الحوسبة المتوازية

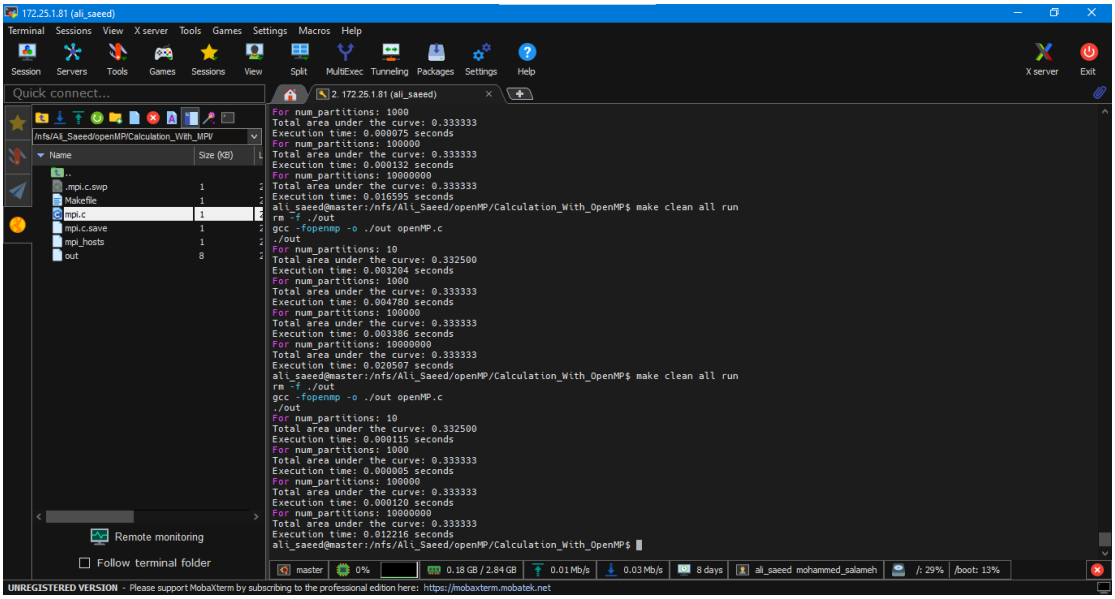
أولاً: واجهة برمجة التطبيقات OpenMP

بالنسبة للمسألة المحددة لحساب المساحة تحت منحنى x^2 باستخدام التكامل العددي والتوازي ضمن المجال من صفر إلى واحد، يمكن استخدام كل من OpenMP وMPI. ومع ذلك، فإن الاختيار بين OpenMP أو MPI أو كليهما يعتمد على عوامل مختلفة.

OpenMP هو نموذج برمجة متوازي للذاكرة المشتركة ومناسب تمامًا للمشكلات التي يمكن أن تعمل فيها نياسب متعددة على البيانات المشتركة. في حالتنا، يمكن استخدام OpenMP لتوزيع حساب المساحات الجزئية بين عدة نياسب تعمل على نفس العقدة. يمكن تعيين مجموعة فرعية من المستطيلات لكل نيسب لحساب المساحات الجزئية، ومن ثم يمكن دمج النتائج للحصول على المساحة الإجمالية.

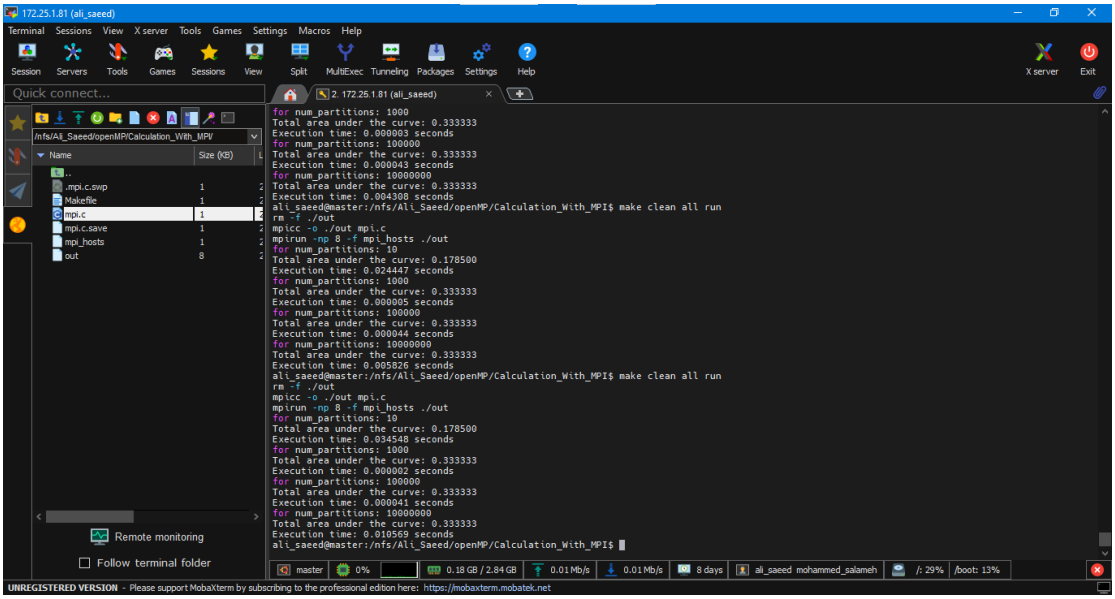
يمكن القول أن نموذج openMP سيكون فعّال جداً لو كانت العقدة (الجهاز) تمتلك موارد عالية جداً وكافية لتشغيل أي عدد نريده من النياسب، حيث أن اللجوء إلى نموذج openMP سيوفّر العبء الناجم عن تواصل النياسب من خلال عمليات الارسال والاستقبال في حالة MPI، مما يحسّن الأداء ويسرّع التنفيذ، يجب أن ننتبه أيضاً أن عملية تزامن النياسب من أجل حماية المعطيات المشتركة أيضاً عملية مكلفة تتطلب وقت إضافي يجب أن يؤخذ بعين الاعتبار وأيضاً يجب الانتباه إلى تقليل عملية التنافس على القفل لتقتصر على أقل وقت وتنفيذ ممكن. ولكن في حالتنا نجد أننا نعمل على Cluster مكون من ثلاث عقد فيزيائية كل منها يضم أربع معالجات افتراضية وبالتالي على عقدة واحدة يمكن تشغيل على الأكثر أربع نياسب يعملون معاً على التوازي، وإن أي زيادة أكثر من أربع نياسب لن تكون مفيدة لأنه عندها سيلجأ نظام التشغيل لجدولة النياسب.

تم تجريب حل المسألة باستخدام نموذج openMP باستخدام اربع نياسب لحساب التكامل المطلوب مع تجريب تقسيم المجال الكلي إلى أعداد مختلفة من المجالات الفرعية فكانت النتائج كما يلي

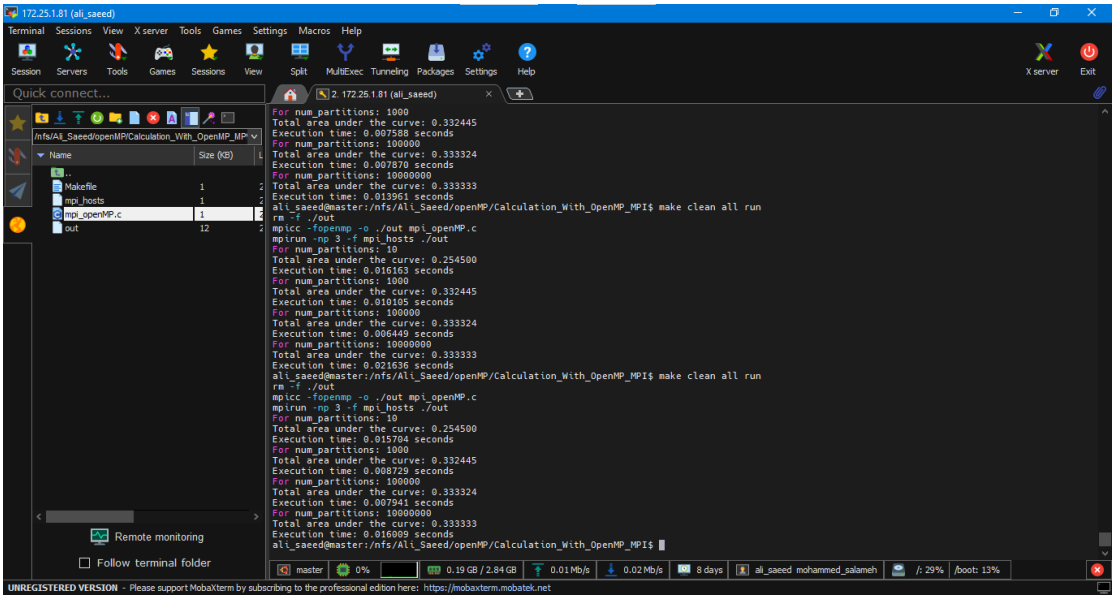


MPI، من ناحية أخرى، هي واجهة لتمرير الرسائل تُستخدم عادةً للبرمجة المتوازية للذاكرة الموزعة. فهو يسمح لاجرائيات متعددة تعمل على أجهزة مختلفة بالتواصل مع بعضها البعض. في سياق حالتنا، يمكن استخدام MPI لتوزيع حساب المساحات الجزئية بين الاجرائيات المتعددة التي تعمل على أجهزة مختلفة. يمكن تخصيص مجموعة فرعية من المستطيلات لكل اجرائية لحساب المساحات الجزئية، ومن ثم يمكن جمع النتائج ودمجها للحصول على المساحة الإجمالية. يمكن القول بأنه في حال وجود أكثر من جهاز يعملون معاً (عنقود) يُفضّل استخدام نموذج MPI أكثر من نموذج openMP لانه سيؤدي إلى الاستفادة من كامل الموارد المتاحة أي تشغيل عدد أكبر من الاجرائيات مما يؤدي إلى تسريع التنفيذ، ولكن أيضاً لا يجب أن يتم اهمال العبء الناجم عن التواصل بين النياسب على الأجهزة المختلفة أي العبء الناجم عن عمليات الارسال والاستقبال وحجم المعطيات المرسلة والمستقبلة، وهنا يجب ان نأخذ بعين الاعتبار المفارقة بين زمن الحساب وزمن الاتصال وأيهما يطغى على الآخر.

وفقا للعنقود الذي نعمل عليه، قمنا بتحديد عدد الاجرائيات ب 12 إجرائية ستعمل على أجهزة العنقود الثلاثة بشكل متوازي ضمن كل عقدة سيتم تشغيل اربع اجرائيات، وكانت النتائج من أجل حساب التكامل العددي المطلوب مع تجريب تقسيم المجال الكلي إلى أعداد مختلفة من المجالات الفرعية كما يلي



نموذج openMP + MPI تأتي أهمية هذا النموذج من كونه يحقق الاستفادة من كامل الموارد المتاحة ومع زمن تواصل بين النياسب أقل بحيث يقلل عمليات الارسال والاستقبال ومع الاخذ بعين الاعتبار أن الزمن الضائع لتحقيق عملية التزامن بين النياسب لحماية المعطيات المشتركة ضمن العقدة الواحدة سيبقى. في حالتنا تم استخدام نموذج MPI لتشغيل إجرائية واحدة على كل عقدة من عقد العنقود الثلاثة، ومن ثم هذه الإجرائية ستقوم باستخدام نموذج openMP من أجل القيام بالحسابات المطلوبة من خلال توليد أربع نياسب سيعملون معاً على العقدة نفسها (أي 12 نيسب يعملون معاً على التوازي من أجل العقد الثلاث)، في هذه الحالة النياسب ضمن العقدة الواحدة لن تحتاج إلى التواصل فيما بينها من خلال عمليات الارسال والاستقبال لانه يوجد فيما بينها ذاكرة مشتركة ولكن يوجد تزامن، وانما سيكون التواصل فقط بين الاجرائيات الثلاثة التي تم انشاؤها باستخدام MPI. النتائج كانت كما يلي



خلاصة

بشكل عام وبناءاً على النتائج التي حصلنا عليها نجد أن لكل نموذج من النماذج السابقة إيجابيات وسلبيات وبالتالي لا يمكن أن نقول بالمطلق أن نموذج معين هو الأفضل، وإنما تحديد أي نموذج هو الأفضل يتعلق بعوامل عديدة منها

- عدد الأجهزة المتاحة.
- موارد كل جهاز.
- حجم المسألة.
- صعوبة/سهولة مزامنة عمل النياسب لحماية المعطيات المشتركة.
- حجم المعطيات المرسلة أو المستقبلة في كل عملية ارسال أو استقبال.

ثانياً: CUDA

1. ضمن البرامج المعطاة يوجد اكثر من برنامج خاطئ:

البرنامج الثاني add_Vec2.cu. لا يحتوي خطأ من حيث التنفيذ وانما يحتوي خطأ في منطق الاستفادة من التفرعية والتوازي في CUDA. حيث يتم تعريف block واحد يحتوي 256 نيسب كل منها يقوم بحساب مجموع الشعاعين كاملاً، أي أن حساب مجموع الشعاعين سيتكرر 256 مرة، وهذا سيء للغاية.

```
__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);
```

يمكن بشكل مبدئي حل هذه المشكلة من خلال تحديد عدد النياسب ضمن الكتلة بنيسب واحد وهكذا نحصل على تنفيذ تسلسلي بحت، أي لا يوجد استفادة من التوازي باستخدام CUDA.

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

البرنامج الثالث يعمل بشكل صحيح ويستفاد من التوازي بشكل كامل، ولكن الجدير بالذكر أنه في حال تم زيادة عدد الكتل، مع بقاء تابع vector_add كما هو سنحصل على تكرار التنفيذ من اجل كل كتلة حيث ان النياسب ضمن كل كتلة ستعيد نفس الحساب الذي قامت به الكتل الأخرى، لانه لا يوجد ما يعرّف الكتلة.

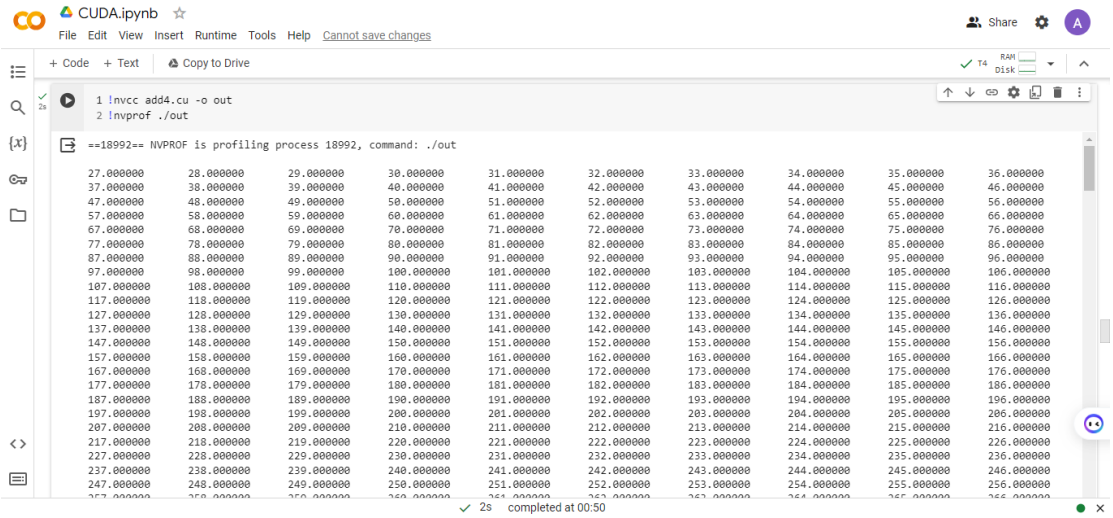
```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x;
    int stride = blockDim.x;

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

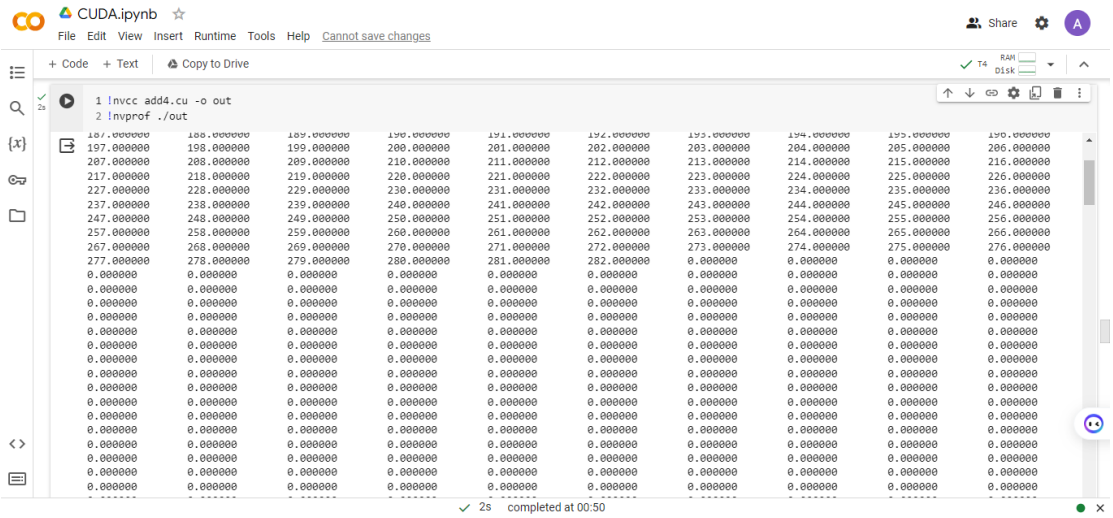
البرنامج الرابع add_Vec4.cu. يحتوي خطأ حيث أن الخطأ موجود ضمن القسم الخاص بتنفيذ النواة (kernel).

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

الخطأ هو أن النواة يتم تشغيلها بكتلة واحدة (gridDim.x = 1) و256 نيسب لكل كتلة (blockDim.x = 256). ونحن لدينا احجام المصفوفات a, b, out هو N=1000، وهو أكبر من عدد النياسب (256) ضمن الشبك grid. هذا يعني أنه لن تتم معالجة جميع عناصر المصفوفات بواسطة النياسب التي تم تشغيلها، مما يؤدي إلى نتائج غير صحيحة.

وهذا ما يمكن أن نلاحظه في حال تنفيذ البرنامج، سنجد انه فقط اول 256 قيمة تمت معالجتهم بشكل صحيح واما باقي القيم لم تتم معالجتها





يمكن حل هذه المشكلة بزيادة عدد النياسب ضمن الكتلة الواحدة بحيث تحتوي 10000 نيسب او زيادة عدد الكتل بحيث يكون لدينا مجموع نياسب الكتل اكبر او يساوي 10000، تجدر الإشارة الى انه في البرنامج الرابع زيادة عدد الكتل لن تؤدي الى تكرار التنفيذ كما في البرنامج الثالث، حيث انه ضمن ضمن تابع vector_add يوجد ما يميز الكتلة.

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
        out[tid] = a[tid] + b[tid];

}
```

تابع vector_add هو المثالي والأفضل كتنجيز من بين البرامج الأربعة المعطاة، حيث يسمح لنا بالاستفادة من التوازي بأكبر شكل ممكن، ويسمح لنا باستخدام اعداد كتل مختلفة، واعداد نياسب مختلفة.

2. للاستفادة من المعالجات المتوازية المتعددة (SMs) في وحدة معالجة الرسومات CUDA وتحسين أداء البرنامج، يمكننا تشغيل النواة (kernel) باستخدام كتل متعددة بدلاً من كتلة واحدة فقط. ومن خلال القيام بذلك، يمكننا توزيع عبء العمل عبر عدة SMs والسماح لهم بمعالجة البيانات بالتوازي.

تم تعديل البرنامج add_Vec4.cu، بحيث تم إضافة حلقة للتكرار على أحجام كتل مختلفة (32، 128، 512، و1024). لكل حجم كتلة، يتم حساب عدد الكتل المطلوبة بناءً على العدد الإجمالي للعناصر (N) وحجم الكتلة. بعد ذلك، يتم تشغيل النواة (kernerl) بالعدد المقابل من الكتل وحجم الكتلة الحالي ونلاحظ أن البرنامج يعمل بشكل كامل وصحيح.

```
1 !nvcc add4.cu -o out
2 !nvprof ./out
```

```
==360== NVPROF is profiling process 360, command: ./out
Executing kernel with 313 block, and each back contains 32 thread
27.000000   28.000000   29.000000   30.000000   31.000000   32.000000   33.000000   34.000000   35.000000   36.000000
37.000000   38.000000   39.000000   40.000000   41.000000   42.000000   43.000000   44.000000   45.000000   46.000000
47.000000   48.000000   49.000000   50.000000   51.000000   52.000000   53.000000   54.000000   55.000000   56.000000
57.000000   58.000000   59.000000   60.000000   61.000000   62.000000   63.000000   64.000000   65.000000   66.000000
67.000000   68.000000   69.000000   70.000000   71.000000   72.000000   73.000000   74.000000   75.000000   76.000000
77.000000   78.000000   79.000000   80.000000   81.000000   82.000000   83.000000   84.000000   85.000000   86.000000
87.000000   88.000000   89.000000   90.000000   91.000000   92.000000   93.000000   94.000000   95.000000   96.000000
97.000000   98.000000   99.000000   100.000000  101.000000  102.000000  103.000000  104.000000  105.000000  106.000000
107.000000  108.000000  109.000000  110.000000  111.000000  112.000000  113.000000  114.000000  115.000000  116.000000
117.000000  118.000000  119.000000  120.000000  121.000000  122.000000  123.000000  124.000000  125.000000  126.000000
127.000000  128.000000  129.000000  130.000000  131.000000  132.000000  133.000000  134.000000  135.000000  136.000000
137.000000  138.000000  139.000000  140.000000  141.000000  142.000000  143.000000  144.000000  145.000000  146.000000
147.000000  148.000000  149.000000  150.000000  151.000000  152.000000  153.000000  154.000000  155.000000  156.000000
157.000000  158.000000  159.000000  160.000000  161.000000  162.000000  163.000000  164.000000  165.000000  166.000000
167.000000  168.000000  169.000000  170.000000  171.000000  172.000000  173.000000  174.000000  175.000000  176.000000
177.000000  178.000000  179.000000  180.000000  181.000000  182.000000  183.000000  184.000000  185.000000  186.000000
187.000000  188.000000  189.000000  190.000000  191.000000  192.000000  193.000000  194.000000  195.000000  196.000000
197.000000  198.000000  199.000000  200.000000  201.000000  202.000000  203.000000  204.000000  205.000000  206.000000
207.000000  208.000000  209.000000  210.000000  211.000000  212.000000  213.000000  214.000000  215.000000  216.000000
217.000000  218.000000  219.000000  220.000000  221.000000  222.000000  223.000000  224.000000  225.000000  226.000000
227.000000  228.000000  229.000000  230.000000  231.000000  232.000000  233.000000  234.000000  235.000000  236.000000
237.000000  238.000000  239.000000  240.000000  241.000000  242.000000  243.000000  244.000000  245.000000  246.000000
247.000000  248.000000  249.000000  250.000000  251.000000  252.000000  253.000000  254.000000  255.000000  256.000000
257.000000  258.000000  259.000000  260.000000  261.000000  262.000000  263.000000  264.000000  265.000000  266.000000
```

+ Code + Text

```
1 !nvcc add4.cu -o out
2 !nvprof ./out
```

```
9777.000000  9778.000000  9779.000000  9780.000000  9781.000000  9782.000000  9783.000000  9784.000000  9785.000000  9786.000000
9787.000000  9788.000000  9789.000000  9790.000000  9791.000000  9792.000000  9793.000000  9794.000000  9795.000000  9796.000000
9797.000000  9798.000000  9799.000000  9800.000000  9801.000000  9802.000000  9803.000000  9804.000000  9805.000000  9806.000000
9807.000000  9808.000000  9809.000000  9810.000000  9811.000000  9812.000000  9813.000000  9814.000000  9815.000000  9816.000000
9817.000000  9818.000000  9819.000000  9820.000000  9821.000000  9822.000000  9823.000000  9824.000000  9825.000000  9826.000000
9827.000000  9828.000000  9829.000000  9830.000000  9831.000000  9832.000000  9833.000000  9834.000000  9835.000000  9836.000000
9837.000000  9838.000000  9839.000000  9840.000000  9841.000000  9842.000000  9843.000000  9844.000000  9845.000000  9846.000000
9847.000000  9848.000000  9849.000000  9850.000000  9851.000000  9852.000000  9853.000000  9854.000000  9855.000000  9856.000000
9857.000000  9858.000000  9859.000000  9860.000000  9861.000000  9862.000000  9863.000000  9864.000000  9865.000000  9866.000000
9867.000000  9868.000000  9869.000000  9870.000000  9871.000000  9872.000000  9873.000000  9874.000000  9875.000000  9876.000000
9877.000000  9878.000000  9879.000000  9880.000000  9881.000000  9882.000000  9883.000000  9884.000000  9885.000000  9886.000000
9887.000000  9888.000000  9889.000000  9890.000000  9891.000000  9892.000000  9893.000000  9894.000000  9895.000000  9896.000000
9897.000000  9898.000000  9899.000000  9900.000000  9901.000000  9902.000000  9903.000000  9904.000000  9905.000000  9906.000000
9907.000000  9908.000000  9909.000000  9910.000000  9911.000000  9912.000000  9913.000000  9914.000000  9915.000000  9916.000000
9917.000000  9918.000000  9919.000000  9920.000000  9921.000000  9922.000000  9923.000000  9924.000000  9925.000000  9926.000000
9927.000000  9928.000000  9929.000000  9930.000000  9931.000000  9932.000000  9933.000000  9934.000000  9935.000000  9936.000000
9937.000000  9938.000000  9939.000000  9940.000000  9941.000000  9942.000000  9943.000000  9944.000000  9945.000000  9946.000000
9947.000000  9948.000000  9949.000000  9950.000000  9951.000000  9952.000000  9953.000000  9954.000000  9955.000000  9956.000000
9957.000000  9958.000000  9959.000000  9960.000000  9961.000000  9962.000000  9963.000000  9964.000000  9965.000000  9966.000000
9967.000000  9968.000000  9969.000000  9970.000000  9971.000000  9972.000000  9973.000000  9974.000000  9975.000000  9976.000000
9977.000000  9978.000000  9979.000000  9980.000000  9981.000000  9982.000000  9983.000000  9984.000000  9985.000000  9986.000000
9987.000000  9988.000000  9989.000000  9990.000000  9991.000000  9992.000000  9993.000000  9994.000000  9995.000000  9996.000000
9997.000000  9998.000000  9999.000000  10000.000000 10001.000000 10002.000000 10003.000000 10004.000000 10005.000000 10006.000000
10007.000000 10008.000000 10009.000000 10010.000000 10011.000000 10012.000000 10013.000000 10014.000000 10015.000000 10016.000000
10017.000000 10018.000000 10019.000000 10020.000000 10021.000000 10022.000000 10023.000000 10024.000000 10025.000000 10026.000000
```

يمكننا ملاحظة أن البرنامج الآن ينفذ النواة (kernel) بأحجام مختلفة للكتل، وذلك باستخدام التوازي المتوفر على وحدة معالجة الرسومات.

على الرغم من أن العدد الإجمالي للنياسب هو نفسه، إلا أن تغيير عدد الكتل وحجم الكتلة قد يكون له تأثير على الأداء والسرعة والتنفيذ. فيما يلي بعض التأثيرات التي يجب مراعاتها:

1. استخدام الموارد: مع حجم كتلة أكبر، يكون لدينا عدد كتل أقل ولكن عدد أكبر من النياسب لكل كتلة. وهذا يمكن أن يؤدي إلى تحسين استخدام الموارد. على سبيل المثال، قد يسمح حجم الكتلة الأكبر بتقليل عدد الكتل التي تحتاج إلى جدولة، مما يؤدي إلى تحسين الأداء.

2. الحمل الزائد للمزامنة: يؤثر تغيير حجم الكتلة على عدد النياسب التي تحتاج إلى المزامنة داخل الكتلة. تميل أحجام الكتل الأكبر إلى أن تحتوي على عدد أكبر من النياسب التي تحتاج إلى المزامنة، مما قد يزيد من حمل المزامنة. على العكس من ذلك، قد يكون لأحجام الكتل الأصغر حمل مزامنة أقل بسبب وجود عدد أقل من النياسب داخل كل كتلة.

3. دمج الذاكرة: يمكن أن يؤثر حجم الكتلة على أنماط الوصول إلى الذاكرة. قد يسمح حجم الكتلة الأكبر بدمج أفضل للذاكرة، حيث تصل النياسب الموجودة داخل الكتلة إلى مواقع الذاكرة المتتالية. يمكن أن يؤدي ذلك إلى تحسين إنتاجية الذاكرة وتقليل زمن الوصول للذاكرة، مما يؤدي إلى أداء أفضل.

4. حمل التشغيل العام: يمكن أن يتأثر حمل التشغيل، بما في ذلك وقت إعداد kernel، وعمليات نقل الذاكرة، وتهيئة النياسب، بعدد الكتل. مع وجود عدد أكبر من الكتل، قد تزيد حمل التشغيل بسبب الإعداد الإضافي والمزامنة المطلوبة. ومع ذلك، إذا كانت عملية حساب kernel طويلة نسبيًا مقارنة بحمل التشغيل، فقد يكون التأثير على الأداء ضئيلًا.

الطالب: علي حسان سعيد.