Laboratoires de Programmation parallèle sur les architectures ARM

SmartComputerLab

Contenu

Laboratoires de Programmation parallèle sur les architectures ARM

SmartComputerLab

Le module de la **Programmation Massivement Parallèle** sur GPU est composé d'une introduction aux architectures GPU et à la programmation massivement parallèle (**GPGPU**) sur ces dispositifs.
GPGPU signifie **General Programming on GPU**.

Les programmes qu'on va développer peuvent être exécutés en **local** ou à la **distance Lab0/Lab1/Lab2/Lab3** ou seulement en local pour le **Lab4**. Les codes du **Lab3** peuvent fonctionner seulement si vous avez sur votre ordinateur client l'environnement graphique **X11**.
X11 est propre au système **Linux** (Ubuntu) et il n'est directement disponible sur **Windows 10** ou sur le **MacOS. L'exploitation de X11 sur Windows 10 nécessite l'installation d'une application spécifique.**

Dans le **Lab4** nous utilisons directement la **mémoire vidéo** de la **GPU** ; **elle ne pas disponible à distance**.

Le travail complet en local nécessite une carte **Nvidia Jetson TX1/Nano** ou la présence d'une carte Nvidia sur votre ordinateur.

Le travail à distance est possible grâce à la mise à disposition d'un serveur **Nvidia Jetson Xavier** (**SmartComputerLab**) ou l'utilisation de **Google Colab** avec un script de préparation.

Jetson Xavier est accessible aux étudiants **SMTR** de l'année **2020/21.**

connection par ssh :

ssh -X levine@86.217.12.14

L'adresse peut être changée – il sera confirmée avant le début de Laboratoire1
Chaque étudiant a un compte qui correspond au **prénom** de l'étudiant, mot de passe étant son **nom de famille**.

Compte : **`levine`**
Mot de passe : **`decure`**

Les exemples du code déposé sont des codes **exécutables**. Ils permettent de tester les fonctionnalités.
Les programmes source **sont à coder et à exécuter par vos propre soins**.

Attention :
Avant de pouvoir accéder au compilateur **`nvcc`** vous devrais avoir les lignes suivantes dans votre **`.bashrc`**

```
export PATH=/usr/local/cuda-10.2/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-10.2/lib64:$LD_LIBRARY_PATH
PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig
export PKG_CONFIG_PATH
```

Vérification avec :
`env $PATH`

Le **Lab0** est un **rappel sur la programmation parallèle** sur la base des unités **NEON** et de **CPUs multi-cores**

Laboratoire 0 (un rappel sur `NEON` et `openMP`)

Dans ce laboratoire initial nous introduisons brièvement deux modes de programmation parallèle : programmation parallèle et vectorielle sur les unités **NEON** et une programmation sur multi-cores avec **openMP**.

La technologie **NEON** est une architecture **SIMD** (Single Instruction, Multiple Data) avancée pour les processeurs de la série Arm Cortex-A. Elle est nécessaire pour accélérer les algorithmes de traitement multimédia et de signal tels que le codeur/décodeur vidéo, les graphiques 2D / 3D, les jeux, le traitement audio et vocal, le traitement d'image, la téléphonie et le son.

Les instructions NEON exécutent le traitement type **Packed SIMD**:

1. Les registres sont considérés comme des vecteurs d'éléments du même type de données
2. Les types de données peuvent être: 8 bits, 16 bits, 32 bits, 64 bits, en virgule flottante simple précision sur la plate-forme ARM 32 bits, à la fois en virgule flottante simple précision et en virgule flottante double précision point sur la plate-forme ARM 64 bits.
3. Les instructions effectuent la même opération dans toutes les voies (**lanes**)

0.1 Programmation de NEON avec C

Programmatio au niveu de l'assembleur pour NEON peut créer du code fortement optimisé au prix de passer plus de temps à écrire du code. L'écriture de code en C signifie un temps de développement plus rapide et un code plus maintenable. Les compilateurs font normalement un bon travail d'optimisation du code. Parfois, il est nécessaire de se tourner vers l'assemblage pour des performances finement réglées, mais dans la plupart des cas, l'utilisation de C donne de bonnes performances, nettement meilleures que le codage manuel à l'aide d'instructions ARM (qui sont déjà rapides).

Un compilateur ne peut pas prendre du code standard et utiliser des instructions NEON, même s'il y a de nombreuses boucles, ou dans les cas où NEON pourrait accélérer le code.
Le **compilateur doit être spécifiquement informé** d'utiliser le moteur NEON.
Il existe plusieurs façons de procéder comme décrit ici.

0.1.1 Programmation avec les extensions intrinsèques (intrinsics)

Les fonctions et types de données **intrinsèques** fournissent un lien direct vers l'assemblage, tout en conservant des fonctions de niveau supérieur telles que la vérification de type et l'allocation automatique de registre. Cela permet le développement du code C lisible et facile à maintenir, sans avoir besoin d'écrire des instructions d'assemblage directes.
Pour utiliser les **intrinsèques NEON**, incluez le fichier d'en-tête `arm_neon.h`.

0.1.1.1 Vector Data Types

Les intrinsèques permet de définir toute sorte de type de données accepté par NEON. Les types de données NEON sont des noms selon ce modèle:

```
<type><size>x<number of lanes>_t
```

Quelques exemples:

```
int32x4_t   // vector of forur 32-bit elements
uint16x8_t  // vector of eight 16-bit elements
int8x8x2_t  // array of two vectors with eight 8-bit elements
```

Le type peut être un `int`, `uint`, `float` ou `poly`. La taille est la taille de chaque voie (lane) et le nombre de voies définit le nombre de voies chargées, et donc le type de registre utilisé (**D** ou **Q**).
Par exemple pour charger une série de pixels dans un registre **D 64 bits**, chaque pixel étant une valeur **non signée 8 bits**, choisissez `uint8x8_t`.

0.1.1.2 Chargement d'un seul vecteur à partir de la mémoire

Pour charger des données dans un **registre NEON**, des éléments intrinsèques ont été créés qui ressemblent au code d'un assembleur, mais ajoutent des types de données pour faciliter la vérification du compilateur. Ils renvoient le type de données que le ou les registres peuvent contenir.

Pour charger un seul vecteur, un intrinsèque est utilisé qui utilise l'adresse mémoire comme argument et renvoie le type de données contenu dans le registre.

```
// VLD1.8 {d0, d1}, [r0]
uint8x16_t vld1q_u8(__transfersize(16) uint8_t const * ptr);
// VLD1.16 {d0, d1}, [r0]
uint16x8_t vld1q_u16(__transfersize(8) uint16_t const * ptr);
// VLD1.32 {d0, d1}, [r0]
uint32x4_t vld1q_u32(__transfersize(4) uint32_t const * ptr);
…
```

0.1.1.3 Chargement de plusieurs vecteurs à partir de la mémoire

Le chargement de plusieurs vecteurs à partir de la mémoire revient à charger un seul vecteur, sauf que **l'entrelacement** doit être spécifié. Les instructions sont presque identiques aux instructions simples; le pointeur de mémoire est passé en argument et le type de données résultant est renvoyé.
L'entrelacement est défini dans l'instruction.

```
uint8x8_t data vld1_u8(src); //Loads one d-word register
uint8x8x2_t data2 vld2_u8(src); //Loads two d-word registers, using interleave 2
uint8x8x3_t data2 vld3_u8(src); //Loads three d-word registers, interleave 3
```

0.1.2 Utilisation de NEON Intrinsics

Les éléments intrinsèques NEON sont bien conçus; ils sont facilement accessibles depuis C sans aucun changement majeur.
Tant que la procédure logique est respectée, lisez les données en utilisant les intrinsèques, exécutez les instructions NEON, puis écrivez les données, à nouveau en utilisant les intrinsèques.
Ensuite, la routine peut bénéficier de l'optimisation NEON.

Il est possible de mélanger les instructions ARM et NEON, mais il y a parfois une pénalité à le faire;
NEON ne peut utiliser **que les registres NEON**, tout comme ARM ne peut utiliser que les instructions ARM.

Les registres devront être transférés vers et depuis le moteur NEON, ce qui coûtera un léger **surcoût**.
Vous pouvez également créer des portions de code qui ne s'exécutent que si un moteur NEON est présent (et défini), en utilisant les sections **`#ifdef`**.

```
ifdef __ARM_NEON__
// NEON code
#else
// ARM code
#endif
```

En utilisant ce système, vous pouvez générer un code source facilement transférable d'une conception de processeur à une autre en utilisant la norme C.

0.1.3 Un exemple simple d'addition de vecteurs avec les intrinsèques NEON

Voici un exemple d'ajout de vecteur de 16 éléments (octets). Le premier extrait du code illustre une addition simple contrôlée par une **boucle `for`**. Afin de comparer les performances de cette solution avec l'implémentation sur l'unité **NEON** la boucle est exécutée **10000 fois**.

```
unsigned char datain[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
unsigned char dataout[16];
..
void addx (unsigned char *in,unsigned char *out,unsigned char x) {
unsigned int i,j;
        for(j=0;j<100000;j++)  // do it 100000 times
        for(i=0;i<16;i++)
                {
                out[i]=in[i]+x;
                in[i]=out[i];
                }
}
```

Maintenant, le même algorithme est préparé pour l'exécution sur **NEON**. Cela implique la **déclaration** et **l'initialisation** des types de données requis par l'utilisation des intrinsèques.
Nous déclarons d'abord le **vecteur de données** comme **`uint8`** (entiers 8 bits non signés). Ensuite, nous chargeons le vecteur dans le registre interne de **128 bits** de l'unité **NEON**. Le contenu de ce registre est considéré comme **16 entiers de 8 bits** – **`uint8x16_t`**.
L'appel de fonction **`addx_neon`** suivant prend deux arguments: le registre de données (**`uint8x16_t * data`**) et la valeur à ajouter aux éléments du registre (**`char x`**).
À l'intérieur de la fonction, la valeur de **`x`** est **répliquée 16 fois** par la fonction intrinsèque **`vmovq_n_u8(x)`** dans **`xvalue`** (**`uint8x16_t`**).
Enfin **`vaddq_u8 (* data, xvalue)`** intrinsic est utilisé pour ajouter **`xvalue`** à tous les éléments du vecteur de données.

```
onst uint8_t uint8_data[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
uint8x16_t data;
/* load our custom data into the 128-bit vector register  */
data = vld1q_u8(uint8_data);
..
void addx_neon(uint8x16_t *data,char x) {
int i,j;
/* set each sixteen values of the vector to x. a 'q' suffix to intrinsics
indicates the instruction run for 128 bits registers.*/
        for(j=0;j<100000;j++)  // do it 100000 times
        {
        uint8x16_t xvalue = vmovq_n_u8(x);
        *data = vaddq_u8(*data, xvalue);
        }
}
```

Le code du programme incluant les fonctions pour tester le temps d'exécution en **micro-secondes**.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include "arm_neon.h"

long timestampsec()
{
struct timeval temps;
gettimeofday(&temps,NULL);
return temps.tv_usec;
}
void print_uint8 (uint8x16_t data, char* name)
{
    int i;
    static uint8_t p[16];
    vst1q_u8 (p, data);
    printf ("%s = ", name);
    for (i = 0; i < 16; i++) { printf ("%02d ", p[i]);}
}

void addx_neon(uint8x16_t *data,char x)
{
int i,j;
// 'q' suffix indicates the instruction run for 128 bits registers.
        for(j=0;j<100000;j++)  // 100000 cycles for test
        {
        uint8x16_t xvalue = vmovq_n_u8 (x); // sets 16 values to x
        *data = vaddq_u8 (*data, xvalue);
        }
}
```

```
void addx (unsigned char *in,unsigned char *out,unsigned char x)
{
unsigned int i,j;
        for(j=0;j<100000;j++)  // 100000 cycles for test
        for(i=0;i<16;i++)
                {
                out[i]=in[i]+x;
                in[i]=out[i];
                }

}

int main (int c, char **a)
{
    int i=0,j=0,t0,t1;
    int mode=0;
    unsigned char datain[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
    unsigned char dataout[16];
    const uint8_t uint8_data[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
    uint8x16_t data;  // vector for data register
    data = vld1q_u8 (uint8_data);  // loads data to register
    if(c<2)
    { printf("usage:%s value_to_add\n",a[0]); exit(1); }
    printf("Chose the execution mode CPU,NEON [0,1]\n");
    scanf("%d",&mode);
    if(mode)print_uint8 (data, "data");
    else { printf("datain="); for(i=0;i<16;i++) printf("%d,",datain[i]);}
    t0=timestampsec();  // start timing
        if(mode) addx_neon(&data,(char)atoi(a[1]));  // NEON selected
    else addx(datain,dataout,(unsigned char)atoi(a[1]));  // CPU selected
    t1=timestampsec();  // stop timing
    printf("\n%d\n",t1-t0);
    if(mode) print_uint8(data,"data (new)");    // print NEON result
    else { printf("dataout=");
        for(i=0;i<16;i++)printf("%d,",dataout[i]);printf("\n");}
    return 0;
}
```

Résultat d'exécution sur **Jetson-Xavier :**

```
bako@xavier:~/MandM/lab0$ ./vectAdd.NEON 5
Chose the execution mode CPU,NEON [0,1]
0
datain=1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
17445
dataout=33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,
bako@xavier:~/MandM/lab0$ ./vectAdd.NEON 5
Chose the execution mode CPU,NEON [0,1]
1
data = 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
5198
data (new) = 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
```

0.1.4 Conversion d'une image en niveaux de gris

Sur presque tous les appareils photo numériques modernes, il existe une option pour convertir les images en **niveaux de gris**. Cette opération est simple; il prend les composants **rouge**, **vert** et **bleu**, calcule une moyenne pondérée, puis écrit le résultat sur un nouveau pixel. C'est le genre de calcul répétitif pour lequel NEON est bien adapté. Pour ce faire, voici un exemple d'application.

Tout d'abord, une petite compréhension de la façon dont nos yeux voient le monde. Les yeux humains sont plus adaptés à voir le **vert** que toute autre couleur.Par conséquent, lors du changement d'une image en niveaux de gris, il ne suffit pas **d'ajouter** le composant rouge, vert et bleu, puis de le **diviser par trois**.

Pour des images claires en niveaux de gris, un **certain poids** est ajouté à chaque couleur. C'est ce qu'on appelle la méthode de la luminosité. Il est courant de multiplier le canal **rouge** par **77**, le canal **vert** par **151** et le canal **bleu** par **28**.
La somme de ces trois nombres est de **256**, ce qui simplifie la division.Pour ce faire, le programme peut remplir **trois registres** avec des valeurs spécifiques, le rapport de poids.

L'application doit lire une série de pixels, séparant les **composants rouge**, **vert** et **bleu** dans des **registres** séparés à l'aide de l'entrelacement. Ensuite, chaque composante de couleur est multipliée par le **coefficient de poids** et le résultat est placé dans un autre registre. Enfin, les nouveaux registres sont ajoutés dans un seul registre, divisés, puis réécrits en mémoire.
Le résultat final est :

```
((rx) + (gy) + (bz)) / (x + y + z)
```

Tout d'abord, **trois registres** doivent être remplis avec des valeurs de 8 bits, les valeurs de coefficient: un pour le composant **rouge**, un pour le **vert** et un pour le **bleu**.
Nous utilisons l'instruction **`vdup_n_u8` p**our prendre une valeur de 8 bits et répéter cette valeur sur le registre **NEON**.

```
uint8x8_t r_ratio = vdup_n_u8(77);
uint8x8_t g ratio = vdup_n_u8(151);
uint8x8_t b ratio = vdup_n_u8(28);
```

Notez qu'en C, il n'est pas nécessaire de spécifier un registre; le compilateur peut le faire automatiquement et garder une trace de quelle variable est contenue dans quel registre.
Regardez d'abord une implémentation de référence en C:

```
void reference_convert(uint8_t* __restrict dest, uint8_t* __restrict src,int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    int r = *src++;int g = *src++; int b = *src++;// load red, green and blue
    int y = (r*77)+(g*151)+(b*28);  // build weighted average:
    *dest++ = (y>>8);
  }
}Maintenant, les données doivent être lues, en utilisant l'entrelacement 3. La
variable rgb est définie comme uint8x8x3_t car elle utilise trois registres.

uint8x8x3_t rgb = vld3_u8(src);
```

`vld3_u8` effectue un **chargement vectoriel** de valeurs 8 bits non signées, en utilisant l'entrelacement 3.
Là encore, vous n'avez pas besoin de spécifier les registres.

Maintenant vient la partie délicate. Chaque pixel a une taille de 8 bits, mais vous devez multiplier chacun d'eux et additionner les résultats de trois multiplications ensemble. Il n'est pas possible de le faire dans une voie 8 bits car il y aura presque certainement une perte de données.

La raison pour laquelle cet exemple utilise uniquement un **registre 64 bits** au lieu d'un **registre 128 bits** est pour cette raison:

Le programme doit élargir les voies de **8 bits à 16 bits** et donc utiliser un registre de sortie plus grand.

Par conséquent, un registre temporaire est défini comme :

```
uint16x8_t temp;
```

Cela réserve un registre **Q** pour un total de **8 variables 16 bits**. Maintenant, multipliez le composant **R** par le **coefficient** et enregistrez-le dans le **registre temporaire**.

```
temp = vmull_u8(rgb.val[0], r_ratio);
```

Cette instruction est une **multiplication vectorielle**, qui dit à NEON de multiplier chaque voie dans `rgb.val[0]` (la composante rouge de chaque pixel) par `r_ratio` (le coefficient de poids) et de mettre les résultats en `temp`. Comme l'instruction est `vmull` avec deux `l`, elle élargit également la voie (**lane**) **de 8 bits à 16 bits**.

Un exemple peut être vu dans la figure suivante.

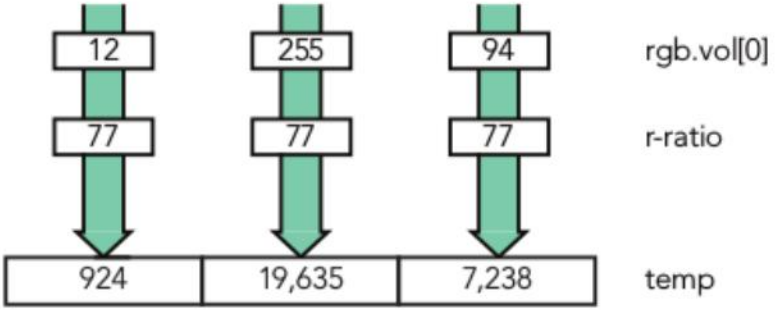


Figure 0.1 Multiplication des coefficients **RGB** (des **arguments 8 bits** aux **résultats 16 bits**)

La variable `temp` contient maintenant chaque composante rouge des 8 pixels, multipliée par le poids rouge. Vous pouvez faire la même chose avec les composantes **vert** et **bleu**: multipliez-les dans des registres séparés puis ajoutez les résultats.

Cependant, NEON a une solution plus élégante: multiplier et accumuler.

```
temp = vmlal_u8 (temp, rgb.val [1], g_ratio);
```

Vector Multiply and Accumulate Long (`vmlal`) est identique à **Vector Multiply**, sauf qu'il permet d'ajouter une valeur au résultat de la multiplication. Dans ce cas, `vmlal` peut multiplier la composante verte de chaque pixel par le rapport de poids vert, puis ajouter les valeurs existantes dans `temp` avant de réécrire les données dans `temp`.
Désormais, la variable `temp` contient les composantes rouges pondérés plus les composantes verts pondérées. Il ne reste plus qu'à faire la même action avec les composantes bleues.

```
temp = vmlal_u8 (temp, rgb.val [2], b_ratio);
```

Finalement, `temp` contient la **valeur pondérée de chaque composante** du pixel, multipliée par **256**.
La valeur 256 n'a pas été choisie au hasard. Elle a été choisie car il s'agit d'une **puissance de 2** et peut être **décalée** pour effectuer une **division rapide**. En outre, la plus grande valeur possible dans une valeur 8 bits est 256, et la plus grande valeur possible de toutes les valeurs pondérées multipliées par les composants de pixel est **65536**, la taille maximale d'une **valeur 16 bits**; il n'y aura donc jamais de perte de données, même pour les valeurs les plus élevées possibles.

Maintenant, chaque pixel pondéré doit être divisé par 256 par le décalage, fournissant les résultats en valeurs sur 8 bits. C'est un travail pour `vshrn`.

```
result = vshrn_n_u16(temp, 8);
```

Vector Shift Right Narrow (`vshrn`) est une instruction qui peut prendre un **registre à quatre mots**, effectuer une division par une puissance de 2, puis sortir les résultats dans un registre à deux mots, rétrécissant les voies (lanes). Vous devez maintenant réécrire les résultats en mémoire.

```
vst1_u8(dest, result);
```

Vous trouverez ci-dessous une **simple fonction C** qui effectue une boucle pour chaque 8 pixels d'une image et convertit automatiquement les pixels **RGB** en **niveaux de gris**.

L'ensemble de la routine C (NEON) ressemble à ceci:

```
void neon_greyscale(uint8_t * dest, uint8_t *src, int num)
{
int i;
uint8x8_t r_ratio=vdup_n_u8(77);
uint8x8_t g_ratio=vdup_n_u8(151);
uint8x8_t b_ratio=vdup_n_u8(28);
num/=8; //NEON will work on 8 pixels a time
for (i=0; i<num; i++)
        {
        uint16x8_t temp;
        uint8x8x3_t rgb = vld3_u8(src);
        uint8x8_t result;
        temp = vmull_u8(rgb.val[0], r_ratio);
        temp = vmlal_u8(temp,rgb.val[1], g_ratio);
        temp = vmlal_u8(temp,rgb.val[2], b_ratio);
        result = vshrn_n_u16(temp, 8);
        vst1_u8(dest, result);
        src += 8*3; // 3 x 8 pixels in RGB format
        dest += 8; // One single 8-bit value per pixel
        }
}
```

Voyons maintenant le code complet et comparons les performances des solutions **CPU** et **NEON** à ce programme.
Notez que nous utilisons les fonctions **openCV** pour charger/décompresser et afficher l'image résultante.

Les performances sont mesurées en millisecondes avec les primitives de synchronisation fournies par **openMP**.

openMP sera étudié plus en détail dans ce texte pour montrer comment construire des solutions de traitement **multicœur** à un certain nombre d'exemples.

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include "arm_neon.h"

using namespace cv;

#define uchar unsigned char
#define bvga 512*128*3     // 512*512
#define width 512
#define height 512

void greyscale(unsigned char *dest, unsigned char *src, int n)
{
  int i;
  int r,g,b;
  int y;
  for (i=0; i < n; i++)
  {
    b = *src++; // load blue
```

```
    g = *src++; // load green
    r = *src++; // load red
    // build weighted average:
    y = (r*77)+(g*151)+(b*28);
    // undo the scale by 256 and write to memory:
    *dest++ = (y>>8);
  }
}

void neon_greyscale(uint8_t * dest, uint8_t *src, int num)
{
int i;
uint8x8_t r_ratio=vdup_n_u8(77);
uint8x8_t g_ratio=vdup_n_u8(151);
uint8x8_t b_ratio=vdup_n_u8(28);
num/=8; //NEON will work on 8 pixels a time
for (i=0; i<num; i++)
        {
        uint16x8_t temp;
        uint8x8x3_t rgb = vld3_u8(src);
        uint8x8_t result;
        temp = vmull_u8(rgb.val[0], r_ratio);
        temp = vmlal_u8(temp,rgb.val[1], g_ratio);
        temp = vmlal_u8(temp,rgb.val[2], b_ratio);
        result = vshrn_n_u16(temp, 8);
        vst1_u8(dest, result);
        src += 8*3; // 3 x 8 pixels in RGB format
        dest += 8; // One single 8-bit value per pixel
        }
}

int main (int argc, char* argv[])
{
  double starttime,stoptime,extime,mips;
  int mode=0;
  Mat bw=Mat(512,512,CV_8UC1);
  const char* window_title = "Hello, NEON!";
  int index; int n=width*height;
  float mean=0.0, var=0.0, svar=0.0, std;
  if (argc<2)
  {
  fprintf (stderr, "usage: %s IMAGE\n", argv[0]); exit(1);}
  printf("Chose the execution mode CPU,NEON [0,1]\n");
  scanf("%d",&mode);
  Mat img = imread(argv[1], IMREAD_COLOR);
  printf("Matrix size: %ld\n",img.total());
  if(img.empty())
    {
        printf("No image %s\n",argv[1]);
        return 1;
    }
  starttime=omp_get_wtime();

  if(!mode) greyscale(bw.data,img.data,n);  // n=512*512
  else neon_greyscale(bw.data,img.data,n);

  stoptime=omp_get_wtime();
  extime=stoptime-starttime;
  printf("Total execution time: %3.6f ms\n",extime*1000);
  imshow("Display window", bw);
    int k = waitKey(0); // Wait for a keystroke in the window
```

```
    if(k == 's')
    {
        imwrite("clip512x512bw.jpg", bw);
    }

  return EXIT_SUCCESS;
}
```

L'exécution se traduit par un temps de traitement en millisecondes:

```
bako@xavier:~/MandM/lab0$ ./convGreyNEON lena512x512.jpg
Chose the execution mode CPU,NEON [0,1]
0
Matrix size: 262144
Total execution time: 5.559285 ms
bako@xavier:~/MandM/lab0$ ./convGreyNEON lena512x512.jpg
Chose the execution mode CPU,NEON [0,1]
1
Matrix size: 262144
Total execution time: 3.133368 ms
```

0.2 Programmation multicœur avec `openMP`

0.2.1 Mode d'opération de `openMP`

openMP implique de travailler avec des **threads**, qui sont essentiellement les **processus légers** qui partagent le **même espace d'adressage** mémoire avec les autres threads créés pour un seul programme.
Si un thread modifie une variable que tous les threads peuvent voir, le prochain accès à cette variable utilisera la nouvelle valeur.

Nous pouvons imaginer toutes nos variables dans un gros bloc de mémoire où tous les processeurs peuvent voir toutes les variables. Les **threads** voient tous cette mémoire et peuvent modifier cette mémoire.

Les threads peuvent tous effectuer des **opérations d'E/S**, créer des **fichiers**, imprimer, etc. Ainsi, des exemples de programmes aussi simples que HelloWorld traditionnels peuvent en fait être capables de fonctionner en parallèle; bien qu'en général, les opérations d'E/S aient tendance à devoir **sérialiser l'accès** aux ressources globeau du système (pointeurs de **fichiers**, **interfaces**, etc.).

En général, **un programme** contient les **sections** qui peuvent être exécutées en **séquence** ou en **parallèle**.
L'exécution des sections parallèles peut être mappée sur une architecture multicœur.
Les sections séquentielles sont exécutées par un seul noyau.

L'efficacité globale ou l'accélération de l'exécution dépend de la proportion de code parallèle. L'équation suivante (**loi d'Amdahl**) fournit le calcul de **l'accélération (speedup)**:

$$speedup = 1/(S+1/N*(1-S))$$

où **S** est la proportion de la partie d'exécution **série** et **N** est le **nombre de cœurs** d'exécution.

Par exemple, l'accélération d'une application avec 20% de code série exécuté sur 4 cœurs est égale à **1,95** :

$$1/(0,2+1/(4*0.8)) = 1/(0,2+1/3,2) = 1(0,2+0,3125)= 1/0,5125 = 1,95$$

0.2.2 Compilateur openMP

openMP fonctionne principalement via les **directives du compilateur**. Une directive du compilateur est un **commentaire** que le compilateur peut prendre en compte si les bibliothèques **openMP** sont fournies.

```
#pragma omp ...
{
...code...
}
```

où le `... code ...` est appelé la région parallèle. C'est la zone du code que vous souhaitez essayer d'exécuter en parallèle, si possible.

Lorsque le compilateur voit le début de la **région parallèle**, il crée un **pool de threads**. Lorsque le programme s'exécute, ces threads commencent à s'exécuter et sont contrôlés par les informations contenues dans les directives. Sans directives supplémentaires, nous avons simplement un «**tas**» de threads.
Une façon de visualiser cela est d'imaginer une boucle implicite autour de la région parallèle, où nous avons N itérations CPU/cœur de la boucle.
Ces **itérations se produisent toutes en même temps**, contrairement à une **boucle explicite** (plate).

Le nombre de cœurs est contrôlé par une **variable d'environnement**, `OMP_NUM_THREADS`. Si elle n'est pas défini, elle peut être **définie par défaut** sur 1, sur 2 ou sur le **nombre de cœurs** de votre CPU.
Nous pouvons **exporter** vers l'environnement shell le nombre de threads à utiliser via la commande suivante:

```
export OMP_NUM_THREADS = 8
```

Nous sommes maintenant prêts à paralléliser `helloMulticore.c`. Dans un premier temps, introduisons les directives explicites du compilateur et ne faisons rien d'autre.
Notez la présence de `#include <omp.h>` nécessaire pour fournir les prototypes des fonctions et constantes **openMP**.

La **commande de compilation** est :

```
gcc -o $1 $1.c -lm -fopenmp
```

0.2.2.1 Le premier code

```
#include "stdio.h"
#include <omp.h>

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    printf("hello multicore user!\n");
  }
  return(0);
}
```

Et voici le résultat:

```
bako@xavier:~/MandM/lab0$ ./helloMulticoreUser
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
```

0.2.2.2 Environnement openMP

Nous pouvons utiliser plusieurs appels de fonction **openMP** pour interroger et contrôler notre **environnement**. Les fonctions les plus fréquemment utilisées sont celles qui renvoient le **nombre de threads** en cours d'exécution et l'**ID de thread actuel**.

Le nouveau `HelloMulticore.c` inclut ces fonctions et ressemble maintenant à ceci:

```
#include "stdio.h"
#include <omp.h>

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    int NCPU,tid,NPR,NTHR;
    NCPU = omp_get_num_procs();  // get the number of available cores
    tid = omp_get_thread_num();  // get current thread ID
    NPR = omp_get_num_threads();  // get total number of threads
    NTHR = omp_get_max_threads(); // get number of threads requested
    if (tid == 0) {     // execute it in master thread
      printf("%i : NCPU\t= %i\n",tid,NCPU);
      printf("%i : NTHR\t= %i\n",tid,NTHR);
      printf("%i : NPR\t= %i\n",tid,NPR);
    }
    printf("%i : hello multicore user! I am thread %i out of %i\n",tid,tid,NPR);
  }
return(0);
}
```

Son exécution donne le résultat suivant :

```
bako@xavier:~/MandM/lab0$ ./helloMulticore
1 : hello multicore user! I am thread 1 out of 8
0 : NCPU    = 8
0 : NTHR    = 8
0 : NPR     = 8
0 : hello multicore user! I am thread 0 out of 8
3 : hello multicore user! I am thread 3 out of 8
7 : hello multicore user! I am thread 7 out of 8
5 : hello multicore user! I am thread 5 out of 8
4 : hello multicore user! I am thread 4 out of 8
2 : hello multicore user! I am thread 2 out of 8
6 : hello multicore user! I am thread 6 out of 8
```

Le premier nombre que nous voyons ici est le **numéro de thread** ou l'**ID de thread** (variable `tid` dans le programme). Notez que l'affichage ne sort pas nécessairement dans l'ordre des threads. Le numéro de **thread 0** est également appelé **thread principal**; ce fil est **toujours actif** et démarre au **début du programme**.

0.2.3 Programmation parallèle avec openMP

Dans les paragraphes suivants, nous présentons quelques techniques de base pour la programmation parallèle avec des threads **openMP**. Dans ce contexte, le plus important est la **parallélisation des boucles** et l'utilisation de **variables partagées et locales**.
Nous montrons comment **initialiser** les **variables locales à partir des variables partagées** et comment «**réduire**» les **variables locales en variables partagées**.

0.2.3.1 Boucles - variables locales et partagées

Considérez le pseudo-code suivant:

```
for(int x=0; x<width; x++)
{
  for(int y=0;y<height;y++) OutImage[x][y]=RenderPixel(x,y, &InImage);
}
```

Ce fragment de code parcourt simplement chaque pixel de l'écran et appelle une fonction, `RenderPixel`, pour déterminer la couleur finale de ce pixel. Notez que les résultats sont simplement stockés dans un tableau (`OutImage[x][y]`). L'image entière en cours de rendu est stockée dans une variable, `InImage`, dont l'adresse est transmise à la fonction `RenderPixel`.

Étant donné que chaque pixel est indépendant de tous les autres pixels et que la fonction `RenderPixel` devrait prendre un certain temps, ce petit extrait de code fournit une bonne parallélisation avec **openMP**.

Considérez la version modifiée suivante du code précédent:

```
int x;
#pragma omp parallel for
for(x=0; x<width; x++)   // shared variable x
{ // local-private variable y
  for(int y=0;y<height;y++)OutImage[x][y]=RenderPixel(x,y,&InImage);
}
```

Le seul changement apporté au code est la ligne directement au-dessus de la boucle for externe:

```
#pragma omp parallel for
```

Cette directive du compilateur indique au compilateur de paralléliser la boucle for avec openMP.
Si un utilisateur utilise une **CPU quadricœur**, l'accélération de ce fragment de programme devrait être de **2,5 à 3,0** avec l'ajout d'une seule ligne de code.

Jetons un coup d'œil sur la variable `y` dans le pseudo code ci-dessus. Comme la variable est effectivement déclarée **à l'intérieur de la région parallélisée**, chaque processeur aura une **valeur unique et privée** pour `y`.
Cependant, considérez l'exemple de **code bogué** suivant ci-dessous:

```
int x,y;   // shared variables  - problem !!!
#pragma omp parallel for
for(x=0; x<width; x++)
{
  for(y=0;y<height;y++) OutImage[x][y]=RenderPixel(x,y,&InImage);
}
```

Ici, les variables x et y sont déclarées **en dehors de la région parallélisée**. La portée par défaut des variables, `x`, `y`, `OutImage` et `InImage`, sont **toutes partagées par défaut**, ce qui signifie que ces valeurs seront les mêmes pour tous les threads. Tous les threads ont accès pour lire et écrire sur ces variables partagées.
Le code ci-dessus est bogué car **la variable y doit être différente pour chaque thread**.

Déclarer `y` à l'**intérieur** de la région parallélisée est un moyen de garantir qu'une variable sera **privée** pour chaque thread, mais il existe un autre moyen d'accomplir cela ci-dessous:

```
int x,y;
#pragma omp parallel for private(y)
for(x=0; x < width; x++)
{
  for(y=0;y<height;y++) OutImage[x][y]=RenderPixel(x,y,&InImage);
}
```

Au lieu de déclarer la variable `y` dans la région parallèle, nous pouvons la **déclarer en dehors** de la région parallèle et la déclarer **explicitement** comme une **variable privée dans la directive du compilateur openMP**.
Cela provoque effectivement que chaque thread ait une variable **indépendante** appelée `y`. Chaque thread n'aura accès qu'à **sa propre copie** de cette variable.

0.2.3.2 Ordonnancement de boucles

La ordonnancement de boucle est utilisé pour contrôler la manière dont les itérations de boucle sont distribuées sur les threads.
La syntaxe pour ordonnancer les opérations de boucle est basée sur deux paramètres:

1. le `type` d'ordonnancement et
2. le `chunk_size`.

Ces paramètres spécifient comment les itérations de la boucle sont affectées aux threads de l'ensemble.
La granularité de la distribution de la charge de travail est un **bloc**, un **sous-ensemble contigu** et **non vide** de l'espace d'itération.
Notez que le paramètre `chunk_size` n'a pas besoin d'être une constante; toute expression d'entier invariant de boucle avec une valeur positive est autorisée.
L'**ordonnancement** le plus simple est **statique**; il peut être utilisé en l'absence de clause d'**ordonnancement explicite**. Les autres modes d'ordonnancement sont **dynamiques** et **guidés**.

L'exemple suivant montre l'utilisation de la clause d'ordonnancement. La boucle externe a été parallélisée avec la construction de `loop`.
La charge de travail dans la boucle interne dépend de la valeur de la variable d'itération de boucle externe `i`.

```
int i,j,n;
#pragma omp parallel for default(none) schedule(static) private(i,j) shared(n)
for (i=0; i<n; i++)
{
printf("Iteration %d executed by thread %d\n", i, omp_get_thread_num());
for (j=0; j<i; j++)
system("sleep 1");
} /*-- End of parallel for --*/
```

Pour illustrer cet exemple, nous avons exécuté la boucle ci-dessus pour `n=400` en utilisant **quatre threads**.

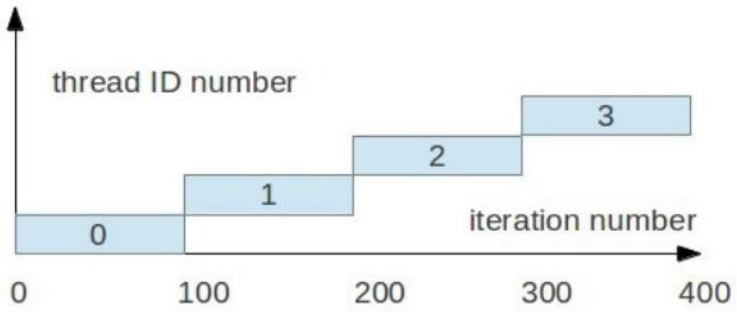


Figure 0.2 Ordonnancement statique dans la boucle `for`

0.2.4 Exemples d'application

Nous allons construir deux applications complètes orientées matrice. Le premier est l'algorithme du produit scalaire (`dotProduct`), le second est la **multiplication matricielle**.

0.2.4.1 Produit scalaire et la « réduction »

L'exemple openMP suivant est un programme qui calcule le **produit scalaire de deux vecteurs**, `a` et `b` (c'est-à-dire **somme de** (`a[i]*b[i]`), en utilisant une **réduction de somme**. Les variables d'entrée `a` et `b` sont des vecteurs partagées avec de valeurs **doubles (`double`)**.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000

int main (int argc, char *argv[]) {
  double a[N], b[N];
  double sum = 0.0;
  int i, n, tid;
  /* Start a number of threads */
#pragma omp parallel shared(a) shared(b) private(i)
  {
    tid = omp_get_thread_num();
    /* Only one of the threads do this */
#pragma omp single
    {
      n = omp_get_num_threads();
      printf("Number of threads = %d\n", n);
    }
    /* Initialize a and b */
#pragma omp for
    for (i=0; i < N; i++) {
      a[i] = 1.0;
      b[i] = 1.0;
    }
    /* Parallel for loop computing the sum of a[i]*b[i] */
#pragma omp for reduction(+:sum)
    for (i=0; i < N; i++) {
      sum += a[i]*b[i];
    }
  } /* End of parallel region */
  printf("   Sum = %2.1f\n",sum);
  exit(0);
}
```

Résultat affiché :

```
bako@xavier:~/MandM/lab0$ ./dotProduct
Number of threads = 8
   Sum = 1000.0
```

La clause de réduction spécifie deux choses:

1. Lorsque le contrôle entre dans la région parallèle, chaque thread de la région obtient une **copie privée** de de **`sum`**, initialisée à **`1.0`** pour **`+`**.
2. Lorsque le contrôle quitte la région parallèle, la somme d'origine est mise à jour en combinant sa valeur avec les valeurs finales des copies privées de thread, en utilisant +.
   Puisque **+** est **associatif**, la somme finale a la même valeur que pour l'exécution en série du code.

Notez que les **opérateurs de réduction doivent être associatifs**, comme **l'addition** ou la **multiplication**.

0.2.4.2 Multiplication matricielle et `firstprivate`

En général, avec plusieurs threads, vous n'obtenez **aucun avantage** de performances par rapport à l'**exécution en série**; **pire**, vous vous retrouvez généralement avec un **désavantage ou une pénalité de performance**.
C'est pourquoi nous essayons de **maximiser la partie exécution parallèle** du programme/algorithme.
La **multiplication matricielle** est l'un des meilleurs exemples utilisés pour illustrer les avantages (accélération) de l'exécution parallèle.
L'algorithme de base est la **boucle for triplement imbriquée**. La boucle de multiplication principale ressemble à ceci:

```
for(i=0;i<DIM;i++) {
  for(j=0;j<DIM;j++) {
    dot=0.0;
    for(k=0;k<DIM;k++)
    dot += a[i][k]*b[k][j];
    c[i][j]=dot;}
}
```

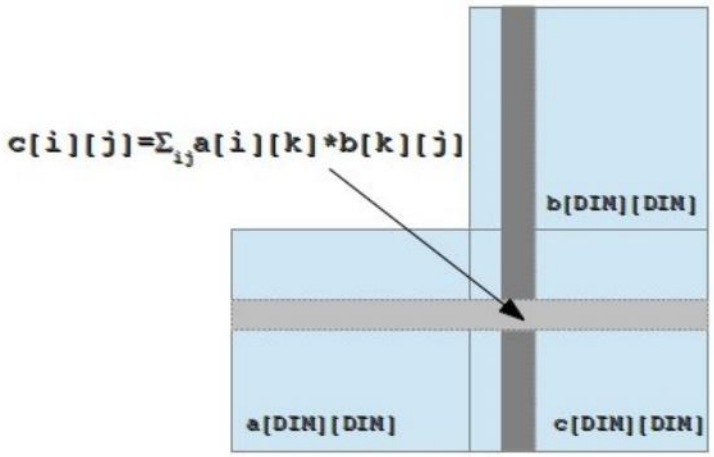


Figure 0.3 Multiplication matricielle – l'algorithme

On voit tout de suite que la boucle intérieure est une somme de réduction. Il s'agit essentiellement d'un produit scalaire entre deux vecteurs. La **boucle interne** est exécutée **N fois au carré**. Cela signifie que la **région parallèle** peut être établie **N fois au carré**.
La préparation des données et la récupération des résultats ne sont pas gratuits; c'est-à-dire qu'ils ont un coût en temps non nul.
Pour la plupart des efforts de parallélisation, nous souhaitons inclure le maximum de travail dans la région parallèle.
Nous essayons de mettre les directives de parallélisation en dehors de la boucle externe, donc nous insérons une simple directive parallèle **`#pragma`** comme suit:

```
int DIM=512;
```

```
#pragma omp parallel for private(i,j,k,dot) shared(a,b,c)
for(i=0;i<DIM;i++) {
  for(j=0;j<DIM;j++) {
    dot=0.0;
    for(k=0;k<DIM;k++)
    dot += a[i][k]*b[k][j];
    c[i][j]=dot; }
}
```

Dans cet exemple, **`private(i,j,k,dot)`** indique au compilateur quelles variables sont **privées** par rapport à chaque thread, et **`shared(a, b, c)`** indique celles qui sont **partagées** entre les threads.

Avec un ajustement de code relativement simple, nous pouvons obtenir des performances nettement meilleures en parallèle. Ce que nous faisons, c'est augmenter la quantité de travail effectué en parallèle. Nous faisons cela **en déroulant une boucle**.

Notre code ressemble maintenant à ceci:

```
#pragma omp parallel for private(i,j,k,dot) shared(a,b,c) firstprivate(DIM)
for(i=0;i<DIM;i+=4) {
  for(j=0;j<DIM;j++) {
    dot[0]=dot[1]=dot[2]=dot[3]=0.0;
    for(k=0;k<DIM;k++) {
      dot[0] += a[i+0][k]*b[k][j];
      dot[1] += a[i+1][k]*b[k][j];
      dot[2] += a[i+2][k]*b[k][j];
      dot[3] += a[i+3][k]*b[k][j];  }
    c[i+0][j]=dot[0];
    c[i+1][j]=dot[1];
    c[i+2][j]=dot[2];
    c[i+3][j]=dot[3];}
}
```

Cette version du programme fonctionne sur **quatre lignes à la fois** (nous avons déroulé la boucle), augmentant ainsi la quantité de travail effectuée par itération.
Nous avons également réduit la pénalité d'**échec du cache** par itération en **réutilisant** certains des éléments les plus coûteux en espace mémoire (le **`b [k] [j]`**). De plus, nous avons ajouté la directive **openMP** **`firstprivate(DIM)`** qui spécifie que chaque thread doit avoir sa **propre instance d'une variable**, et que la **variable doit être initialisée avec la valeur de la variable telle qu'elle existe avant la construction parallèle**.

Bien entendu, après toutes ces modifications, nous vérifions les résultats pour nous assurer que l'ajout de parallélisation n'a pas également provoqué l'ajout de bogues!

```
bako@xavier:~/MandM/lab0$ ./matrixMultiplication.1
Starting matrix multiplication with 4 threads
Size of matrix: 512x512
Initializing matrices...
Time for parallel matrix multiplication: 0.60 s
Time for sequential matrix multiplication: 2.40 s
Done.
```

0.2.5 Traitement d'image à l'aide des sections `omp` et des fonctions `openCV`

Dans ce paragraphe, nous allons programmer quelques exemples simples de traitement d'image. Cela nécessite l'installation de l'API openCV fournissant les fonctions essentielles pour charger et stocker les images.
Attention. La compilation d'un programme en C pour openCV et openMP nécessite une command/script suivant :

```
g++ -o $1 $1.c  `pkg-config --cflags opencv4` `pkg-config --libs opencv4`  -lm
-fopenmp -lstdc++ -lopencv_core -lopencv_highgui
```

0.2.5.1 Travailler avec des sections

L'exemple suivant montre comment nous pouvons accélérer le traitement d'image simple avec plusieurs cœurs de traitement. Le programme utilise le mécanisme de **`section`** qui permet l'exécution parallèle de différentes fonctions.
Dans ce cas, chaque fonction traite une partie de l'image. L'image en format VGA est divisée en **4 blocs horizontaux avec différente intensité**. L'image elle-même est lue et affichée par les fonctions **openCV**.

Le code du programme est le suivant:

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
using namespace cv;
#define uchar unsigned char
#define bvga 640*120*3

void negimage0 (uchar *array) // function for section 0
{ int i;
for(i=0;i<bvga;i++)  array[i] = 255 - array[i];  // byte value complement
}
void negimage1 (uchar *array) // function for section 1
{ int i;
for(i=0;i<bvga;i++)  array[i] =0.8*(255 - array[i]);  // byte value complement
}
void negimage2 (uchar *array) // function for section 2
{ int i;
for(i=0;i<bvga;i++)  array[i] = 0.6*(255 - array[i]);  // byte value complement
}
void negimage3 (uchar *array) // function for section 3
{ int i;
for(i=0;i<bvga;i++)  array[i] = 0.4*(255 - array[i]);  // byte value complement
}

int main (int argc, char* argv[])
{
  double starttime,stoptime,extime,mips;
  uchar *data;
  const char* window_title = "Hello, OpenCV!";
  char sp; int par=0;
  if (argc < 2)
  {
    fprintf (stderr, "usage: %s IMAGE\n", argv[0]);
    return EXIT_FAILURE;
  }
    Mat img = imread(argv[1], IMREAD_COLOR);
    printf("Matrix size: %ld\n",img.total());
    if(img.empty())
```

```
    {
        printf("No image %s\n",argv[1]);
        return 1;
    }
omp_set_num_threads(4);
starttime=omp_get_wtime();

  #pragma omp parallel   // try the execution without this line !!!
  #pragma omp sections
  {
    #pragma omp section
    {
    printf("%d\n",omp_get_thread_num());
    negimage0(img.data);  // section call
    }
    #pragma omp section
    {
    printf("%d\n",omp_get_thread_num());
    negimage1(img.data+bvga);  // section call
    }
    #pragma omp section
    {
    printf("%d\n",omp_get_thread_num());
    negimage2(img.data+2*bvga);  // section call
    }
    #pragma omp section
    {
    printf("%d\n",omp_get_thread_num());
    negimage3(img.data+3*bvga);  // section call
    }
  }

stoptime=omp_get_wtime();
extime= stoptime-starttime;
printf("Total execution time: %3.6f ms\n",extime*1000);
imshow("Display window", img);
    int k = waitKey(0); // Wait for a keystroke in the window
    if(k == 's')
    {
        imwrite("clipVGAneg.jpg", img);
    }

  return EXIT_SUCCESS;
}
```

Quelques exercices à faire avec et sans `#pragma omp parallel`

1. Analysez et testez le code ci-dessus en utilisant une exécution parallèle avec #pragma omp une exécution parallèle et séquentielle sans ce pragma.
2. Modifiez le programme pour n'utiliser qu'une seule fonction (negimage ()) partagée entre les sections et analysez les performances. Pourquoi le temps d'exécution est-il largement différent?
3. Réécrivez les sections de programme afin de traiter en parallèle 4 blocs verticaux.

Laboratoire 1 – programmation de base avec CUDA

Dans ce premier laboratoire nous allons écrire quelques simples exemple en code CUDA qui combine le code **C/C++** pour la partie **CPU** avec les **extensions de C** pour la partie à exécuter sur la **GPU**.

1.0 Introduction

L'arrivée de cartes graphiques adaptées à la programmation générale nous a permis d'utiliser des architectures many-core pour l'exécution massivement parallèle de programmes et d'applications. De plus, la disponibilité des CPU-GPU intégrées basées sur des architectures ARM, telles que Tegra K1/X1/X2/**Xavier** et plus récemment **Nano** et Nano NX, nous fournit les plateformes embarquées pour GPGPU (General Programming on GPU).
L'architecture de **Jetson Nano** est illustrée ci-dessous.

	JETSON NANO
GPU	128-core NVIDIA Maxwell 0.5 TFLOPS (FP16)
CPU	4-core ARM A57
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	16 GB eMMC
Encode	4K @ 30 (H.265)
Decode	4K @ 60 (H.265)
Camera	12 (3x4 or 4x2) MIPI CSI-2 D-PHY 1.1 lanes (18 Gbps)

Figure 1.1. L'architecture de la carte Jetson Nano

En terme de spécifications techniques, la carte est équipée d'un GPU architecture NVIDIA Maxwell™ avec **128 cœurs** NVIDIA **CUDA**® et un CPU **quad-core ARM**® Cortex®-A57 MPCore. La carte a une mémoire vive de **4Go** 64-bit LPDDR4.
Il faut ajouter une **microSD** pour avoir une mémoire de stockage. Pour cela, il est nécessaire de flasher une image sur une microSD UHS-1 (non fourni). Considérez donc que la Nano n'est pas équipée d'une mémoire dédiée – la carte SD est tout ce que vous aurez.
Nvidia qui en recommande une de **32Go**..

Jetson Nano est équipé de nombreux connecteurs notamment 4 ports USB (1x 3.0 et 3x 2.0), HDMI, Ethernet ou encore des connecteurs de caméra MIPI CSI-2 (compatible avec le module de caméra Raspberry Pi **v2**).
Il existe plusieurs façons d'alimenter la carte, soit par le connecteur **coaxial 2.1mm 5V 4A** (**J25** sur le schéma ci-dessous) ou par le port microUSB 5V 2A (J28). On change de configuration à l'aide du pont de court-circuit (J48), la présence de celui-ci pour l'utilisation du connecteur axial et l'absence pour le microUSB.

Par défaut l'état de consommation d'énergie est en `mode 0` c'est-à-dire **10W** uniquement pour la carte en elle-même. Dès qu'un périphérique est branché, par exemple sur un port USB, la carte ne fonctionne plus.
Une solution est donc de changer l'état de consommation en **mode 1** (**5W**) avec la commande suivante mais il y a une **perte de performance**.

Pour obtenir les performances optimales il faut utiliser un adaptateur qui fourni 4A par le connecteur **coaxial 2.1mm.**
Jetson Nano est livré avec la stack Nvidia via le **JetPack SDK** qui contient un OS (**Ubuntu 18.04**) , **TensorRT**, **CUDA** et autres fonctions. Cette carte est également compatible avec des framework de Deep Learning open source tel que **Tensorflow**, **PyTorch**, **Caffe** et **mxnet**.

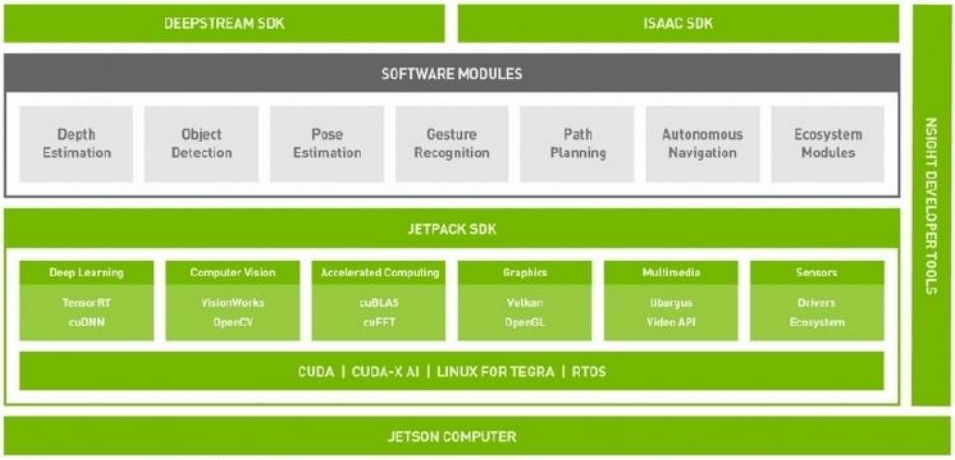


Figure 1.2 L'architecture logicielle de la carte Jetson Nano

1.0.1 Setup et lancement

Il existe deux modes d'utilisation du Jetson Nano : le mode **head less** et le mode **standalone**.
Le mode **headless** est l'utilisation **sans écran**, clavier et souris **depuis une machine hôte**. La carte est alimentée par le câble coaxial et branchée à la machine hôte par le port microUSB. Attention à ne pas oublier de placer le pont J18. Il faut ensuite connecter la carte par USB à la machine hôte.

On peut alors accéder à un **Jupyter Lab** via l'adresse 192.168.55.1:8888. En cas d'échec, il peut être nécessaire de vérifier la configuration de sa machine hôte

Le mode indépendant (**standalone**) correspond quant à lui à une utilisation normal avec clavier, écran et souris. Un utilisateur **`dlinano`** est configuré par défaut avec pour mot de passe **`dlinano`** (attention le clavier est en **qwerty**!

Et ce à chaque démarrage si la configuration du clavier n'est pas modifié). On accède alors une **interface graphique Ubuntu classique**.
Pour information, l'installation de certaines librairies python peuvent prendre beaucoup de temps car les images basées sur la distribution linux de Nvidia et l'architecture du processeur de la carte ne sont pas nécessairement disponibles. C'est pourquoi, lors des installations en utilisant python *pip*, ceux-ci sont compilés à partir des fichiers sources.

Utilisation de la caméra Raspberry Pi v2
Rien de plus simple, il suffit de la brancher sur le connecteur de caméra (les broches vers l'intérieur de la carte et la partie bleue vers l'extérieur) puis de lancer le notebook **`csi_camera.ipynb`** embarqué avec la distribution (dossier **`nvdli-nano`**).

1.0.2 Les moyens de traitement massivement parallèle

Les microprocesseurs basés sur une unité centrale mono ou multicœur, telle que la famille x86 (Intel, AMD) améliorent leurs performances de traitement grâce à l'introduction de mécanismes architecturaux complexes tels que pipelines, mémoires cache, unités d'exécution spécialisées, unités de traitement multicœur et en augmentant la fréquence de fonctionnement. À un certain stade, le coût ou le nombre de portes logiques prévues pour des mécanismes supplémentaires devient prohibitif.

L'augmentation de la fréquence de fonctionnement (***f***) implique l'augmentation de la consommation d'énergie proportionnelle au carré de la fréquence (***f*f***).

De nos jours, les applications nécessitant des performances de traitement élevées sont exécutées sur des grappes de microprocesseurs multicœurs. Les architectures avec des centaines ou des milliers de cœurs de microprocesseurs sont considérées comme des supercalculateurs, leurs prix dépassant plusieurs millions de dollars.
Le développement des unités de traitement graphique (GPU) suit une trajectoire différente. Dès le début, les circuits graphiques étaient composés de plusieurs cœurs avec des dizaines ou des centaines d'unités de traitement simples. Ils étaient spécialisés dans le calcul d'objets graphiques (images, vidéo). Progressivement, ces mini-unités de traitement sont devenues plus flexibles et plus programmables. Les fonctionnalités

supplémentaires, y compris les circuits de contôle et les fonctions d'ordonnancement des ressources, ont ouvert les GPUs pour la programmation générale.
Les systèmes mobiles et embarqués modernes contiennent les SoC basés sur des cœurs de processeurs ARM. Ces SoC intègrent différents types d'unités de traitement de médias, y compris les GPU.

1.0.3 Les caractéristiques d'un GPU compatible CUDA

Les processeurs GPU intègrent un grand nombre d'unités de traitement. Ces unités appelées cœurs CUDA sont organisées en entités appelées **Streaming Multiprocessor** - **SMX**. Un **SMX** peut contenir plusieurs cœurs, et un circuit GPU peut contenir plusieurs SMX. Dans le cas de Tegra Nano, illustré sur Figure 2, nous avons un SMX avec 128 cœurs. Chaque cœur est capable de traiter des données entières et à virgule flottante en simple et double précision. Les cœurs fonctionnent directement sur des blocs de registres et de la mémoire partagée intégrée à SMX. La mémoire locale est soutenue par un cache L1. Les mémoires de constante et de texture sont utilisées pour conserver les données partagées entre les threads. Pour les ensembles de données plus importants, la mémoire globale intégrée doit être utilisée.

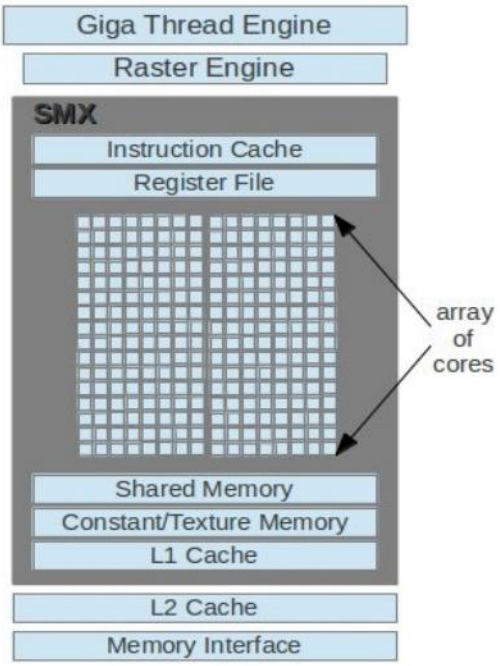


Figure 1.3 L'architecture d'un **SM** (**Streaming Multiprocessor**) de Tegra Nano

L'exécution de programmes écrits en langage C étendu est réalisée en projetant la séquence d'opérations sur plusieurs **threads**. Les **threads** sont ensuite organisées en blocs. Un ensemble de blocs avec leurs **threads** forme une grille. La figure ci-dessous montre une grille bidimensionnelle avec (3 colonnes, 2 rangées) de 6 blocs tridimensionnels représentés par des **threads** en format **4 * 4 * 4**.

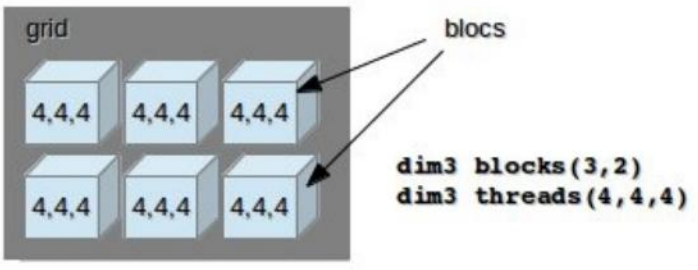


Figure 1.4 Une grille bi-dimensionnelle de CUDA

Les cœurs et le **SMX** communiquent avec la mémoire. Une mémoire globale est disponible pour tous les multiprocesseurs de streaming ainsi que pour la CPU dans la machine hôte. Les mémoires partagées sont disponibles pour les groupes de threads s'exécutant dans le même bloc. La mémoire **constante** et la mémoire de **texture** sont disponibles pour tous les threads globalement en mode **lecture seule**. En plus de ces mémoires, chaque thread a son propre ensemble de registres.

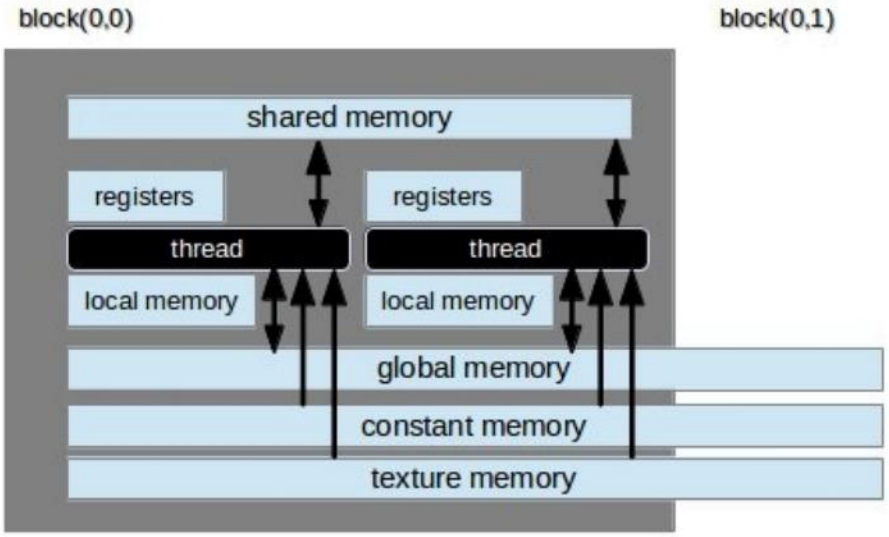


Figure 1.5 Un **bloc CUDA** avec les différentes types de mémoire

Avec la présence de plusieurs contrôleurs de mémoire et de bus très larges (256-512 bits), la bande passante de communication entre la mémoire globale et les cœurs GPU est très élevée. Pour le cycle d'horloge de 1,6 GHz et la largeur de bus de 512 bits (64 octets), le débit de données est égal à **102,4 Go/s**. Ce débit est presque 20 fois supérieur au débit de communication entre une CPU et sa mémoire externe. Néanmoins, l'utilisation efficace de tous les cœurs disponibles peut nécessiter un débit de données beaucoup plus élevé, environ 1 To/s. Dans ce cas, il est nécessaire d'exploiter pleinement les registres locaux et la mémoire partagée dans chaque bloc d'exécution.

1.0.4 L'élaboration d'un programme CUDA

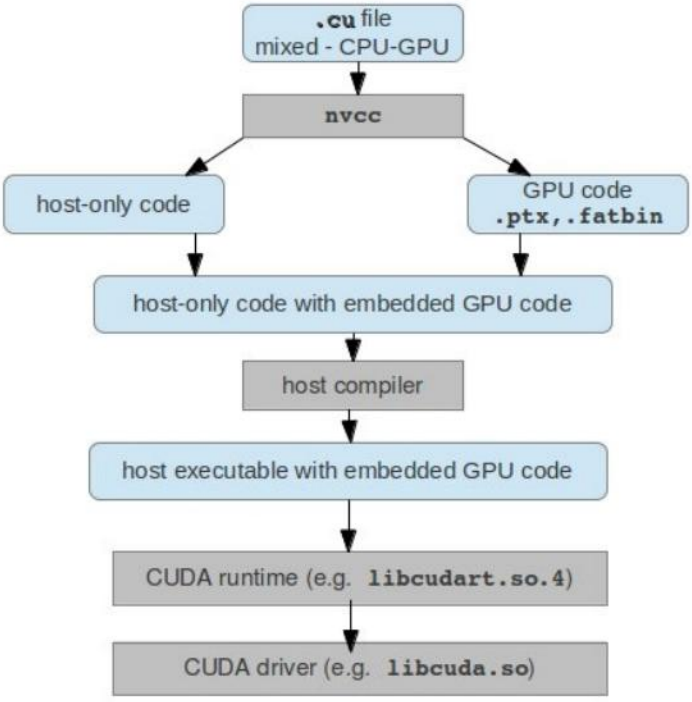


Figure 1.6 L'élaboration d'un programme CUDA avec le compilateur et l'exécution sur l'API de *run-time*

1.0.4.1 `nvcc` et `PTX`

Le code de type **`PTX`** (Parallel Thread eXecution) est la représentation intermédiaire du code compilé pour la GPU qui peut être compilé plus loin en **microcode GPU natif**. C'est le mécanisme qui permet aux applications CUDA d'être «à l'épreuve du futur» contre les innovations des **jeux d'instructions de NVIDIA**. Tant que le **`PTX`** pour un noyau CUDA donné est disponible, le pilote CUDA peut le traduire en microcode pour n'importe quel GPU sur lequel l'application s'exécute (même si le GPU n'était pas disponible lors de l'écriture du code).

1.0.5 L'analyse de `device` (GPU)

Avant de commencer le travail analysons notre **device** :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
cudaDeviceProp dP;
int device=0;
cudaGetDeviceProperties (&dP , device) ;
printf("Name:%s\n",dP.name);
printf("Memory total:%lu MB\n",dP.totalGlobalMem/(1024*1024));
printf("Shared memory per block:%d in B\n",(int)dP.sharedMemPerBlock);
printf("MaxThreads block:%d\n",(int)dP.maxThreadsPerBlock);
printf("warpSize:%d\n",(int)dP.warpSize);
printf("major:%d\n",(int)dP.major);
printf("minor:%d\n",(int)dP.minor);
printf("number of SM:%d\n",(int)dP.multiProcessorCount);
printf("Clock frequency:%1.3f inGHz\n",(int)dP.clockRate/1000000.0);
return 0;
}
```

Effectuer la compilation par le simple script (**`comp.sh`**)

```
nvcc -o $1 $1.cu -lm
```

`nvcc` est un script **nvidia** pour lancer la **compilation mixte** CPU/GPU de votre code (**`code.cu`**).

L'exécution du code ci-dessus sur une carte **Nvidia Jetson Xavier**.

```
bako@xavier:~/MandM/lab1$ ./deviceStat
Name:Xavier
Memory total:15822 MB
Shared memory per block:49152 in B
MaxThreads block:1024
warpSize:32
major:7
minor:2
number of SM:8
Clock frequency:1.377 inGHz
```

1.1. Programmation CUDA de base

La carte Jetson Nano/Xavier est fournie avec le système Ubuntu 18.04 LTS. Le **toolkit** CUDA est déjà intégré dans **JetPack**. Nous sommes prêts à utiliser le **script** du compilateur **`nvcc`**.

Les programmes CUDA consistent essentiellement en code C pur pour l'exécution sur CPU et en code **C étendu** pour l'exécution sur **GPU**. Dans ce contexte, trois types de fonctions sont définis:

`__host__` fonctionnant uniquement sur le processeur (facultative)
`__global__` fonctionnant sur le GPU, appelé par le CPU
`__device__` fonctionnant sur le GPU, appelé par le GPU

Note: La fonction marquée par le préfixe **`__global__`** est aussi appelée **noyau** (**kernel**)

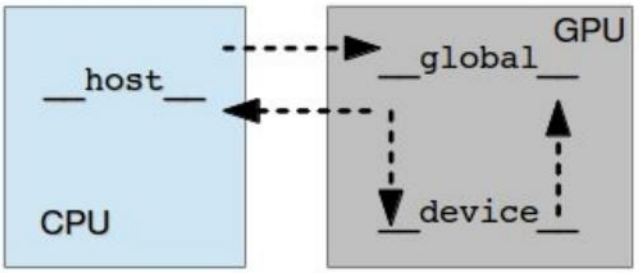


Figure 1.7 Les appels des fonctions **CUDA** : **`__host__`**, **`__global__`**, **`__device__`**

1.1.1 Blocs et threads

L'appel d'une fonction globale est organisé autour de l'ensemble des **threads** et des **blocs** à activer. Ceci est défini par une entrée du type:

```
noyau<<< blocks,threads >>>(arguments)
```

Les paramètres **`blocks`** et **`threads`** doivent être définis avant d'appeler le noyau.
L'exemple le plus simple est:

```
noyau<<<1,10>>>(arguments);
```

Dans ce cas, notre noyau fonctionne dans un seul bloc avec 10 threads.
Un autre exemple est:

```
noyau <<< 2, 5 >>> (arguments);
```

Dans ce cas, notre noyau fonctionne sur 2 blocs de 5 threads.

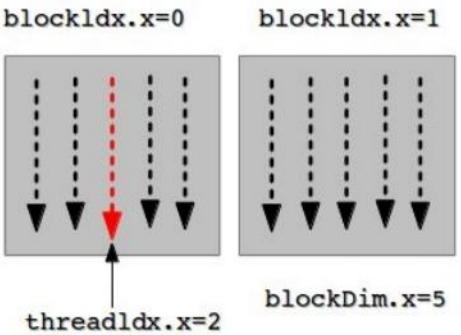


Figure 1.8 Les blocs et threads de CUDA avec leurs paramètres automatiques

Dans de nombreux applications, nous devons exploiter les données organisées en tableaux multidimensionnels. Pour affecter une organisation, nous utilisons des structures de type **`dim3`**. Une structure de type **`dim3`** contient 3 éléments identifiant chacun une dimension (**`x`**, **`y`** et **`z`**).

Par exemple pour la **structure d'un bloc** organisé autour des threads:

```
dim3 threads = (4,4,2);
```

définit l'organisation des threads dans un bloc sur une matrice tridimensionnelle **`4*4*2`**.

La même technique est utilisée pour définir la **structure d'une grille**.
Par exemple:

```
dim3 blocks = (3,3,1);
```

définit l'organisation des blocs dans une matrice de grille de **`3*3.`**

Un appel du noyau avec cette configuration inclut les paramètres des threads et des blocs:

```
noyau<<<blocks,threLaboratoire 1 – programmation « algorithmique » avec CUDAads>>>(arguments);
```

1.1.2 Structure interne d'un *kernel*

A l'intérieur du noyau, nous pouvons obtenir les paramètres de l'organisation à travers les **variables automatiques** associées à chaque thread.

Ces variables sont: **`threadIdx`**, **`blockIdx`**, **`blockDim`**, **`gridDim`**.
Pour une organisation simple et unidimensionnelle, nous traitons l'index **`x`** fourni pour les threads et les blocs: **`threadIdx.x`**, **`blockIdx.x`**, **`blockDim.x`** et **`gridDim.x`**.

Ce qui suit est notre **premier exemple complet**, une addition de deux vecteurs à virgule flottante.

1.2 Premier exemple complet - addition des vecteurs

1.2.1 Addition avec une grille linéaire

```
// on the GPU side :
__global__ void addVect(float* in1, float* in2, float* out)
{
int  i =  threadIdx.x  + blockIdx.x* blockDim.x;
out[i] = in1[i] + in2[i];
}
// on the CPU side :
#include <stdio.h>
int main()
{
float v1[]={1,2,3,4,5,6,7,8,9,10};
float v2[]={1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9};
int memsize = sizeof(v1);
int vsize = memsize/sizeof(float);
float res[vsize];
float* Cv1; cudaMalloc((void **)&Cv1,memsize);
float* Cv2; cudaMalloc((void **)&Cv2,memsize);
float* Cres; cudaMalloc((void **)&Cres,memsize);
cudaMemcpy(Cv1,v1,memsize,cudaMemcpyHostToDevice);
cudaMemcpy(Cv2,v2,memsize,cudaMemcpyHostToDevice);
addVect<<<2,vsize/2>>>(Cv1,Cv2,Cres);  // 2 blocks
cudaMemcpy(res,Cres,memsize,cudaMemcpyDeviceToHost);
int i=0;
printf("res= { ");
for(i=0;i<vsize;i++)
  { printf("%2.2f ", res[i]); }
printf("}\n");
}
```

Ce programme consiste en plusieurs parties:

1. initialisation et allocation de la mémoire (statique) sur CPU (*host*) :
   ```
   float v1[]={1,2,3,4,5,6,7,8,9,10};
   float v2[]={1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9};
   int memsize = sizeof(v1);
   int vsize = memsize/sizeof(float);
   float res[vsize];
   ```

2. allocation de la mémoire sur GPU (device)
   ```
   float* Cv1; cudaMalloc((void **)&Cv1,memsize);
   float* Cv2; cudaMalloc((void **)&Cv2,memsize);
   float* Cres; cudaMalloc((void **)&Cres,memsize);
   ```

3. copie des données vers la mémoire GPU
   ```
   cudaMemcpy(Cv1,v1,memsize,cudaMemcpyHostToDevice);
   cudaMemcpy(Cv2,v2,memsize,cudaMemcpyHostToDevice);
   ```

4. appel du *kernel*
   ```
   addVect<<<2,vsize/2>>>(Cv1,Cv2,Cres);  // 2 blocks
   ```
   Notez que le noyau génère un index simple pour chaque thread dans le bloc, puis il exécute les threads en parallèle.

5. copie du résultat dans la mémoire CPU
   ```
   cudaMemcpy(res,Cres,memsize,cudaMemcpyDeviceToHost);
   ```

6. affichage du résultat
   ```
   int i=0;
   ```

```
printf("res= { ");
for(i=0;i<vsize;i++)
  { printf("%2.2f ", res[i]); }
printf("}\n");
}
```

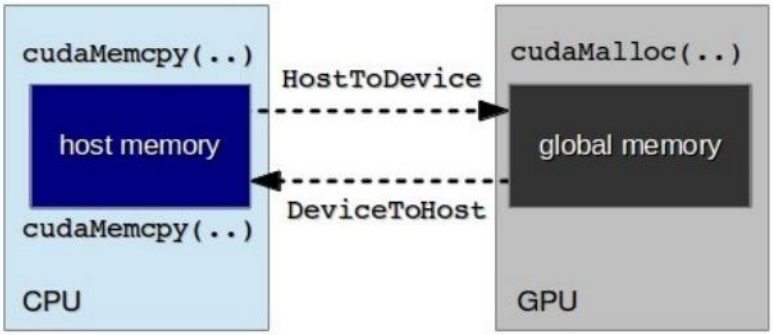


Figure 1.9 Transfert des données entre la mémoire de la **CPU** et de la **GPU**

A faire:

1. Analyser le programme ci-dessus
2. Effectuer la compilation par le simple script (**comp.sh**)
   nvcc -o $1 $1.cu -lm

1.2.2 Addition des vecteurs : mécanisme `zero-copy`

Le mécanisme de copie zéro est disponible sur toutes les cartes Jetson. La carte Jetson dispose de 4 Go de mémoire physique partagée par le processeur ARM et le GPU CUDA.
Si une opération de type **`cudaMemcpy Host->Device`** est fait sur la Jetson, la donnée est simplement copiée à un nouvel emplacement sur la même mémoire physique et en récupérant un pointeur CUDA.
Dans un tel scénario, l'accès **Zero Copy** est solution parfaite.

L'extrait de code suivant montre comment utiliser le mécanisme de *zero-copy:*

```
// Set flag to enable zero copy access
cudaSetDeviceFlags(cudaDeviceMapHost);
// Host Arrays
float* h_in  = NULL;
float* h_out = NULL;
// Allocate host memory using CUDA allocation calls
cudaHostAlloc((void **)&h_in,  sizeIn,  cudaHostAllocMapped);
cudaHostAlloc((void **)&h_out, sizeOut, cudaHostAllocMapped);
// Device arrays
float *d_out, *d_in;
// Get device pointer from host memory. No allocation or memcpy
cudaHostGetDevicePointer((void **)&d_in,  (void *) h_in , 0);
cudaHostGetDevicePointer((void **)&d_out, (void *) h_out, 0);
// Launch the GPU kernel
kernel<<<blocks, threads>>>(d_out, d_in);
// No need to copy d_out back
```

1.2.2 Addition des vecteurs avec une grille bi-bi-dimensionnelle

Ecrire une nouvelle version avec une architecture d'exécution de blocs et de la grille: **2x2** et **2x2** et les vecteurs d'entrée avec 16 éléments.

Utilisez :
```
dim3 block(2,2);  dim3 grid(2,2);
```

Préparez le nouveau **kernel** avec les **variables automatiques** :

```
threadIdx.x, blockIdx.x, blockDim.x, gridDim.x
threadIdx.y, blockIdx.y, blockDim.y, gridDim.y

// on the GPU side :
__global__ void addVect(float* in1, float* in2, float* out)
{
int ix =  threadIdx.x  + blockIdx.x* blockDim.x ;
int iy =  threadIdx.y  + blockIdx.y* blockDim.y ;
int i = ix + iy*blockDim.x*gridDim.x;
out[i] = in1[i] + in2[i];
}
// on the CPU side :
#include <stdio.h>
int main()
{
float v1[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
float v2[]={1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,1.0,2.1,2.2,2.3,2.4,2.5};
int memsize = sizeof(v1);
int vsize = memsize/sizeof(float);
float res[vsize];
dim3 grid(2,2);
dim3 block(2,2);
float* Cv1; cudaMalloc((void **)&Cv1,memsize);
float* Cv2; cudaMalloc((void **)&Cv2,memsize);
float* Cres; cudaMalloc((void **)&Cres,memsize);
cudaMemcpy(Cv1,v1,memsize,cudaMemcpyHostToDevice);
cudaMemcpy(Cv2,v2,memsize,cudaMemcpyHostToDevice);
addVect<<<grid,block>>>(Cv1,Cv2,Cres);  // 2 blocks
cudaMemcpy(res,Cres,memsize,cudaMemcpyDeviceToHost);
int i=0;
printf("res= { ");
for(i=0;i<vsize;i++)
  { printf("%2.2f ", res[i]); }
printf("}\n");
}
```

A faire :

1. Analyser le programme ci-dessus
2. Effectuer la compilation par le simple script (**comp.sh**)
   ```
   nvcc -o $1 $1.cu -lm
   ```
3. Modifier la taille des vecteurs à 128/256 en les initialisant par une fonction interne.

1.3 Produit matriciel et l'évaluation des performances

Dans cette partie du Laboratoire 1, nous allons étudier quelques exemples de programmation nous permettant de souligner les performances de la programmation parallèle avec CUDA. Nous commençons par le calcul matriciel – produit matriciel et le programme **`matrixMult.cu`** .

1.3.1 Produit matriciel

La multiplication matricielle nous permet d'effectuer simultanément la même opération de multiplication et d'addition sur tous les éléments de deux matrices. Une telle organisation du traitement est optimale pour l'architecture CUDA où chaque coeur peut exécuter une multiplication et une addition simultanément.
Le principe de la multiplication matricielle est illustré dans la figure suivante.

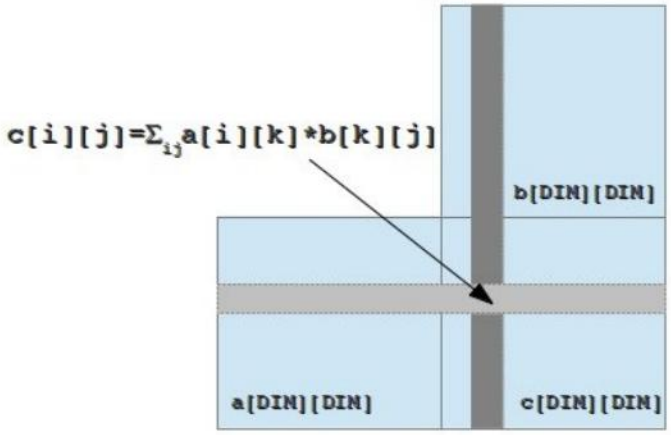


Figure1.10. Multiplication matricielle - calcul d'un élément
On remarque que pour la valeur **`DIM=512`**, il faut exécuter **`512*512*512`** multiplications et additions.
La version séquentielle de la fonction ci-dessus exécutée sur la CPU ressemble à ceci:

```
void matrix_mul_cpu(float* bA, float* bB, float* bC)
{
    for(int i=0;i<DIM;i++)
      for(int j=0;j<DIM;j++)
        for(int k=0;k<DIM;k++)
          bC[j + i*DIM] += bA[k+j*DIM]*bB[j+k*DIM];
}
```

Voici le code du noyau; il utilise une structure (grille) bidimensionnelle sur **X** et **Y** avec des blocs également définis en deux dimensions.:

```
__global__ void
  matrix_mul_gpu(float* dev_A,float* dev_B,float* dev_C,int Width)
  {
  // 2D thread ID
  int tx = threadIdx.x + blockDim.x*blockIdx.x;
  int ty = threadIdx.y + blockDim.y*blockIdx.y;
  float Pvalue =0;
  for(int k=0;k<Width;++k)
  {
    float Ael=dev_A[ty*Width + k];
    float Bel=dev_B[k*Width + tx];
    Pvalue += Ael*Bel;
  }
  dev_C[ty*Width+tx]=Pvalue;
}
```

Notez que l'utilisation du noyau GPU réduit le nombre d'opérations séquentielles à **512**. Les éléments de deux boucles supérieures sont pris en charge indépendamment par les threads qui peuvent fonctionner en parallèle.

1.3.2 Evaluation des performances

L'évaluation des performances nécessite la mesure du temps d'exécution de la tâche donnée. Avec l'API CUDA, ces mesures sont prises en charge par le mécanisme d'événement lié au temporisateur GPU. La séquence suivante initialise et démarre le temporisateur GPU.

```
float  eT;  // short for  elapsedTime
cudaEvent_t start, stop;
cudaEventCreate (&start);
cudaEventCreate (&stop);
cudaEventRecord (start, 0);
// here we call the GPU kernel or CPU function
cudaEventRecord (stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime (&eT, start, stop);
```

Le résultat de cette évaluation peut être affiché comme suit:

```
printf ("GPU time to multiply %d*%d:%3.2fms \n",Width,Width,eT);
```

1.3.3 Produit matriciel – code complet

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/unistd.h>
#include <sys/ioctl.h>
#include <math.h>
#include <device_functions.h>
#define DIM 256

__global__ void
  matrix_mul(float* dev_A,float* dev_B,float* dev_C,int Width)
  {
  // 2D thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  float Pvalue =0;
  for(int k=0;k<Width;++k)
  {
    float Ael=dev_A[ty*Width + k];
    float Bel=dev_B[k*Width +tx];
    Pvalue += Ael*Bel;
  }
  dev_C[ty*Width+tx]=Pvalue;
}

void CPU_matrix_mul(float* bA, float* bB, float* bC)
{
    for(int i=0;i<DIM;i++)
      for(int j=0;j<DIM;j++)
        for(int k=0;k<DIM;k++)
          bC[j + i*DIM] += bA[k+j*DIM]*bB[j+k*DIM];
}

float* random_block(int size)
{
float  *ptr;
ptr = (float *)malloc(size*sizeof(float));
for (int i=0;i<size;i++) ptr[i] = rand();
return ptr;
}
```

```
float* random_block(int size)
{
float  *ptr;
ptr = (float *)malloc(size*sizeof(float));
for (int i=0;i<size;i++)
  ptr[i] = rand();
return ptr;
}

int main(void)
{
  float *buffA = (float *)random_block(DIM*DIM);
  float *buffB = (float *)random_block(DIM*DIM);
  float *buffC = (float *)malloc(DIM*DIM*sizeof(float));
  float *dev_A, *dev_B, *dev_C;
  float elapsedTime;
  int impl=0;
  cudaEvent_t start,stop;
  printf("Test GPU or CPU [0,1]:");
  scanf("%d",&impl);
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start,0);
  cudaMalloc((void **)&dev_A,DIM*DIM*sizeof(float));
  cudaMalloc((void **)&dev_B,DIM*DIM*sizeof(float));
  cudaMalloc((void **)&dev_C,DIM*DIM*sizeof(float));
  cudaMemcpy(dev_A,buffA,DIM*DIM*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(dev_B,buffB,DIM*DIM*sizeof(float),cudaMemcpyHostToDevice);
  dim3 dimBlock(DIM/16,DIM/16); // max 32x32 -
  dim3 dimGrid(16,16);
  if(impl==0)   // GPU
   matrix_mul<<<dimGrid,dimBlock>>>(dev_A,dev_B,dev_C,DIM);
  else          // CPU
   CPU_matrix_mul(buffA,buffB,buffC);
  cudaMemcpy(buffC,dev_C,DIM*DIM*sizeof(float),cudaMemcpyDeviceToHost);
  cudaEventRecord(stop,0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime,start,stop);
  if(impl==0)
      printf("GPU time to multiply %d*%d matrix: %3.2f ms\n",
DIM,DIM,elapsedTime);
  else
      printf("CPU time to multiply %d*%d matrix: %3.2f ms\n",
DIM,DIM,elapsedTime);
  cudaFree(dev_A); cudaFree(dev_B); cudaFree(dev_C);
  free(buffA); free(buffB); free(buffC);
}
```

A faire :

1. Tester le code ci-dessus
2. Trouver une erreur importante dans la gestion des indices
3. Modifier la taille de la matrice => 512, => 1024 et analyser les résultats (quelles sont les problèmes?)

Laboratoire 2: Mémoire partagée, réductions, synchronisation

Dans ce laboratoire nous abordons quelques éléments de programmation qui « cassent » la programmation purement parallèle en exigeant l'introduction des opérations en série à l'intérieur d'un **kernel** (GPU).

2.1. Produit scalaire de 2 vecteurs (la somme de produits)

L'opération de calcul scalaire implique une addition de l'ensemble de produits simples effectués sur les élément de 2 vecteurs. Tandis que les produits peuvent être calculés en parallèle, la somme doit être effectuée en série ou partiellement en série.

2.1.1 Produit en parallèle puis la somme en série par CPU

Dans l'exemple suivant nous allons calculer le **produit de 2 vecteurs en parallèle** par une fonction **GPU**. Le résultat sera stocké dans un vecteur de sortie de la même taille qu'un vecteur de données (**N**) .
L'opération de l'**addition sera intégralement effectuée par la CPU** (en série).
Une telle solution est simple (proche de l'addition des vecteurs) mais impose une section importante d'exécution en série.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/unistd.h>
#include <sys/ioctl.h>
#include <math.h>
#define uchar unsigned char
#define imin(a,b) (a<b?a:b)
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
#define HtoD  cudaMemcpyHostToDevice
#define DtoH  cudaMemcpyDeviceToHost

const int N= 16*1024;
const int threads = 256;

const int blocks = imin(32,(N+threads-1)/threads);
//int((16*1024+256-1)/256)=64 => 32

__global__ void dot(float *a,float *b, float *c)

{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  // addition of products by blocks of threads
  while (tid<N)
  {
    c[tid] += a[tid]*b[tid];
    // addition of threads in block
    tid += blockDim.x*gridDim.x;  // number of blocks - increment
  }
}

float sum_sqrs()
{
  float sum=0;
  for(int i=0;i<N;i++) sum += i*i;
  return sum;
}

int main(void)
{
  float *a,*b,c, *partial_c;
  float *dev_a,*dev_b, *dev_partial_c;
  a = (float *)malloc(N*sizeof(float));
  b = (float *)malloc(N*sizeof(float));
```

```
  partial_c = (float *)malloc(N*sizeof(float));
  cudaMalloc((void**)&dev_a,N*sizeof(float));
  cudaMalloc((void**)&dev_b,N*sizeof(float));
  //cudaMalloc((void**)&dev_partial_c,blocks*sizeof(float));
  cudaMalloc((void**)&dev_partial_c,N*sizeof(float));
  for(int i=0;i<N;i++)
  { a[i] = i; b[i] = i; }
  cudaMemcpy(dev_a,a,N*sizeof(float),HtoD);
  cudaMemcpy(dev_b,b,N*sizeof(float),HtoD);
  dot<<<blocks,threads>>>(dev_a,dev_b,dev_partial_c);
  cudaMemcpy(partial_c,dev_partial_c,N*sizeof(float),DtoH);
  c=0;
  for(int i=0; i<N;i++)
  { c +=partial_c[i]; }
  printf("Does GPU value %f = %f\n",c,sum_sqrs());
  cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_partial_c);
  free(a); free(b); free(partial_c);
}
```

2.1.2 Produit et une somme partièlle en parallèle – GPU puis en série par CPU

Dans cet exemple nous allons calculer le produit et une partie de la somme par une fonction GPU.
Les produits scalaires partiels seront accumulés dans les mémoire cache résidant dans la mémoire partagée de chaque bloc participant dans l'exécution des threads.

Chaque bloc utilise une mémoire cache avec les indexes **`threadIdx.x`**.

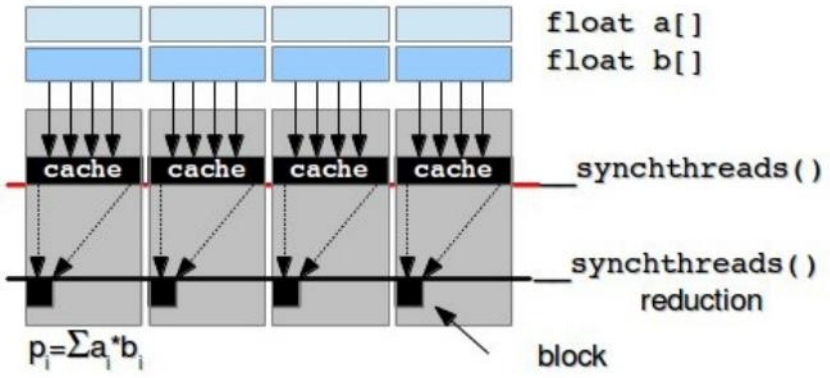


Figure 2.1 Mémoire **cache locale** partagée et synchronisation des threads avec la fonction **`synchtreads()`**

Le produits partielles seront accumulés dans ces caches par une boucle de type:

```
while (tid<N)
  {
    temp += a[tid]*b[tid];
    // addition of threads in block
    tid += blockDim.x*gridDim.x;  // number of blocks - increment
  }
  cache[cacheIndex] = temp;
```

Notons que le nombre d'éléments (**`N`**) est plus important que la taille de notre grille de threads (**`blockDim.x*gridDim.x`**), donc chaque thread accumule plusieurs produits puis il les stocke dans sa mémoire cache à l'indexe **`cacheIndex`** qui lui est propre.

A la fin d'exécution de cette partie nous avons l'ensemble de mémoires caches remplis de produits accumulés.

`cache_block0,cache_block1, … , cache_block_gridDim.x-1`

Pour être sur que l'ensemble de threads ait bien terminé l'exécution nous attendons sur un barrière de synchronisation **`__syncthreads()`**;

Dans la phase suivante nous travaillons **en parallèle sur l'ensemble de caches** , et dans chaque cache nous effectuons une addition de 2 éléments les plus éloignés.

Dans un cache de taille 32 nous commençons par 16 additions en parallèle d'élément 0 avec élément 31, élément 1 et 30, élément 2 et 29 , ..
et nous stockons les résultat dans la première moitié du cache – nous avons donc 16 éléments par bloc.

Dans la phase suivante nous recommençons par 8 additions en parallèle pour obtenir 8 sommes enregistrée dans les 8 premières cases du cache.
Nous continuons afin d'obtenir la somme accumulée sur la case 0 du cache.

L'ensemble de cases `0` de toutes les mémoire cache (nombre de blocs) est notre résultat de calcul du **produit scalaire partiel**.

Ce résultat est récupéré par la CPU pour finir l'addition en série d'un ensemble de taille correspondant au nombre de blocs.

Voici le code complet :

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/unistd.h>
#include <sys/ioctl.h>
#include <math.h>
#define uchar unsigned char
#define imin(a,b) (a<b?a:b)
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
#define HtoD  cudaMemcpyHostToDevice
#define DtoH  cudaMemcpyDeviceToHost

const int N=16*1024;
const int threads = 256;
const int blocks = imin(32,(N+threads-1)/threads);
//int((16*1024+256-1)/256)=64 => 32

__global__ void dot(float *a,float *b, float *c)
{
  __shared__ float cache[threads];  // each block has the same cache for 256
threads
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp =0;
  // addition of products by blocks of threads
  while (tid<N)
  {
    temp += a[tid]*b[tid];
    // addition of threads in block
    tid += blockDim.x*gridDim.x;  // number of blocks - increment
  }
  cache[cacheIndex] = temp; // the sum of the same index in all blocks
__syncthreads();

float sum_sqrs()
{
  float sum=0;
  for(int i=0;i<N;i++) sum += i*i;
  return sum;
}
```

```
int main(void)
{
  float *a,*b,c, *partial_c;
  float *dev_a,*dev_b, *dev_partial_c;
  a = (float *)malloc(N*sizeof(float));
  b = (float *)malloc(N*sizeof(float));
  partial_c = (float *)malloc(blocks*sizeof(float));
  cudaMalloc((void**)&dev_a,N*sizeof(float));
  cudaMalloc((void**)&dev_b,N*sizeof(float));
  cudaMalloc((void**)&dev_partial_c,blocks*sizeof(float));
  for(int i=0;i<N;i++)
  { a[i] = i; b[i] = i; }
  cudaMemcpy(dev_a,a,N*sizeof(float),HtoD);
  cudaMemcpy(dev_b,b,N*sizeof(float),HtoD);
  dot<<<blocks,threads>>>(dev_a,dev_b,dev_partial_c);
  cudaMemcpy(partial_c,dev_partial_c,blocks*sizeof(float),DtoH);
  c=0;
  for(int i=0; i<blocks;i++)
  { c +=partial_c[i]; }
  printf("Does GPU value %f = %f\n",c,sum_sqrs());
  cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_partial_c);
  free(a); free(b); free(partial_c);
}
  // cache reduction phase by step 2
  int i= blockDim.x/2;
  while(i!=0)
  {
  if(cacheIndex<i)
    cache[cacheIndex] += cache[cacheIndex+i];
  __syncthreads();
  i/=2;
  }
  if(cacheIndex ==0) c[blockIdx.x] = cache[0];
}

float sum_sqrs()
{
  float sum=0;
  for(int i=0;i<N;i++) sum += i*i;
  return sum;
}

int main(void)
{
  float *a,*b,c, *partial_c;
  float *dev_a,*dev_b, *dev_partial_c;
  a = (float *)malloc(N*sizeof(float));
  b = (float *)malloc(N*sizeof(float));
  partial_c = (float *)malloc(blocks*sizeof(float));
  cudaMalloc((void**)&dev_a,N*sizeof(float));
  cudaMalloc((void**)&dev_b,N*sizeof(float));
  cudaMalloc((void**)&dev_partial_c,blocks*sizeof(float));
  for(int i=0;i<N;i++)
  { a[i] = i; b[i] = i; }
  cudaMemcpy(dev_a,a,N*sizeof(float),HtoD);
  cudaMemcpy(dev_b,b,N*sizeof(float),HtoD);
  dot<<<blocks,threads>>>(dev_a,dev_b,dev_partial_c);
  cudaMemcpy(partial_c,dev_partial_c,blocks*sizeof(float),DtoH);
  c=0;
```

```
for(int i=0; i<blocks;i++)
  { c +=partial_c[i]; }
  printf("Does GPU value %f = %f\n",c,sum_sqrs());
  cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_partial_c);
  free(a); free(b); free(partial_c);
}
```

A faire :

1. Analyser et tester le code ci-dessus
2. Modifier la taille des données N=16*1024 => N=32*1024, N=64*1024
   et analyser les résultats (quelles sont les problèmes?)

2.2 Calcul de Pi

Le calcul de la valeur de **Pi** peut être effectué de multiples manières. Nous cherchons ici des méthodes d'accélération sur GPU.

2.2.1 Calcul de Pi par l'intégration de la surface

La première méthode est basée sur la sommation de petits rectangles. Techniquement elle ressemble au calcul du **produit scalaire** abordé dans le point précédent.
Le seul élément supplémentaire et l'utilisation de la fonction `sqrt()` intégrée dans les extensions fonctionnelles de notre GPU.

2.2.1.1 La méthode

Nous pouvons le faire avec des intégrales. Jetez un œil à la figure ci-dessous.

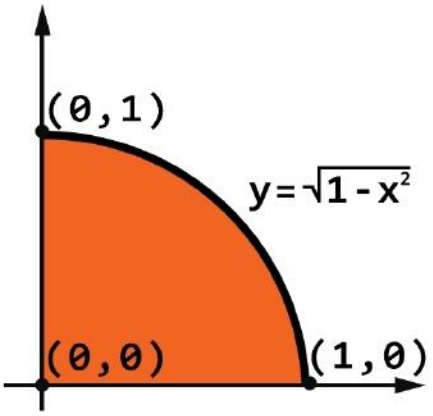


Figure 2.2 Pi calculé à partir de la surface unitaire

C'est la partie supérieure droite d'un cercle de rayon 1 centré sur (**0,0**). L'équation de ce cercle est $x^2 + y^2 = 1$. Cependant, nous ne nous soucions que de la partie au-dessus de l'axe des x afin de pouvoir réorganiser la formule en celle ci-dessous.
Puisque l'aire du cercle entier est Pi, l'aire de ce qui précède est Pi / 4. On peut intégrer de 0 à 1 et le résultat sera Pi/4. Mais, encore une fois, que signifie vraiment intégrer une fonction? L'intégrale provient de Riemann Sums, développé par Bernhard Riemann (1826). Les sommes de Riemann nous permettent d'approximer les intégrales en additionnant des rectangles sous la courbe, comme indiqué ci-dessous. Plus nous utilisons de rectangles, meilleure est l'approximation.

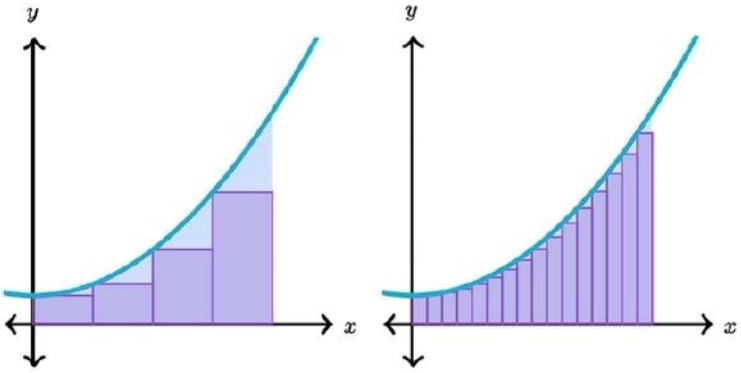


Figure 2.3 Pi –l'intégration de la surface

Le code ci-dessous utilise la méthode de Riemann Sums pour approximer la zone sous notre cercle. Cette approximation est égale à (pi/4).

2.2.1.2 Le code

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/unistd.h>
#include <sys/ioctl.h>
#include <math.h>
#define HtoD  cudaMemcpyHostToDevice
```

```
#define DtoH  cudaMemcpyDeviceToHost

const int N= 1024*1024;
const float step= 1.0/N;
const int threads = 1024;
const int blocks = 1024;

float pi_cpu()
{
int i;
float sum=0;
for(i=0;i<1024*1024;i++)
  sum+=sqrt(1.0 - step*i*step*i);
return sum;
}

__global__ void pi_gpu(float *c)
{
  __shared__ float cache[threads];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp =0;
  // addition of products by blocks of threads
  while (tid<N)
  {
    temp += sqrt(1 - step*tid*step*tid);
    // addition of threads in block
    tid += blockDim.x*gridDim.x;
  }
  cache[cacheIndex] = temp;
__syncthreads();

  // cache reduction phase by step 2
  int i= blockDim.x/2;
  while(i!=0)
  {
  if(cacheIndex<i)
    cache[cacheIndex] += cache[cacheIndex+i];
  __syncthreads();
  i/=2;
  }
  if(cacheIndex ==0) c[blockIdx.x] = cache[0];
}

int main(void)
{
  float c, *partial_c,pi;
  float *dev_partial_c;
  float elapsedTime;
  cudaEvent_t start,stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start,0);
  partial_c = (float *)malloc(blocks*sizeof(float));
  cudaMalloc((void**)&dev_partial_c,blocks*sizeof(float));
  pi_gpu<<<blocks,threads>>>(dev_partial_c);
  cudaMemcpy(partial_c,dev_partial_c,blocks*sizeof(float),DtoH);
  c=0;
  for(int i=0; i<blocks;i++)
  { c +=partial_c[i]; }
  pi=c*4.0/(1024*1024);
```

```
  cudaEventRecord(stop,0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime,start,stop);
  printf("GPU: time=%f, value=%f \n",elapsedTime,pi);
  cudaFree(dev_partial_c);
  free(partial_c);
  cudaEventRecord(start,0);
  c=pi_cpu();
  pi=c*4.0/(1024*1024);
  cudaEventRecord(stop,0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime,start,stop);
  printf("CPU: time=%f, value=%f \n",elapsedTime,pi);
}
```

A faire

Compléter le programme ci-dessus afin de pouvoir comparer les vitesses de calcul pour la CPU et la GPU

2.2.2 Calcul de Pi par la méthode Monte Carlo (valeurs aléatoires)

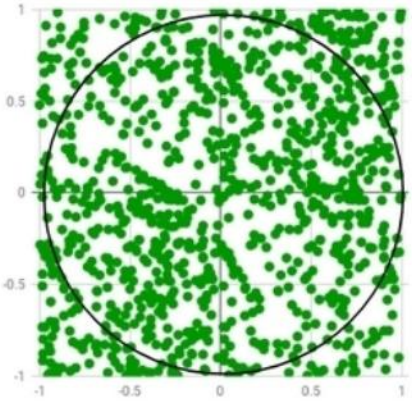
Comme dans le sujet précédant le calcul de Pi par la méthode Monte Carlo s'appuie sur l'addition des résultats partiels.
Cette fois ci les résultats partiels sont des événements créés d'une façon aléatoire. Chaque événement est une valeur double ou point (`x,y`) qui correspond ou non au placement dans un cercle.

2.2.2.1 La méthode

L'idée est de simuler des **points aléatoires** (**x,y**) dans un plan 2D avec un domaine comme un carré d'unité de côté 1. Imaginez un cercle à l'intérieur du même domaine avec le même diamètre et inscrit dans le carré. Nous calculons ensuite le ratio du nombre de points qui se trouvaient à l'intérieur du cercle et le nombre total de points générés.

Reportez-vous à l'image ci-dessous:.

Figure 2.4 Pi – calcul par points aléatoires



2.2.2.2 Le code

Les valeurs aléatoires sont générées par la fonction de **GPU**:

```
curand_uniform(&rng);  // rng - range
```

initialisée précédemment par :

```
curandState_t rng;
curand_init(clock64(), tid, 0, &rng);
```

La valeur de Pi correspond à :

```
4*(nombre_de_cas_dans_le_cercle)/(nombre_de_cas_total)
```

```
#include <stdio.h>
#include <stdlib.h>
#include <limits>
#include <cuda.h>
#include <curand_kernel.h>
typedef unsigned long long Count;
const Count WARP_SIZE = 32; // Warp size
const Count NBLOCKS = 640; // Number of total cuda cores on my GPU
const Count ITERATIONS = 1000000; // Number of points to generate (each thread)

// This kernel is
__global__ void picount(Count *totals) {
        // Define some shared memory: all threads in this block
        __shared__ Count counter[WARP_SIZE];
        // Unique ID of the thread
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        // Initialize RNG
        curandState_t rng;
        curand_init(clock64(), tid, 0, &rng);
        // Initialize the counter
        counter[threadIdx.x] = 0;
        // Computation loop
        for (int i = 0; i < ITERATIONS; i++) {
                float x = curand_uniform(&rng); // Random x position in [0,1]
                float y = curand_uniform(&rng); // Random y position in [0,1]
                counter[threadIdx.x] += 1 - int(x * x + y * y); // Hit test
        }
        // The first thread in *every block* should sum the results
        if (threadIdx.x == 0) {
                // Reset count for this block
                totals[blockIdx.x] = 0;
                // Accumulate results
                for (int i = 0; i < WARP_SIZE; i++) {
                        totals[blockIdx.x] += counter[i];
                }
        }
}

int main(int argc, char **argv)
{
        printf("Starting simulation with: %llu blocks, %llu threads, %llu
iterations\n",NBLOCKS,WARP_SIZE,ITERATIONS);
        // Allocate host and device memory to store the counters
        Count *hOut, *dOut;
        hOut = new Count[NBLOCKS]; // Host memory
        cudaMalloc(&dOut, sizeof(Count) * NBLOCKS); // Device memory
        // Launch kernel
        picount<<<NBLOCKS, WARP_SIZE>>>(dOut);
        // Copy back memory used on device and free
        cudaMemcpy(hOut, dOut, sizeof(Count) * NBLOCKS, cudaMemcpyDeviceToHost);
        cudaFree(dOut);
        // Compute total hits
        Count total = 0;
        for (int i = 0; i < NBLOCKS; i++) {
                total += hOut[i];
        }
        Count tests = NBLOCKS*ITERATIONS*WARP_SIZE;
        printf("Approximated PI using %llu random test\n",tests);
        // Set maximum precision for decimal printing
        printf("PI ~= %f\n",4.0*(double)total/(double)tests);
        return 0;}
```

A faire

Compléter le programme ci-dessus afin de pouvoir comparer les vitesses de calcul pour la CPU et la GPU.

Annexe - Google Colab pour programmation CUDA

Avant de pouvoir effectivement programmer et exécuter nos programmes CUDA dans l'environnement Google-Colab il nous faut préparer cet environnement. Notez qu'avec Colab, vous pouvez travailler gratuitement avec CUDA C / C ++ sur le GPU type Nvidia Tesla **K80** avec **11,5 Go gratuit !**
Après l'inscription au Google-Colab vous pouvez ouvrir un lien :

https://colab.research.google.com/notebooks/intro.ipynb

A.1 Commencez par créer un nouveau bloc-notes

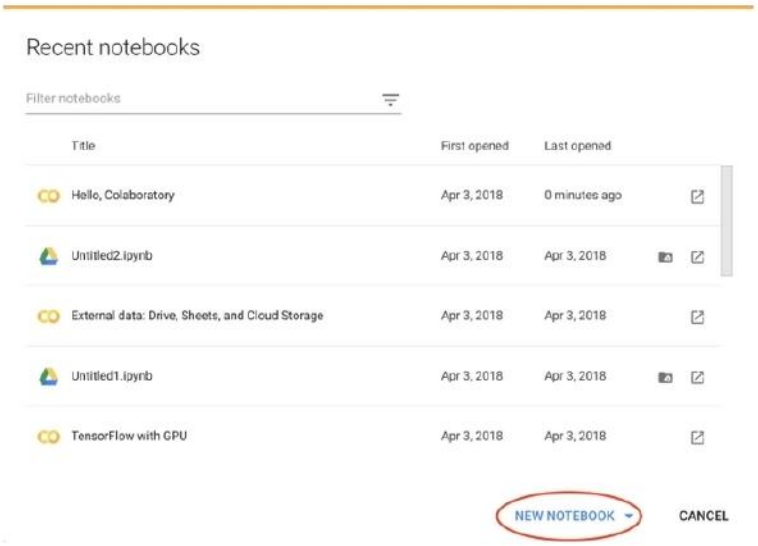


Figure A.1 Liste de **notebooks** après l'ouverture de **Colab** (un exemple)

Veuillez sélectionner Python 3 Notebook dans la fenêtre contextuelle.
Si vous avez déjà travaillé avec **Jupyter**, l'interface vous semblera familière. Un peu plus élégant cependant. Si vous ne l'avez pas fait, ne vous inquiétez pas. C'est un outil assez simple et très puissant, c'est ainsi qu'il est si populaire.

A.2 Choisisez le *run-time*

Ensuite, nous devons passer notre runtime du **CPU** au **GPU**. Nous sommes à seulement 2 clics.

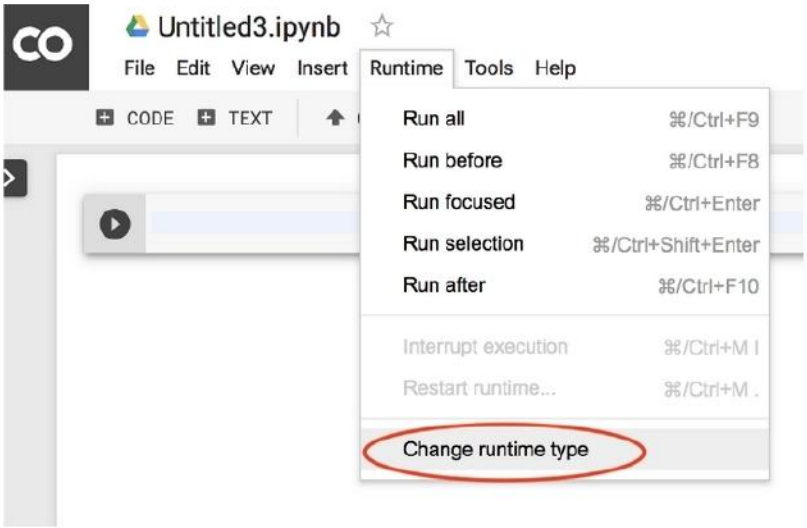


Figure A.2 Préparation de runtime (CPU, GPU, TPU)

Modifiez le type d'exécution dans les paramètres de Notebook sous l'onglet **Runtime** du menu supérieur:

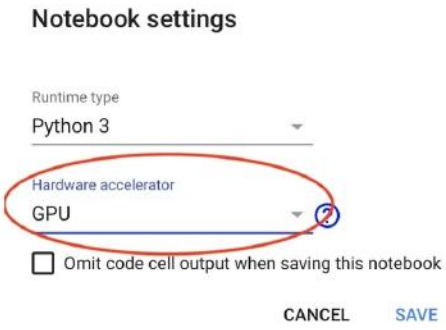


Figure A.3 Sélection de runtime (CPU, GPU, TPU)

Et cliquez sur Save.

Bien que les bibliothèques CUDA soient disponibles pour l'environnement Tensorflow, Colab n'a pas installé **NVCC** (**Nvidia CUDA Compiler**).

A.3 Préparation – installation de CUDA (Version 9)

Commencez par désinstaller complètement toutes les versions précédentes de CUDA.
Notez que **'!'** est ajouté au début d'une ligne c'est qui permet d'être exécuté en tant qu'une commande

```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

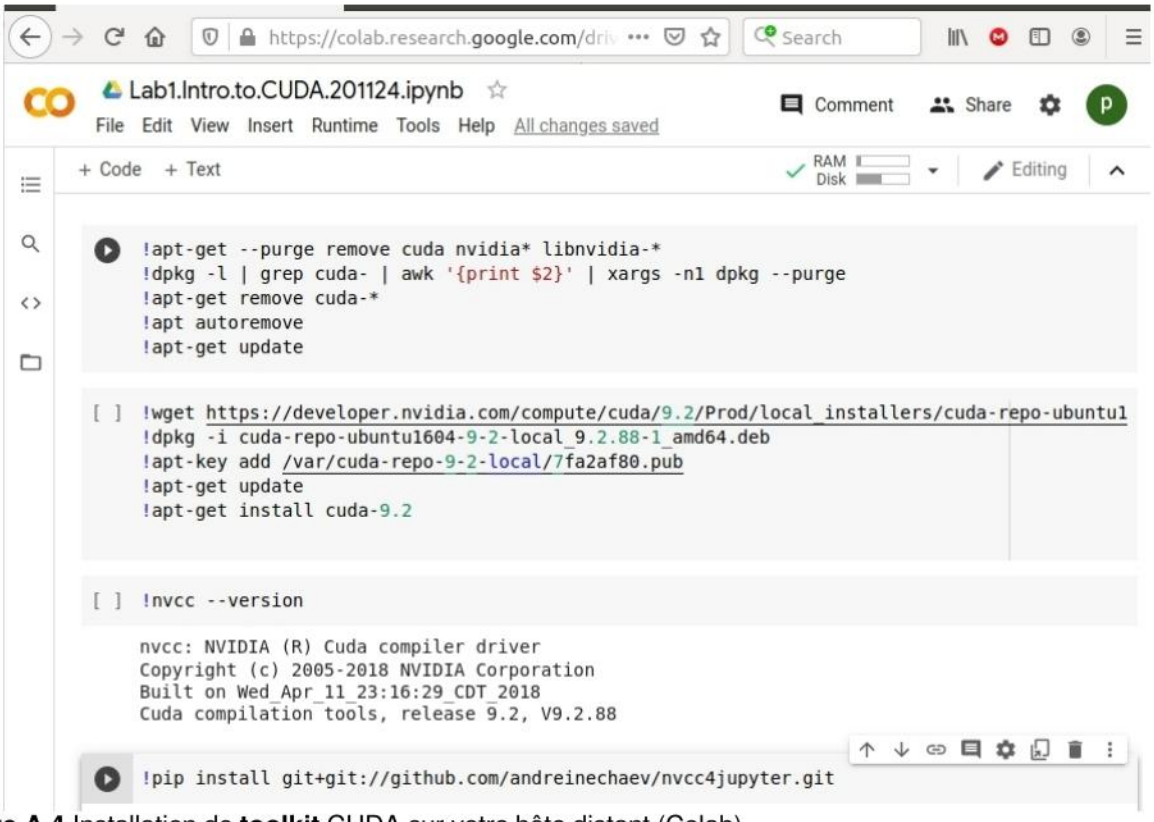


Figure A.4 Installation de **toolkit** CUDA sur votre hôte distant (Colab)

Puis **installez** CUDA Version 9.

```
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64 -O cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
!apt-get update
!apt-get install cuda-9.2
```

Vous pouvez maintenant tester votre installation CUDA en exécutant

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Wed_Apr_11_23:16:29_CDT_2018
Cuda compilation tools, release 9.2, V9.2.88
```

Nous avons presque fini. On acréé une petite extension pour exécuter **NVCC** à partir de cellules Notebook. Installez-la avec :

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

Vous devez maintenant charger l'extension installée, en exécutant:

```
%load_ext nvcc_plugin
```

A.4 Compiler et exécuter votre premier programme CUDA sur un hôte distant

Nous sommes prêts à exécuter le code CUDA C/C ++ directement dans votre ordinateur portable.
Pour cela, nous devons dire explicitement à l'interpréteur que nous voulons utiliser l'extension en ajoutant **%%cu** au début de chaque cellule avec le code CUDA.

A.4.1 Premier exemple de test

```
%%cu
#include <iostream>int main() {
    std::cout << "Hello world\n";
    return 0;
}
```

A.4.2 L'analyse du circuit GPU sur Google Colab

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
```

```
int main(void)
{
cudaDeviceProp dP; //dP short for deviceProperties
int device = 0;
cudaGetDeviceProperties(&dP, device);
printf ("Name:%s \n", dP.name);
printf ("Memory total:%d MB \n", dP.totalGlobalMem / (1024*1024));
printf ("Shared memory per block:%d in B \n", dP.sharedMemPerBlock);
printf ("MaxThreads block:%d \n", dP.maxThreadsPerBlock);
printf ("warpSize:%d \n", dP.warpSize);
printf ("major:%d \n", dP.major);
printf ("minor:%d \n", dP.minor);
printf ("number of SM:%d \n", dP.multiProcessorCount);
printf ("Clock frequency:%1.3f inGHz \n", dP.clockRate/1000000.0);
return 0;
}
```

Résultat :

```
Name:Tesla K80
Memory total:11441 MB
Shared memory per block:49152 in B
MaxThreads block:1024
warpSize:32
major:3
minor:7
number of SM:13
Clock frequency:0.824 inGHz
```

Analysez le résultat – combien il y a de cores CUDA d'exécution (recherche sur Internet) ?

Passez aux exercices de Lab1, Lab2

Table of Contents