وظيفة برمجة متوازية

OpenMP && CUDA

January 31 , 2024

تقديم: أحمد عبدالله

السؤال الأول:

الهدف من هذا السؤال هو ايجاد قيمة التكامل بطريقة عددية للمنطقة الواقعة تحت منحنىX^2 ، في المجال[0,1] , وذلك من خلال اتباع طريقة التجزئة والتي تقوم على تقسيم المساحة الكلية إلى مساحات منفصلة جزئية يمكن حسابها، فتكون المساحة الكلية هي مجموع المساحات الجزئية (بتقريب مقبول).

و للقيام بذلك تم كتابة برنامج متوازي باستخدام واجهة OpenMP لحساب هذه المساحة عن طريق توزيع حساب المساحات الجزئية بين عدة نياسب ثم تجميعها عند النيسب الرئيسي لتعبر عن المساحة الكلية.

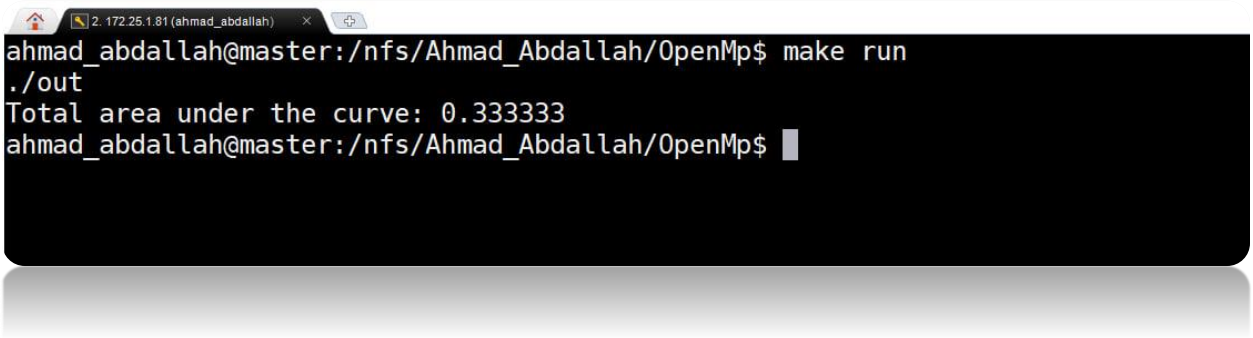
حيث تم تقسيم المساحة الكلية الى مستطيلات ذات ارتفاعات متزايدة ولكن لها العرض نفسه وعددها 1000000، و بحيث يقوم كل نيسب بحساب المساحة الجزئية للمستطيلات المخصصة له، ويتم دمج النتائج للحصول على المساحة الإجمالية. ثم تتم طباعة النتيجة النهائية على وحدة التحكم.

حيث من خلال هذه التعلمية

#pragma omp parallel for reduction(+:total_area)

نقوم بالبدء بمنطقة متوازية (parallel region) مع حلقة تكرارية و تضمن بأن المتحول total_area خاص بكل نيسب على حدة، وفي نهاية الحلقة يتم دمج النتائج الجزئية للحصول على الناتج النهائي.

و كان الخرج على الشكل التالي:



ما هو التنفيذ المتوازي الأفضل لهذه المسألة OpenMP أو MPI أو الاثنين معاً؟

في العموم لنذكر خصائص كل من الطريقتين وايجابياتها.

بالنسبة ل OpenMP **:**

- نموذج برمجة متوازي للذاكرة المشتركة، اي ان تتمتع الاجرائيات (processes) أو النوى (cores) بإمكانية الوصول إلى جزء مشترك من الذاكرة.
- سهل الاستخدام مع توجيهات المترجم (compiler directives) لتحقيق الموازاة.
- عادة ما تكون أسهل في التنفيذ والفهم للمهام التي يمكن موازنتها داخل جهاز واحد.

و في مسألتنا قمنا بتطبيق ال OpenMP من خلال توزيع حساب المساحات الجزئية للمستطيلات بين عدة نياسب تعمل على نفس العقدة(الحاسب). يمكن تعيين مجموعة فرعية من المستطيلات لكل نيسب لحساب المساحات الجزئية، ومن ثم يمكن دمج النتائج للحصول على المساحة الاجمالية.

بالنسبة ل MPI:

- مناسب تمامًا لأنظمة الذاكرة الموزعة.
- يتطلب تواصل بين العمليات، من خلال عمليات(ارسال/ استقبال) وهذا ما يزيد زمن التواصل
- يُستخدم عادةً في مجموعات الحوسبة عالية الأداء.

في مسألتنا يمكن تطبيق ال MPI من خلال تقسيم حساب المساحات الجزئية للمستطيلات على عدة عقد (حواسب) ومن ثم يمكن جمع النتائج ودمجها للحصول على المساحة الإجمالية.

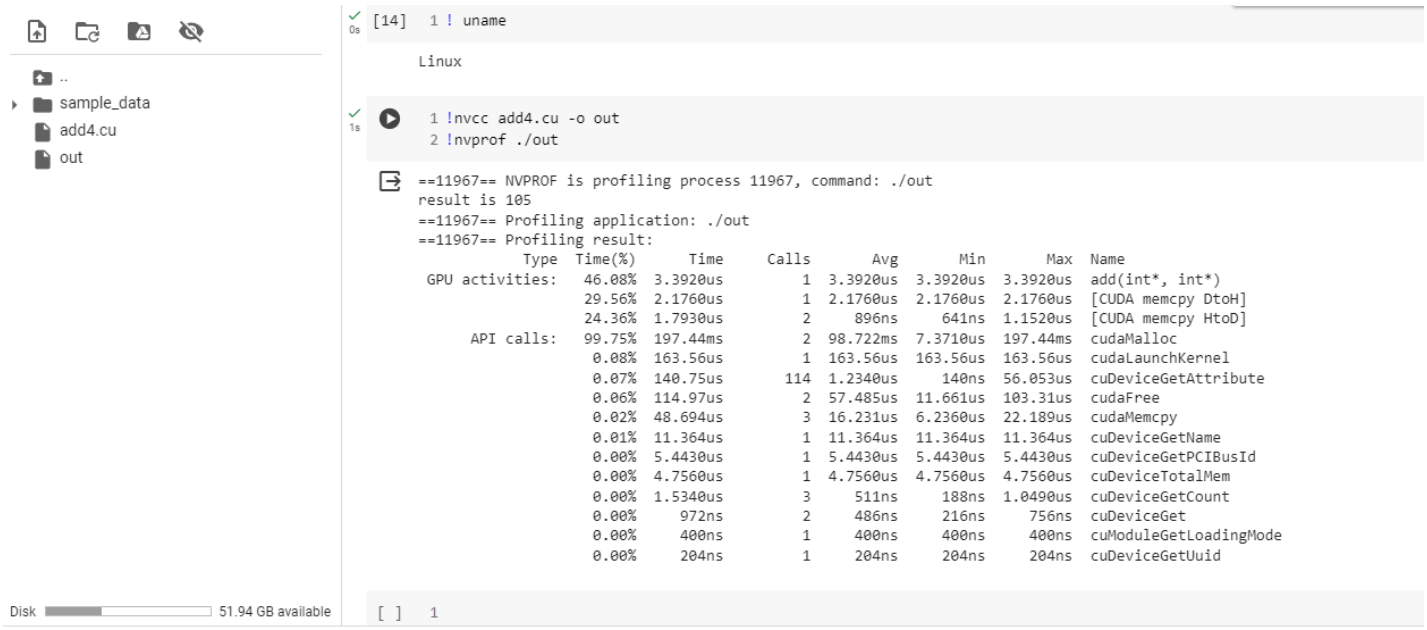
الجمع بين OpenMP و MPI يمكن تطبيقه في مسألتنا من خلال تقسيم حساب المساحات الجزئية للمستيطلات على عدة عقد، وضمن كل عقدة يتم توزيع مجموعة المساحات الخاصة بها على عدة نياسب مشتركة بالذاكرة و بذلك نكون قد خففنا زمن التواصل الذي سيكون اكبر فيما لو استخدمنا MPI لوحدها و سيتم الاستفادة من موارد حاسوبية اكبر و ايجاد الحل بوقت اقل ربما من استخدام OpenMP لوحدها.

ولكنني اجد و تبعا لمعطيات المسألة ان تطبيق OpenMP لوحدها افضل، حيث اننا نواجه مسألة بسيطة لا تحتاج الى موارد حاسوبية كبيرة و بالتالي يمكننا الاستفادة من فكرة الذاكرة المشتركة و تطبيق OpenMP ، من خلال تقسيم عبء العمل والحساب بين النياسب التي تتقاسم نفس الذاكرة. و بذلك نكون قد خففنا زمن التواصل بشكل كبير وحصلنا على نتائج جيدة ضمن وقت جيد.

قد يكون MPI او الجمع بين الطريقتين افضل ولكن في حالة إذا كنا تتعامل مع نظام ذاكرة موزعة، و لدينا معطيات المسألة اعقد وبالتالي نحتاج الى موارد حاسوبية اكثر.

السؤال الثاني:

1) لايجاد البرنامج الخطأ ضمن البرامج الأربعة تم تشغيل كل منها على حدا و كانت النتائج كالتالي:
بالنسبة للبرنامج الأول



و اعطى الخرج المتوقع منه حيث كان الهدف من هذا البرنامج هو جمع القيمتين a, b والنتيجة التي اعطاها الخرج كانت قيمة صحيحة
و بالمرور على البرنامج وقرائته و جدت ان البرنامج صحيح و لا يوجد فيه اي خطأ منطقي او خطأ في التنفيذ.

بالنسبة للبرنامج الثاني :

```
==18972== NVPROF is profiling process 18972, command: ./out
```

5.000000	6.000000	7.000000	8.000000	9.000000	10.000000	11.000000	12.000000
15.000000	16.000000	17.000000	18.000000	19.000000	20.000000	21.000000	22.000000
25.000000	26.000000	27.000000	28.000000	29.000000	30.000000	31.000000	32.000000
35.000000	36.000000	37.000000	38.000000	39.000000	40.000000	41.000000	42.000000
45.000000	46.000000	47.000000	48.000000	49.000000	50.000000	51.000000	52.000000
55.000000	56.000000	57.000000	58.000000	59.000000	60.000000	61.000000	62.000000
65.000000	66.000000	67.000000	68.000000	69.000000	70.000000	71.000000	72.000000
75.000000	76.000000	77.000000	78.000000	79.000000	80.000000	81.000000	82.000000
85.000000	86.000000	87.000000	88.000000	89.000000	90.000000	91.000000	92.000000
95.000000	96.000000	97.000000	98.000000	99.000000	100.000000	101.000000	102.00000
105.000000	106.000000	107.000000	108.000000	109.000000	110.000000	111.000000	112.00000
115.000000	116.000000	117.000000	118.000000	119.000000	120.000000	121.000000	122.00000
125.000000	126.000000	127.000000	128.000000	129.000000	130.000000	131.000000	132.00000
135.000000	136.000000	137.000000	138.000000	139.000000	140.000000	141.000000	142.00000
145.000000	146.000000	147.000000	148.000000	149.000000	150.000000	151.000000	152.00000
155.000000	156.000000	157.000000	158.000000	159.000000	160.000000	161.000000	162.00000
165.000000	166.000000	167.000000	168.000000	169.000000	170.000000	171.000000	172.00000
175.000000	176.000000	177.000000	178.000000	179.000000	180.000000	181.000000	182.00000
185.000000	186.000000	187.000000	188.000000	189.000000	190.000000	191.000000	192.00000
195.000000	196.000000	197.000000	198.000000	199.000000	200.000000	201.000000	202.00000
205.000000	206.000000	207.000000	208.000000	209.000000	210.000000	211.000000	212.00000
215.000000	216.000000	217.000000	218.000000	219.000000	220.000000	221.000000	222.00000
225.000000	226.000000	227.000000	228.000000	229.000000	230.000000	231.000000	232.00000
235.000000	236.000000	237.000000	238.000000	239.000000	240.000000	241.000000	242.00000
245.000000	246.000000	247.000000	248.000000	249.000000	250.000000	251.000000	252.00000
255.000000	256.000000	257.000000	258.000000	259.000000	260.000000	261.000000	262.00000
265.000000	266.000000	267.000000	268.000000	269.000000	270.000000	271.000000	272.00000
275.000000	276.000000	277.000000	278.000000	279.000000	280.000000	281.000000	282.00000
285.000000	286.000000	287.000000	288.000000	289.000000	290.000000	291.000000	292.00000

+ Code + Text

T4 RAM Disk

9835.000000	9836.000000	9837.000000	9838.000000	9839.000000	9840.000000	[illegible]	[illegible]
9845.000000	9846.000000	9847.000000	9848.000000	9849.000000	9850.000000		
9855.000000	9856.000000	9857.000000	9858.000000	9859.000000	9860.000000	9861.000000	9862.0000
9865.000000	9866.000000	9867.000000	9868.000000	9869.000000	9870.000000	9871.000000	9872.0000
9875.000000	9876.000000	9877.000000	9878.000000	9879.000000	9880.000000	9881.000000	9882.0000
9885.000000	9886.000000	9887.000000	9888.000000	9889.000000	9890.000000	9891.000000	9892.0000
9895.000000	9896.000000	9897.000000	9898.000000	9899.000000	9900.000000	9901.000000	9902.0000
9905.000000	9906.000000	9907.000000	9908.000000	9909.000000	9910.000000	9911.000000	9912.0000
9915.000000	9916.000000	9917.000000	9918.000000	9919.000000	9920.000000	9921.000000	9922.0000
9925.000000	9926.000000	9927.000000	9928.000000	9929.000000	9930.000000	9931.000000	9932.0000
9935.000000	9936.000000	9937.000000	9938.000000	9939.000000	9940.000000	9941.000000	9942.0000
9945.000000	9946.000000	9947.000000	9948.000000	9949.000000	9950.000000	9951.000000	9952.0000
9955.000000	9956.000000	9957.000000	9958.000000	9959.000000	9960.000000	9961.000000	9962.0000
9965.000000	9966.000000	9967.000000	9968.000000	9969.000000	9970.000000	9971.000000	9972.0000
9975.000000	9976.000000	9977.000000	9978.000000	9979.000000	9980.000000	9981.000000	9982.0000
9985.000000	9986.000000	9987.000000	9988.000000	9989.000000	9990.000000	9991.000000	9992.0000
9995.000000	9996.000000	9997.000000	9998.000000	9999.000000	10000.000000	10001.000000	10002.000

```
==18972== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.61%  1.2443ms         1  1.2443ms  1.2443ms  1.2443ms  vector_add(float*, float*, float*, int)
                    0.99%  12.512us         2  6.2560us  6.1120us  6.4000us  [CUDA memcpy HtoD]
                    0.40%  5.0240us         1  5.0240us  5.0240us  5.0240us  [CUDA memcpy DtoH]
      API calls:   98.99%  193.76ms         3  64.586ms  3.0840us  193.75ms  cudaMalloc
                    0.69%  1.3557ms         3  451.89us  25.002us  1.2960ms  cudaMemcpy
                    0.13%  262.56us         3  87.519us  8.0290us  204.09us  cudaFree
                    0.10%  189.39us         1  189.39us  189.39us  189.39us  cudaLaunchKernel
                    0.07%  139.81us       114  1.2260us     140ns  56.036us  cuDeviceGetAttribute
                    0.01%  10.882us         1  10.882us  10.882us  10.882us  cuDeviceGetName
                    0.00%  5.2660us         1  5.2660us  5.2660us  5.2660us  cuDeviceGetPCIBusId
                    0.00%  5.2160us         1  5.2160us  5.2160us  5.2160us  cuDeviceTotalMem
                    0.00%  2.6690us         2  1.3340us     181ns  2.4880us  cuDeviceGet
                    0.00%  1.5220us         3     507ns     201ns  1.0140us  cuDeviceGetCount
```

ايضا اعطى البرنامج الثاني اعطى الخرج المتوقع له حيث الهدف منه هو حساب مجموع عناصر مصفوفتين بطول 10000 الاولى عناصرها بالترتيب من 0 الى 9999 و الثانية عناصرها جميعها 5، واعطت الخرج المتوقع بشكل صحيح اي انه لا يوجد خطأ على مستوى التنفيذ.

لكن هناك خطأ على مستوى الكود وذلك في الاستفادة من فكرة التوازي و هو انه تم تعريف block واحد يحوي 256 نيسب جميعها تقوم بنفس المهمة اي اننا لم نستفد من اي نيسب تم تعريفه لان نيسب ال main لوحده قام باجراء الحساب مرة و باقي 256 نيسب قاموا بتكرار العملية ذاتها، وهذا خطأ كبير للغاية. أثر هذا الخطأ هو تكرار الكود نفسه 256 مرة و هذا سيزيد من التنفيذ. وهذا ما توضحه هذه الصورة.

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i += 1){
        out[i] = a[i] + b[i];
    }
}

    // Executing kernel
    vector_add<<<1,256>>>(d_out, d_a, d_b, N);
```

بالنسبة للبرنامج الثالث:
يشبه البرنامج الثاني من حيث الهدف حيث ولكن الفرق في المصفوفة الثانية التي قيمها جميعها 26 و المصفوفة الاولى عناصرها هي من 1 الى 10000 بالترتيب و اعطت الخرج المتوقع منها.
ولكن بالنسبة للبرنامج الثالث فقد عمل بشكل صحيح واستفاد من التوازي بشكل كامل.
ولاتوجد فيه اي خطا لا على مستوى التنفيذ ولا على مستوى منطق التوازي.

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x;
    int stride = blockDim.x;

    for(int i = index; i < n; i += stride){
        out[i] = a[i] + b[i];
      }
}
```

بينما البرنامج الرابع :

```
178.000000  179.000000  180.000000  181.000000  182.000000  183.000000  184.000000  185.000000  186.00000
188.000000  189.000000  190.000000  191.000000  192.000000  193.000000  194.000000  195.000000  196.00000
198.000000  199.000000  200.000000  201.000000  202.000000  203.000000  204.000000  205.000000  206.00000
208.000000  209.000000  210.000000  211.000000  212.000000  213.000000  214.000000  215.000000  216.00000
218.000000  219.000000  220.000000  221.000000  222.000000  223.000000  224.000000  225.000000  226.00000
228.000000  229.000000  230.000000  231.000000  232.000000  233.000000  234.000000  235.000000  236.00000
238.000000  239.000000  240.000000  241.000000  242.000000  243.000000  244.000000  245.000000  246.00000
248.000000  249.000000  250.000000  251.000000  252.000000  253.000000  254.000000  255.000000  256.00000
258.000000  259.000000  260.000000  261.000000  262.000000  263.000000  264.000000  265.000000  266.00000
268.000000  269.000000  270.000000  271.000000  272.000000  273.000000  274.000000  275.000000  276.00000
278.000000  279.000000  280.000000  281.000000  282.000000  0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
```

هو الذي يحتوي خطأ في التنفيذ حيث لم يقم بالذي مفروض ان يقوم به.
والخطأ موجود في هذا القسم:

```
// Executing kernel
vector_add<<<1,256>>>(d_out, d_a, d_b, N);
```

الخطأ هو أن النواة يتم تشغيلها بكتلة واحدة و256 نيسب لكل كتلة ، و نحن لدينا احجام المصفوفات a, b, out هو 10000، وهو أكبر من عدد النياسب 256 ضمن ال grid هذا يعني أنه لن تتم معالجة جميع عناصر المصفوفات بواسطة النياسب التي تم تشغيلها، مما يؤدي إلى نتائج غير صحيحة وهذا ما توضحه الصورة حيث نجد ان المجموع وصل الى القيمة 256+16=282 و باقي عناصر المصفوفة اصفار.

حيث فقط اول 256 قيمة تمت معالجتهم بشكل صحيح واما باقي القيم لم تتم معالجتها.

ببساطة يمكن حل المشكلة من خلال زيادة عدد النياسب ضمن الكتلة الواحدة بحيث تحتوي 10000 نيسب او زيادة عدد الكتل بحيث يكون لدينا مجموع نياسب الكتل اكبر او يساوي 10000.

2) للاستفادة من المعالجات المتوازية المتعددة في وحدة معالجة الرسومات CUDA لتحسين اداء البرنامج، يمكن تشغيل النواة باستخدام كتل متعددة بدلا من كتلة واحدة فقط بذلك، نكون قد وزعنا عبء العمل عبر عدة SMs والسماح لهم بمعالجة البيانات بالتوازي.

تم اجراء التعديلات على البرنامج cu4.Vec_a حيث تم اضافة حلقة تكرارية للمرور على الاحجام المختلفة (32,128,512,1024). لكل حجم كتلة، يتم حساب عدد الكتل المطلوبة بناء على العدد اإلجمالي للعناصر N وحجم الكتلة. وقد تم تشغيل البرنامج و اعطى نتائج صحيحة.