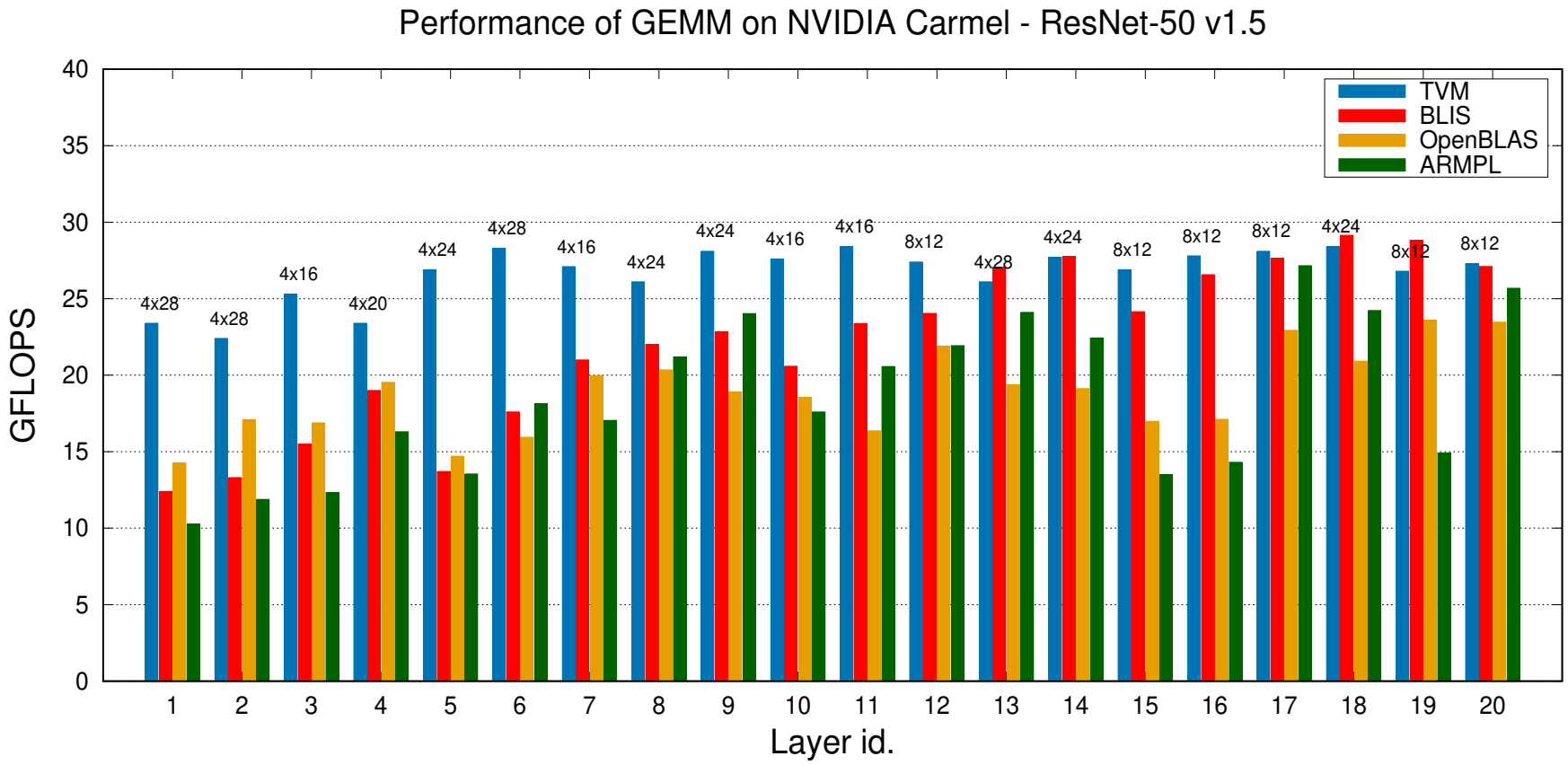
C) Apache TVM

```
def packed_GEMM(m, n, k, mc, nc, kc, mr, nr):
  # P1) Define operation
  A = te.placeholder((m, k), name="A")
  B = te.placeholder((k, n), name="B")

  # 4D view into A and B to induce creation
  # of buffer by TVM
  Ac = te.compute((math.ceil(k/kc),
                   math.ceil(m/mr), kc, mr),
                   lambda i, j, q, r:
                   A[j * mr + r, i * kc + q],
                   name="Ac")
  Bc = te.compute((math.ceil(k/kc),
                   math.ceil(n/nr), kc, nr),
                   lambda i, j, q, r:
                   B[ i * kc + q, j * nr + r ],
                   name="Bc")
```

```

  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda i, j:
         te.sum(Ac[p//kc, i//mr,
                   tvm.tir.indexmod(p, kc),
                   tvm.tir.indexmod(i, mr)] *
                Bc[p//kc, j//nr,
                   tvm.tir.indexmod(p, kc),
                   tvm.tir.indexmod(j, nr)],
                axis=p), name="C")

  # P2) Prepare schedule
  sched = te.create_schedule(C.op)
  ic, jc, \
  ir, jr  = sched[C].tile(C.op.axis[0],
                          C.op.axis[1],
                          mc, nc)
  pc, pr = sched[C].split(p, factor=kc)

  # P3) Place Ac, Bc in the desired loops
  sched[Bc].compute_at(sched[C], pc)
  sched[Ac].compute_at(sched[C], ic)

  # P4) Loop schedule as in B3A2C0
  sched[C].reorder(jc, pc, ic, jr, ir, pr)

  # P5) Generate code with LLVM backend
  return tvm.build(sched, [A, B, C],
                   target="llvm")
```

Generación automática de GEMM

C) Apache TVM



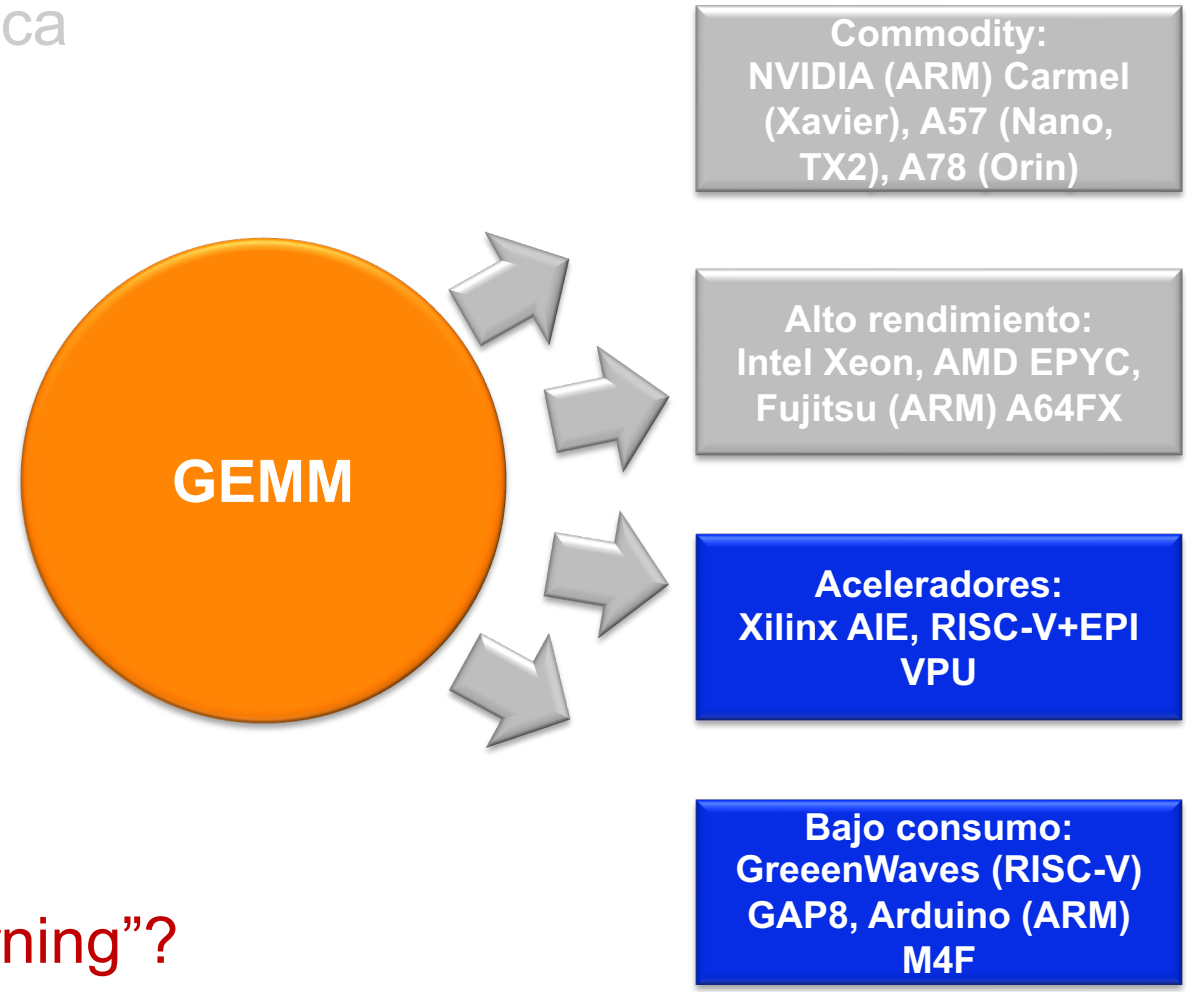
ACM Trans. Math. Software, 2016

Algorithm XXX: Automatic Generators for a Family of Matrix Multiplication Routines with Apache TVM

GUILLERMO ALAEJOS, Universitat Politècnica de València, Spain
ADRIÁN CASTELLÓ, Universitat Politècnica de València, Spain
PEDRO ALONSO-JORDÁ, Universitat Politècnica de València, Spain
FRANCISCO D. IGUAL, Universidad Complutense de Madrid, Spain
HÉCTOR MARTÍNEZ, Universidad de Córdoba, Spain
ENRIQUE S. QUINTANA-ORTÍ, Universitat Politècnica de València, Spain

Índice

- …
- Generación automática
- Portabilidad

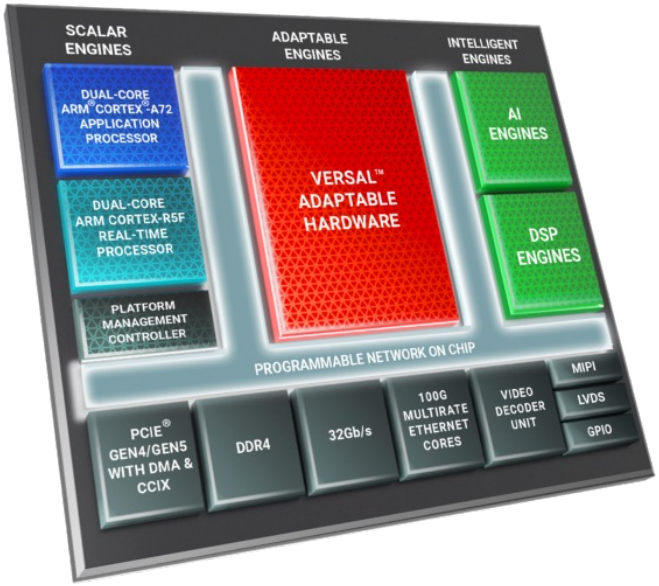


¿“Transfer learning”?

Nuevos Procesadores

Xilinx AIE tile en Xilinx Versal AI

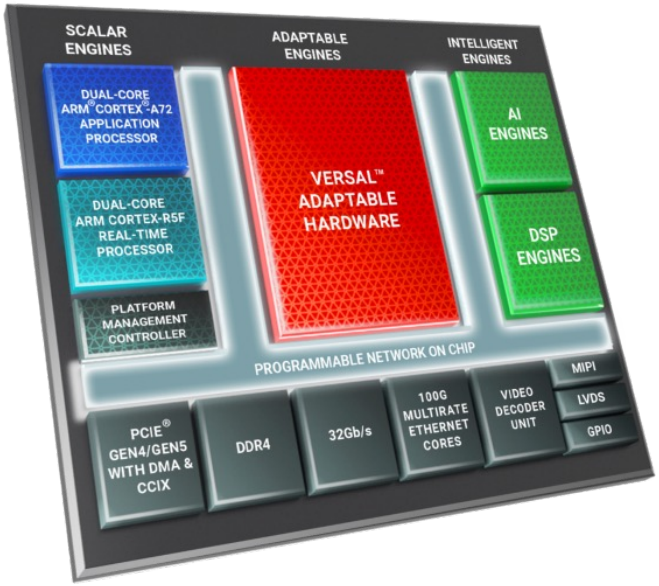
- Diseño heterogéneo:
  - AIE, FPGA, ARM
- Memoria local en AIE + RAM
- Acumuladores, muchas posibilidades para aritmética mixta



Nuevos Procesadores

Xilinx AIE tile en Xilinx Versal AI

- Diseño heterogéneo:
  - AIE, FPGA, ARM
- Memoria local en AIE + RAM
- Acumuladores, muchas posibilidades para aritmética mixta



- Uso de memoria local como caché L1; FPGA como caché L2 (scratchpad)
- Pre-empaquetar el tensor de filtros en FPGA

IEEE 31st Euromicro PDP, 2023

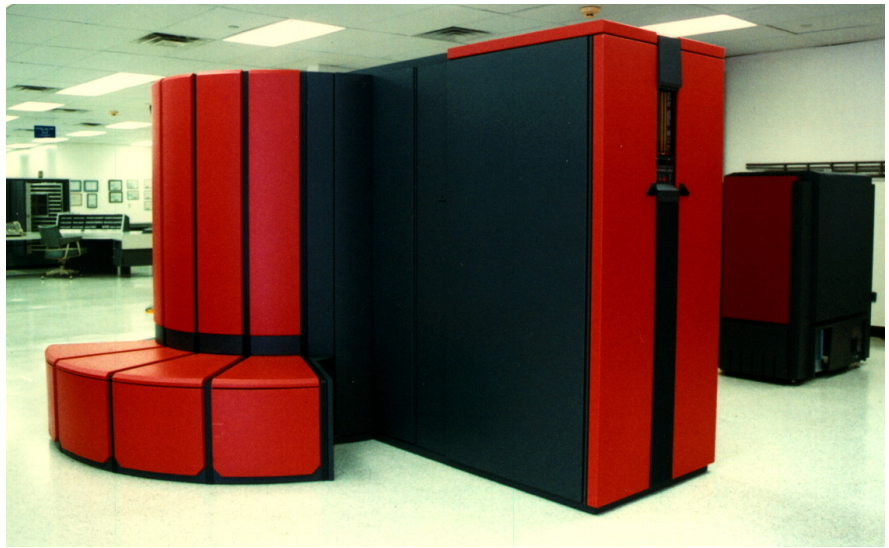
Toward Matrix Multiplication for Deep Learning Inference on the Xilinx Versal

Jie Lei, José Flich, Enrique S. Quintana-Ortí
Depto. de Informática de Sistemas y Computadores
Universitat Politècnica de València, Valencia, Spain
{jlei,jflich,quintana}@disca.upv.es

Nuevos Procesadores

Cray YMP, NEC pero también RISC-V + Vector Processing Unit (VPU)

- Érase una vez,… hace 25 años

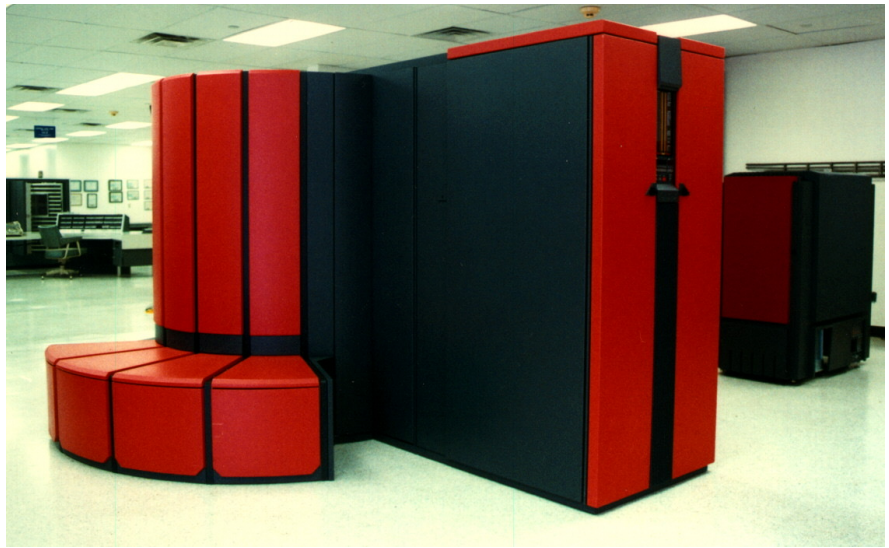


- Compilador vectorizante
- Sin caché
- Registros SIMD muy largos (p.e., 512 FP32)

Nuevos Procesadores

Cray YMP, NEC pero también RISC-V + Vector Processing Unit (VPU)

- Érase una vez,… hace 25 años



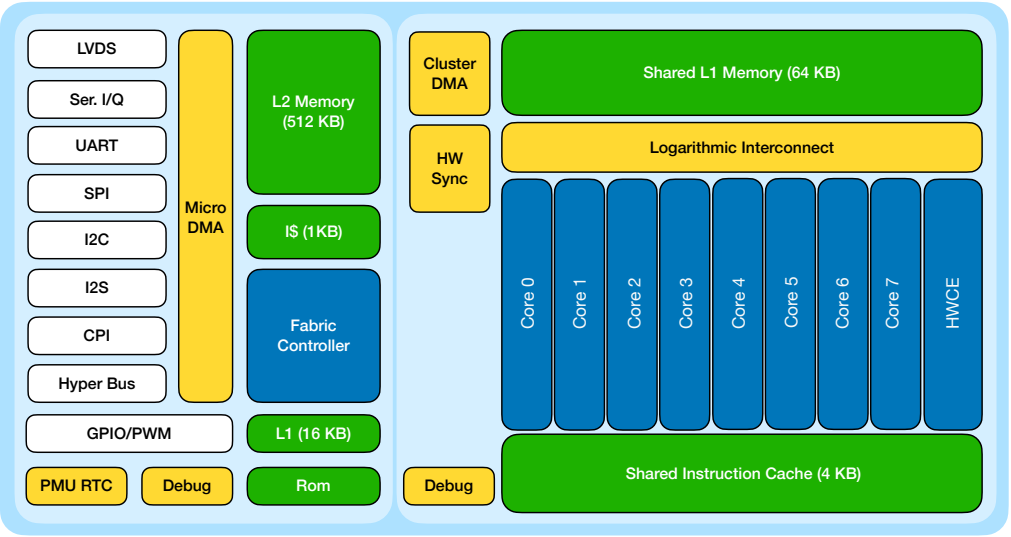
- Compilador vectorizante
- Sin caché
- Registros SIMD muy largos (p.e., 512 FP32)

- Personalizar el micro-kernel de GEMM
- No empaquetar los datos en buffers

Nuevos Procesadores

GreenWaves (RISC-V) GAP8

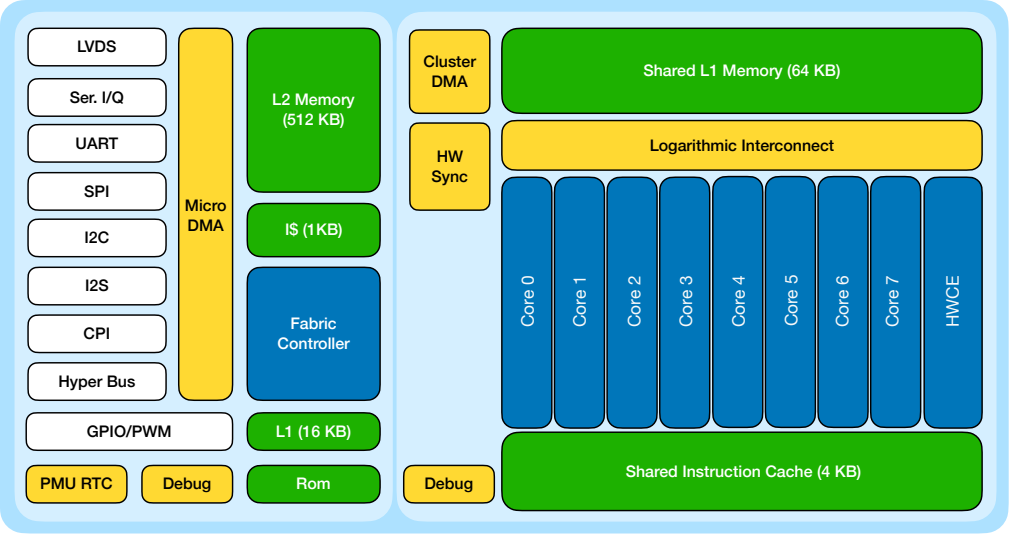
- Procesador IoT de bajo consumo
- Memorias scratpad, sin caché
- Aritmética INT
- Producto DOT en lugar de AXPY



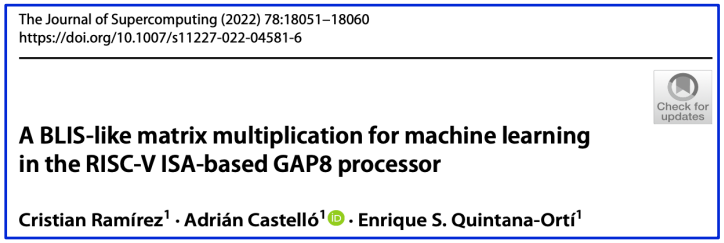
Nuevos Procesadores

GreenWaves (RISC-V) GAP8

- Procesador IoT de bajo consumo
- Memorias scratpad, sin caché
- Aritmética INT
- Producto DOT en lugar de AXPY



- Integrar los movimientos de datos en las rutinas de empaquetamiento
- Reorganizar el algoritmo GEMM para utilizar A en registros vectoriales


The Journal of Supercomputing (2022) 78:18051–18060
https://doi.org/10.1007/s11227-022-04581-6

Check for updates

A BLIS-like matrix multiplication for machine learning in the RISC-V ISA-based GAP8 processor

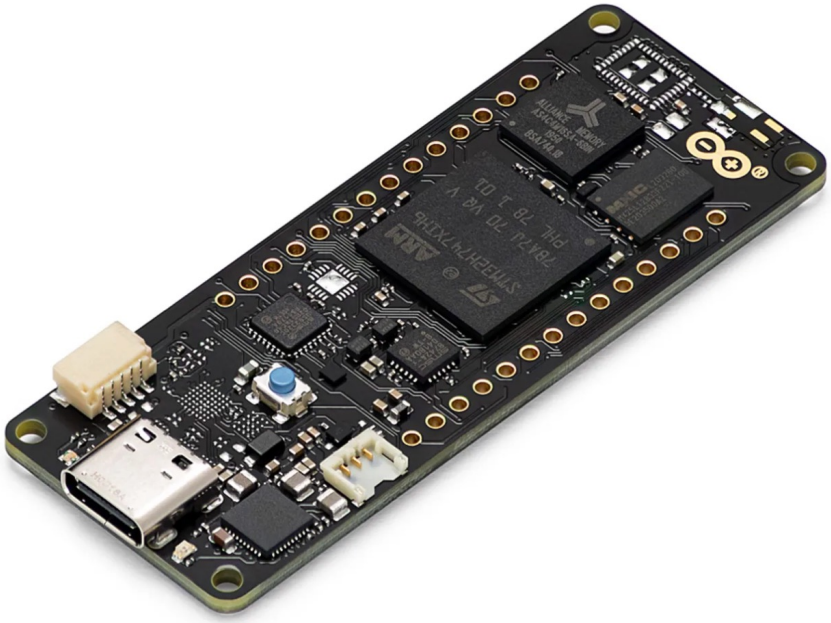
Cristian Ramírez[1] · Adrián Castelló[1] · Enrique S. Quintana-Ortí[1]

Nuevos Procesadores

STMicroelectronics STM32H747XI

- ARM Cortex M4
- Sin caches, sin scratchpad

  864 KB SRAM, 2 MB FLASH
- Aritmética INT/FP

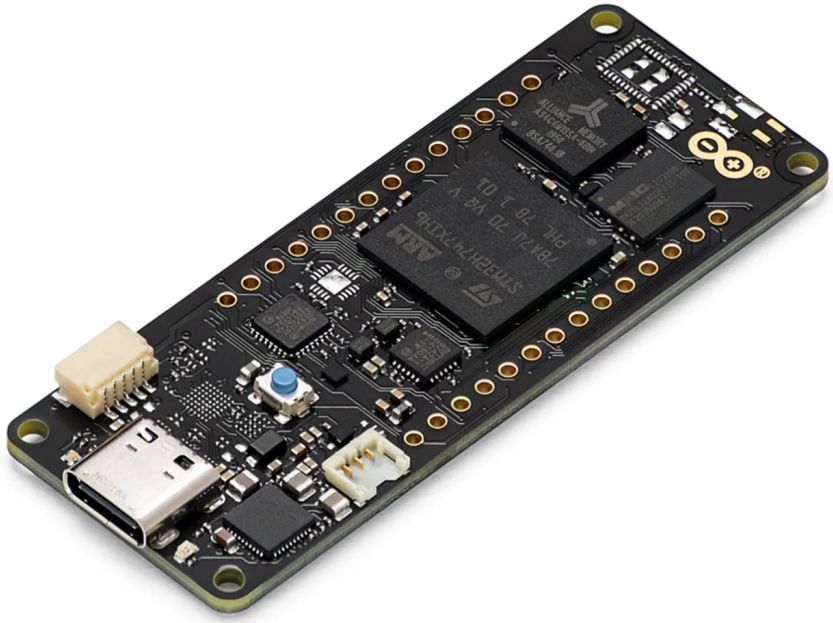


Nuevos Procesadores

STMicroelectronics STM32H747XI

- ARM Cortex M4
- Sin caches, sin scratchpad

  864 KB SRAM, 2 MB FLASH
- Aritmética INT/FP



- Empaquetar IM2COL/IM2ROW en grupos de 2-3 columnas
- Desarrollar micro-kernels específicos
- Pre-cargar filtros en SRAM

IEEE INTERNET OF THINGS JOURNAL

Optimising Convolutions for Deep Learning Inference on ARM Cortex-M Processors

Antonio Maciá-Lillo, Sergio Barrachina, Germán Fabregat, Manuel F. Dolz

Conclusiones

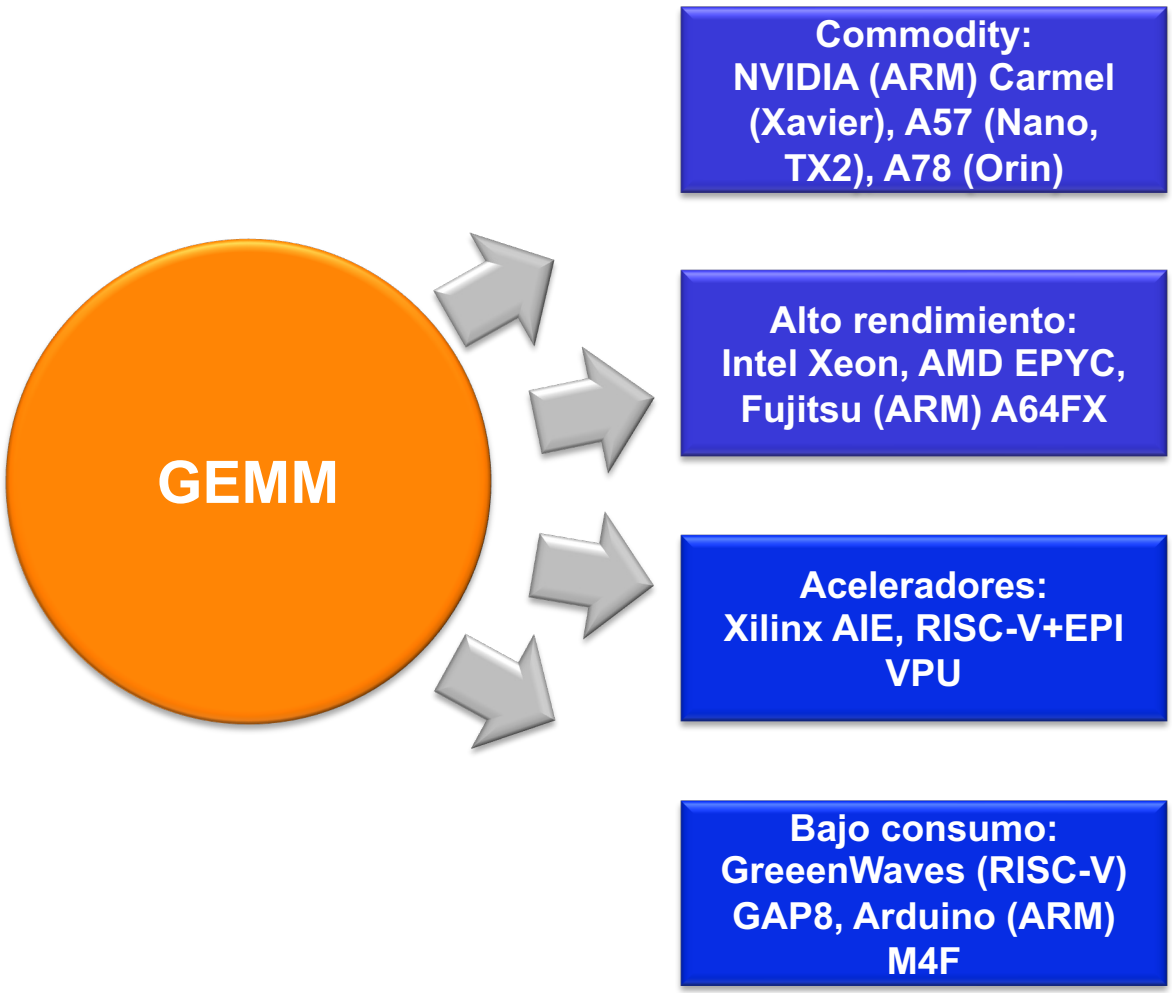
- GEMM es un kernel muy conocido en computación científica

  La optimización para procesadores commodity y HPC es un tema bien entendido

…pero las implementaciones actuales de BLAS no están optimizadas para DL

Conclusiones

- La generación automática es possible para procesadores convencionales
- Las ideas generales son portables a arquitecturas más novedosas



Conclusiones

- Matrix Engines
- DL y compresión dispersa
- Precisión mixta, cuantización y entrenamiento
- …

Agradecimientos

España: PID2020-113656RB, TED2021-129334-B-I00, CIPROM/2022/20, SUBV-36/2021

Europa: 955558, 955776, 956090, 101070779, 101057457

Otros:

Agradecimientos

Una Visión Computacional a las RNPs

Enrique S. Quintana Ortí

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

G. Alaejos
P. Alonso-Jordá
S. Barrachina
A. Castelló
M. F. Dolz
F. D. Igual
J. Lei
T. M. Low
A. Maciá
H. Martínez
C. Ramírez
R. Rodríguez
P. San Juan
U. Sridhar
A. E. Tomás

Carnegie Mellon University

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSIDAD COMPLUTENSE MADRID

UNIVERSITAT JAUME I

Universitat d'Alacant
Universidad de Alicante

UNIVERSIDAD DE CÓRDOBA