المشاكل التي تواجه نظم التشغيل في إدارة موارد وحدة معالجة الرسوم(GPU)

د. م. وسيم السمارة[1]

الملخص

أصبحت وحدة معالجة الرسومات (GPU) منصة قوية جدا لتسريع الرسومات وأيضاً لمعالجة التطبيقات ذات البيانات المتوازية ومعقدة الحساب. وهذه الوحدة تتفوق بشكل كبير على المعالجات متعددة النواة التقليدية في الأداء والفعالية في الطاقة. وتزداد مجالات تطبيقها أيضاً على نطاق واسع في الأنظمة المدمجة وأنظمة الحساب ذات الأداء العالي. على الرغم من أن دعم أنظمة التشغيل لها ليس كافياً حيث تفتقر أنظمة التشغيل إلى الوحدات والتصميم وإلى الجهود المطبقة في إدارة موارد (GPU) من أجل بيئات متعددة المهام. وهذا البحث يقوم بتعريف نموذج لإدارة موارد (GPU) لتأمين الأساس لأبحاث أنظمة التشغيل التي تستخدم تكنولوجيا (GPU).

وبشكل خاص، سوف يتم تقديم المفاهيم الأساسية لإدارة موارد (GPU)، وقائمة التحديات التي تواجه أنظمة التشغيل وأيضاً سيتم تسليط الضوء على التوجهات المستقبلية لمجال البحث.

الكلمات المفتاحية: وحدة معالجة الرسومات(GPU)، إدارة الموارد، البرمجيات الاحتكارية.

[1] مدرس – قسم هندسة الحواسيب والأتمتة – كلية الهندسة الميكانيكية والكهربائية – جامعة دمشق.

Problems facing Operating Systems In Resource Management Of Graphics Processing Unit (GPU)

Dr. Wasim AL-Samara [(1)]

Abstract

The Graphics Processing Unit (GPU) has become a very powerful platform for graphics acceleration, and also for processing applications with parallel and complex computing data. This unit significantly outperforms traditional multi-core processors in performance and energy efficiency. Its application areas are also increasing widely in integrated and high-performance computing systems. Although, the support of operating systems for them is not sufficient , because operating systems lack the modules, design, and efforts applied in Resource Management (GPU) for multitasking environments.

This research defines a resource management model (GPU) to secure the foundation for operating system research using GPU technology, in particular, the basic concepts of resource management (GPU) and challenges facing operating systems and will also highlightedthe future directions of the research field

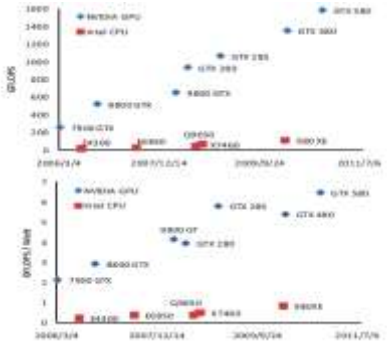
Keywords: GPU, resource management, proprietary software**.**

[(1)] Lecturer -Computer and Automation Engineering Department- Faculty of Mechanical and Electrical Engineering, Damascus University.

المقدمة:

يعد الأداء والطاقة من المحاور الأساسية في أنظمة التشغيل الحالية، فمنذ بداية عام (2000) تم تصنيع شريحة إلكترونية تشترك مع المعالج في معدل نبضات الساعة من أجل تحسين الأداء في خطوط إنتاجها. فكان معالج (Intel Pentium 4) المنتج التجاري الأول الذي وصل إلى معدل نبضات ساعة (3GHz) في عام (2002). وهذا الأداء يقاس بمدى الاستفادة من معدل نبضات الساعة للمعالج، ومع ذلك وبسبب مشاكل الطاقة والحرارة فإنها تقف عائق أمام تطوير تصميم الشريحة للاستفادة قدر المستطاع من معدل تردد نبضات الساعة للمعالج من أجل تكنولوجيا أحادية النواة الكلاسيكية. ومنذ عام (2000) إلى يومنا هذا، ازدادت التحسينات في الأداء وذلك من خلال الابتكارات الجديدة في مجال تكنولوجية متعددة النوى، بدلا من زيادة معدل نبضات الساعة وهذا التطور يشكل تحول كبير في زيادة الأداء مع انخفاض الطاقة المستهلكة. واليوم، وبعد الوصول إلى تقنية تعدد النوى للحصول على متطلبات الأداء العالي من أجل البيانات المتوازية الناشئة وتطبيقات الحساب المعقدة، وأيضا من أجل التطبيقات المضمنة مثل (المركبات المستقلة، والأنظمة الروبوتية)، حيث يتم الاستفادة من الطاقة المقدمة من معالجات متعددة النوى من أجل معالجة كمية كبيرة من البيانات التي يتم الحصول عليها من بيئات التشغيل الخاص بها [12,14].

أصبحت وحدة معالجة الرسومات (GPU) من الوحدات التي تبنى على فكرة تعدد النوى، ويوضح الشكل (1) التطورات الأخيرة لأداء وحدة (GPU) التي يتم تصنيعها من قبل شركتي (NVIDIA , INTEL). إن الأداء العالي لشريحة واحدة وفقاً لحالة الرسم البياني للـ (GPU) تتجاوز (1500 GFLOPS)، في حين أن المعالجات الميكروية التقليدية حوالي (100 GFLOPS) في أحسن الأحوال. وميزة الأداء العالي غير العادية للـ (GPU) تأتي نتيجة وجود المئات من نوى المعالجة المتكاملة على الشريحة. وأيضاً الـ (GPU) هو أفضل أداء من (CPU) لكل واط. وبشكل خاص، فإن الـ (GPU) يتفوق على الـ (CPU) في استهلاك الطاقة بحوالي (7) مرات



الشكل (1) التطورات الأخيرة لأداء وحدة (GPU)

هذا البحث يعرض عدة اتجاهات متعلقة بالتحديات التي تواجه نظام التشغيل في إدارة موارد (GPU). حالياً، ينقصنا النموذج الأساسي في إدارة موارد الـ (GPU) يمكن أن تكتشف في الجهود البحثية المستقبلية. إن نقص المعلومات في البرمجيات مفتوحة المصدر تعتبر قضية حرجة خاصة من أجل استكشاف تصميم النظم وتطبيق إدارة موارد (GPU) في هذا البحث، نقدم الأفكار والحلول الأولية لهذه المشكلة المفتوحة. ونثبت أيضاً أن البرمجيات مفتوحة المصدر هي جاهزة الآن من أجل الاعتماد عليها لأغراض بحثية [4].

ويتم تنظيم هذا البحث كالتالي، الافتراضات في هذا البحث تم عرضها في الفصل الثاني، والفصل الثالث يعرض الحالة الفنية لنموذج برمجة (GPU)، والقسم الرابع يتضمن نموذج إدارة موارد الـ (GPU) الأساسية من أجل نظام التشغيل. والفصل الخامس يقدم التحديات التي تواجه

أنظمة التشغيل من أجل إدارة موارد الـ (GPU)، تتضمن تقييم مبدئي لبرمجيات المصدر المفتوح الموجودة. وفي الفصل السادس يتم تقديم الحلول الحديثة لمعالجة المشاكل التي تواجه نظم التشغيل في إدارة موارد وحدة المعالجة الرسومية والآفاق المستقبلية.

2- الافتراضات التي يبنى عليها النظام:

نفترض في هذا البحث النظم المتنوعة التي تتركب من (CPUs) متعددة النوى ومن (GPUs). بنية الـ (GPU) المتعددة الموجودة اليوم، بينما العديد من المعالجات الميكروية التقليدية مصممة وفقاً لبنية (X86 CPU) وهي نفس البنية عبر عدة عقود، بنية (GPU) تغيرت عبر السنوات وشركة NVIDIA أصدرت معمارية (Fermi) وهي أول معمارية تدعم كل من البرامج الرسومية والحسابية. ويركز هذا البحث على بنية (Fermi)، ولكن هذا المفهوم أيضاً طبق على بنى أخرى. وعلى الرغم من أن شركة Intel قدمت بنية جديدة معتمدة على معالج (X86) تسمى (Sandy Bridge)، والتي يقوم بدمج (GPU) على الشريحة، (GPUs) المدمجة عل اللوحة هي الأكثر شيوعاً اليوم، بكل الأحوال سوف ندرس التأثيرات المترتبة على اللوحة وعلى شريحة الـ (GPUs) من وجهة نظر نظام التشغيل[1,7,8] .

من أجل نموذج (GPU) المدمج ، سنفترض أن الـ (GPU) و(CPU) يعملان بشكل غير متزامن. بكلام آخر يتم معالجة حالات الـ (GPU) وحالات (CPU) بشكل منفصل. ما إن تقوم برامج المستخدم بتنفيذ جزء من الكود على (GPU) للحصول على طلب مستعجل، يفرغ هذا الجزء من (CPU). برامج المستخدم يمكن أن تستمر في التنفيذ على (CPU) بينما الجزء من الكود البرمجي يعالج في (GPU)، وحتى يمكنه تنفيذ جزء آخر من الكود على الـ (GPU) قبل الانتهاء من الكود السابق. قائمة انتظار (GPU) تحتوي على طلبات الدفع، وتنفذ تعليماتها البرمجة في وقت لاحق عندما تكون (GPU) متاحة [15].

3- نموذج البرمجة:

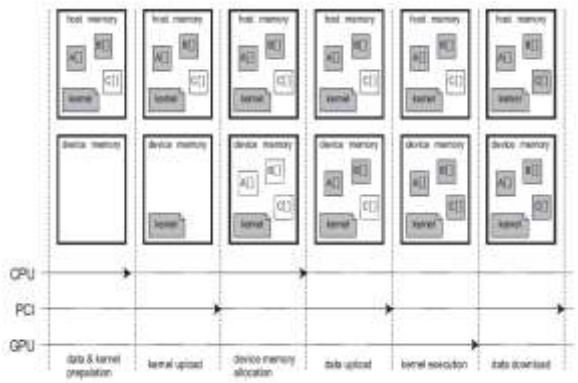
الـ (GPU) هو جهاز يقوم على تسريع جزء من الكود البرمجي بدلاً من تنفيذه على وحدة تحكم مثل (CPU). وبالتالي برامج المستخدم تبدأ بالتنفيذ على (CPU)، القطع البرمجية الدفعية، غالباً تشير إلى قناة (GPU)، للحصول على السرعة المطلوبة في التنفيذ. وهناك ثلاثة خطوات رئيسية لبرامج المستخدم التسارع على (GPU) وهي:

1- تخصيص الذاكرة: قبل كل شيء يجب أن يكون لبرامج المستخدم مساحة ذاكرة كافية من أجل عملية المعالجة. وهناك عدة أنواع من الذاكرة الموجودة على (GPU) وهي: المشتركة، المحلية، الشاملة، الثابتة، المتغيرة .

2- نسخ البيانات: البيانات الداخلة يجب أن تنسخ من الجهاز المضيف إلى ذاكرة الجهاز قبل أن تبدأ نواة الـ(GPU) بالعمل. وغالباً ما يتم نسخ البيانات الخارجة بالعكس من الجهاز إلى ذاكرة المضيف لإعادة نتيجة المعالجة إلى برامج المستخدم.

3- بدأ القناة: الكود البرمجي لتسارع الـ (GPU) يجب أن يدفع من الـ (CPU) إلى الـ (GPU) عند بدأ التشغيل، حيث أن الـ (GPU) بحد ذاته ليس وحدة تحكم، ويجب أن تكون إدارة مناطق عناوين ذاكرة الجهاز متاحة لكل طلب. البيانات المنسوخة ومراحل الدفع للقناة، من ناحية أخرى، تحتاج إلى الوصول للـ (GPU) لتحريك البيانات بين المضيف وذاكرة الجاهز، كود برنامج الـ (GPU) المدفع. الشكل (2) يوضح موجز عن تدفق التنفيذ لمصفوفة الضرب السريعة لـ (GPU)[13,14].

إلـى الوصـول للـ (GPU) لتحريـك البيانـات بـين المضيف وذاكرة الجاهز، كود برنامج الـ (GPU) المدفع. الشكل (2) يوضـح مـوجز عـن تـدفق التنفيـذ لمصـفوفة الضرب السريعة للـ (GPU).



الشكل (2) مثال على تدفق تنفيذ مصفوفة الضرب[ ]C = [ ]B×[ ]A

مثلاً: [ ]C = [ ]B×[ ]A. صورة نواة (GPU) يجب أن تحمـل علـى ذاكـرة المضـيف. اثنـان مـن مخزنـات المـدخلات، يجـب أن تحمـل القـيم الصـالحة مـن أجـل المعالجـة بينمـا مخـزن المخـرج يجـب أن يكـون فـارغ. عـادة يتم تحميل صورة القناة في البداية. ولأن الـ (GPU) يستخدم ذاكرة الجهاز للوصل إلى البيانات، يجب تخصيص مساحات البيانـات علـى ذاكـرة الجهـاز. وبعـد ذلـك يـتم نسـخ مخـازن المدخلات علـى مساحات البيانات المخصصـة علـى ذاكرة الجهاز عن طريق ناقل الـ (PCI). وبعدها تصبح البيانات الداخلـة جـاهزة علـى ذاكرة الجهـاز، تبـدأ نـواة الـ (GPU) بالتنفيـذ. وهـذا هـو تـدفق عـام لتسـريع البرنـامج علـى الـ (GPU). ويجـب أن تتوافـق لغـة البرمجـة المكتوبـة مـع واجهـة الجهـاز (API) للاتصـال مـع جـزء الجهـاز. وبعـض الأطر المستخدمة في برمجة (GPU):

- Open Graphics Language (OpenGL).
- Open Computing Language (OpenCL).
- Compute Unified Device Architecture (CUDA).
- HybridMulticore Parallel Programming (HMPP).

4- نموذج إدارة الموارد:

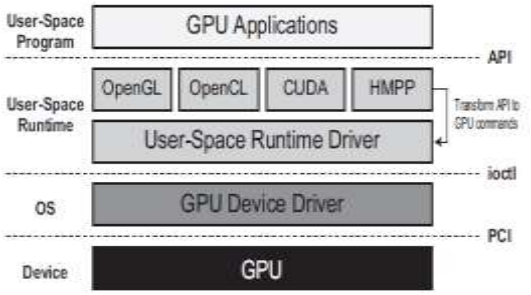
في هذا الفصل سيتم عرض النموذج الأساسي لإدارة موارد (GPU)، بشكل خاص على حزمة نظام (Linux)، وكمـا ذكرنـا فـي الفصـل الثـاني، النقـاش التـالي يفتـرض معمارية (NVIDIA Fermi)، ولكن أيضاً يمكن تطبيقها نظرياً على معظم بنى الـ (GPU) اليوم[3].

4-1 نظام المكدس:

بنية (GPU) تعرف كمجموعة من الأوامر التي تمكن الجهـاز مـن العمـل وتـوفير المسـاحة للمسـتخدم عنـد بـدأ التشغيل من أجل نسخ بيانات التحكم ودفعات القناة. يؤمن برنـامج تشـغيل الجهـاز البـدايات مـن أجـل بـرامج فضـاء المسـتخدم لإرسـالها إلـى (GPU). يولـد فضـاء المسـتخدم عـن بـدء التشـغيل واجهـة (API) محـددة لكتابـة برنـامج المستخدم. أمر استدعاء النظام (IOCTL) يستخدم غالباً للتفاعل بين برنامج تشغيل الجهاز وتوليد وقت التشـغيل. أوامر (GPU) هي التعليمات المختلفة التي في كود قنوات (GPU)، وهنـاك العديـد مـن أنـواع أوامـر الـ (GPU) برنـامج تشـغيل الجهـاز يرسـل أوامـر (GPU) لإدارة مـوارد (GPU) تتضـمن سـياق عمـل (GPU) ووحـدة إدارة ذاكـرة الجهـاز، فـي حـين أن عامـل وقـت التشـغيل يولـد أوامـر (GPU) لتـتحكم فـي سـير تنفيـذ بـرامج المسـتخدم، مـثلاً: نسخ البيانات وتدفق النواة [12.16].

الشكل (3) يوضح مكدس النظام في نموذج إدارة موارد (GPU) لـدينا. التطبيقـات تسـتدعي توابـع مكتبـة (API) التـي يوفرها برنـامج الإقـلاع. الواجهـة الأماميـة مـن برنـامج الإقـلاع تعتمـد علـى إطـار البرمجـة، التـي تحـول اسـتدعاء

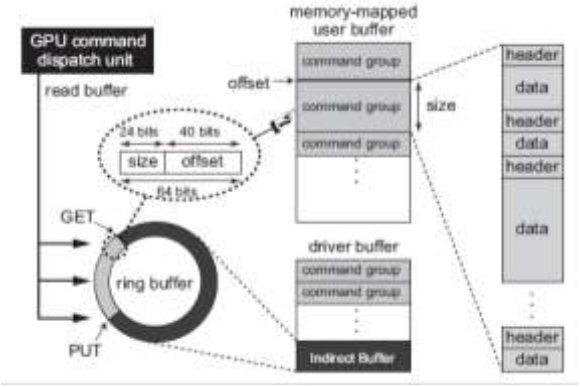
(API) إلـى أوامـر (GPU) التـي يـتم تنفيـذها عبـر (GPU).



الشكل (3) نظام المكدس من أجل معالج (GPU)

4-2 إدارة قناة (GPU):

برنامج تشغيل الجهاز يجب أن يدير قنوات (GPU). قناة الـ (GPU) هي كالواجهة التي تشكل جسور عبور بين ال (CPU) وبين سياقات (GPU)، وخاصةً عند إرسال أوامر (GPU) من الـ (CPU) إلى (GPU). وهو موصول مباشرةً إلى وحدة الإرسال داخل الـ (GPU)، الذي يقوم بتمرير أوامر (GPU) الواردة إلى وحدة المعالجة حيث يتم تنفيذ التعليمات البرمجية للـ (GPU). قناة الـ (GPU) هي الطريق الوحيد لإرسال الأوامر إلى جهاز الـ (GPU). وبالتالي يجب تخصيص قنوات الـ (GPU) لبرامج المستخدم. العديد من القنوات يتم وضعها في بنية الـ (GPU). على سبيل المثال، معمارية (NVIDIA Fermi) مدعومة بـ (128) قناة. وبما أن كل سياق يتطلب قناة واحدة على الأقل لاستخدام (GPU)، يسمح بوجود (128) من السياقات في نفس الوقت. وقنوات (GPU) مستقلة عن بعضها البعض، وتمثل مساحات عناوين منفصلة. ويوضح الشكل (4) كيفية تسليم أوامر (GPU) إلى (GPU) داخل القناة. قناة (GPU) تستخدم نوعين من المخازن في فضاء عنوان نظام التشغيل لتخزين أوامر الـ (GPU)[5,7,10].

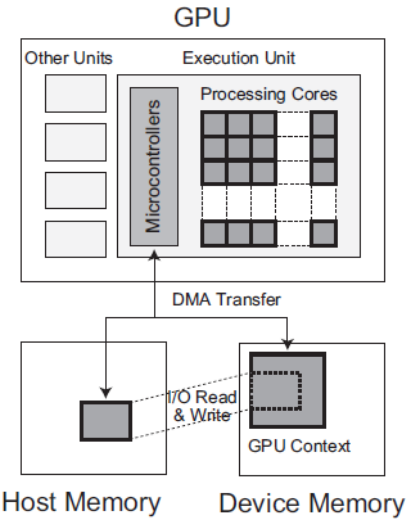


الشكل (4) كيفية تسليم أوامر (GPU) إلى (GPU) داخل القناة

4-3 إدارة سياق (GPU):

سياقات (GPU) تتكون من سجلات ذاكرة (I\O) وسجلات الأجهزة ، التي يجب تهيئتها من قبل برنامج تشغيل الجهاز في البداية. بينما مسجلات ذاكرة (I\O) يمكن أن يتم قراءتها والكتابة فيها بشكل مباشر من قبل برنامج مشغل الجهاز من خلال ناقل (PCI)، وتحتاج مسجلات الأجهزة إلى وحدات تفرعيه للـ (GPU) لكي يتم القراءة منها أو الكتابة فيها. وهناك طرق متعددة تؤمن الوصول إلي قيم السياق. القراءة من والكتاب في مسجلات ذاكرة (I\O) على الـ (GPU) هي الطريق الأكثر بساطة، ويتم إنشاء الاتصالات عبر الناقل (PCI) لجميع هذه العمليات. والـ (GPU) يوفر بديلاً لعدة وحدات أجهزة لنقل البيانات بين ذاكرة المضيف، وذاكرة الجهاز ومسجلات (GPU) بطريقة الدفعات.

يوضح الشكل (5) نموذج نظري لكيفية إدارة سياق الـ (GPU). عموماً، سياق الـ (GPU) يخزن على ذاكرة الجهاز. ويمكن أن يتم تخزينها على ذاكرة المضيف، والـ (GPU) يمكنه الوصول إلى كل من الذاكرتين، ولكن يفضل التخزين على ذاكرة الجهاز وذلك بسبب قضايا الأداء، على سبيل المثال، الـ (GPU) يصل إلى ذاكرة الجهاز بشكل أسرع من ذاكرة المضيف [4,11,12].

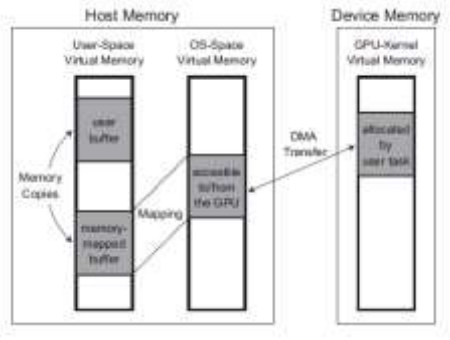


الشكل (5) وحدة إدارة السياق في الـ (GPU)

4-4 إدارة الذاكرة:

إدارة الذاكرة من أجل تطبيقات (GPU)ترتبط على الأقل مع ثلاثة فضاءات عناوين. إن برنامج المستخدم يبدأ تنفيذه على الـ (CPU) أولاً، المخزن المؤقت يتم إنشاءه ضمن ذاكرة افتراضية لفضاء المستخدم على ذاكرة المضيف. هذا المخزن يجب أن ينسخ الذاكرة الافتراضية لنظام التشغيل، ولأن برنامج مشغل الجهاز يجب أن يصل إليها لنقل البيانات إلى ذاكرة الجهاز. والهدف من نقل البيانات إلى ذاكرة الجهاز هو أنه يجب أن تكون متطابقة مع فضاء العناوين المخصص من قبل برنامج المستخدم مسبقاً من أجل الانسجام مع برنامج قناة (GPU). ونتيجة لذلك، يوجد ثلاثة فضاءات عناوين مرتبطة مع إدارة الذاكرة: 1- الذاكرة الافتراضية لفضاء المستخدم. 2- ذاكرة افتراضية لنظام التشغيل. 3- ذاكرة افتراضية لقناة (GPU).

ويبين الشكل (6) كيفية نسخ البيانات من المخزن المؤقت للمستخدم على ذاكرة المضيف إلى المخزن المؤقت المخصص على ذاكرة الجهاز [17].



الشكل (6) وحدة نسخ بيانات الجهاز المضيف

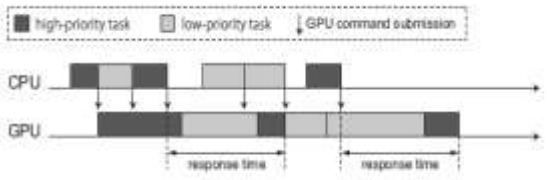
5- التحديات التي تواجه نظام التشغيل:

في هذا القسم، سنعرض أن التحديات التي تواجه نظام التشغيل في إدارة موارد (GPU). قائمة التحديات الواردة في هذا البحث يجب أن تكون مقدمة للمزيد من بحوث أنظمة التشغيل المتعلقة بتكنولوجيا (GPU) ومع ذلك نعتقد أن المناقشة التالية تعطي أفكار أين نحن وإلى أين سوف نذهب [6].

5-1 جدولة (GPU):

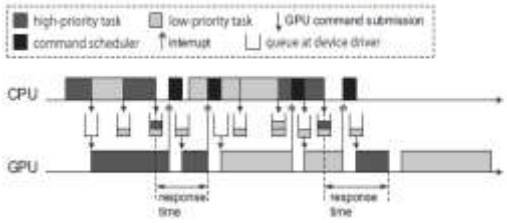
إن جدولة الـ (GPU) ربما من أهم التحديات أمام الاستفادة من (GPU) في بيئات متعددة المهام. بدون جدولة (GPU)، فإن برامج نواة الـ (GPU) تعمل بنموذج (FIFO)، ولأن وحدة إرسال الأمر للـ (GPU) تسحب مجموعة أوامر (GPU) في طلب الوصول. وبالتالي معالجة (GPU) تصبح غير شفعيه في التحسس القوي.

الشكل (7) يوضح مشكلة الاستجابة لمرة واحدة وذلك بسبب غياب الجدولة للـ (GPU). وبالتالي نحن بحاجة إلى الوصول لجدولة مناسبة للـ (GPU) لتجنب التداخل في الـ (GPU).



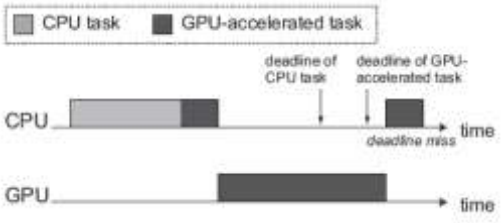
الشكل (7) معالجة (GPU) بدون جدولة

مفهوم التصميم: تصميم مجدولات (GPU) يقسم إلى فئتين: أول نموذج يطبق الجدولة على مستوى برنامج تشغيل الجهاز لترتيب مجموعة أوامر (GPU) المقدمة، نفترض أن تنفيذ السياقات للـ (GPU) تقدم بأوامر الـ (GPU). مجدولات الـ (GPU) في حالة الرسم مصممة على هذا النهج. على سبيل المثال قوائم انتظار الرسم البياني لمجموعة أوامر (GPU) في فضاء برنامج تشغيل الجهاز، وبتكوين الـ (GPU) لتوليد المقاطعات للـ (CPU) عند اكتمال التعليمات البرمجية التي يقوم بمعالجتها من قبل مجموعة أوامر الـ (GPU) بحيث المجدول يمكن أن يفعل ليقوم بإرسال مجموعة أوامر الـ (GPU) التالية. نقاط الجدولة تنشأ من هنا حدود مجموعة أوامر (GPU)، ترسل التلغراف وبخاصة مجموعة أوامر (GPU) وفقاً لأولوية المهمة، ويبين الشكل (8) أن المهمة ذات الأولوية الأعلى يمكن أن يتم الاستجابة لها بسرعة من قبل الـ (GPU) في حين يتم إدخال حمل إضافي لعملية الجدولة. وبالتالي هنا يحصل تمييز، ويتضح ذلك بأن الحمل الزائد أمر لا مفر منه لحماية تطبيقات الـ (GPU) المهمة من تضارب الأداء [14].

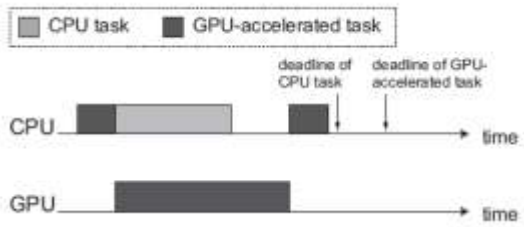


الشكل (8) معالجة (GPU) مع الجدولة

خوارزمية الجدولة: على افتراض أن مجدولات الـ (GPU) يتم تأمينها، ولكن السؤال ما هي خوارزمية الجدولة المطلوبة؟ سنفرض أنه على الأقل اثنين من خوارزميات الجدولة مطلوبة بسبب هرمية (CPU) والـ (GPU). أولاً: نحن نشدد على أن خوارزمية جدولة الـ (CPU) يجب أن تأخذ بعين الاعتبار وجود (GPU). ولاسيما في أنظمة الزمن الحقيقي، خوارزميات النهاية الميتة الكلاسيكية، مثل ( ) (Earliest Deadline First (EDF، ليست فعالة لوحدها. ويبين الشكل (9) مهمتين، واحدة تصل إلى (GPU) وهي مهمة مستعجلة، والأخرى ليست مهمة (CPU)، والجدولة في الـ (CPU) تستخدم خوارزمية (EDF). نعتبر أن مهمة الـ (CPU) هي تعيين المهمة ذات الأولوية العليا في بعض الوقت، بينما المهمة المستعجلة في الـ (GPU) تأخذ وقت معالجة كبير في الـ (GPU). ولأن الـ (GPU) لا تقوم بالدفع حتى يتم الانتهاء من مهمة (CPU)، الـ (GPU) تبقى في حالة خمول (idle) بينما مهمة الـ (CPU) يتم تنفيذها، وتبقى الـ (CPU) بحالة خمول (idle) بينما يتم تنفيذ المهمة المستعجلة في الـ (GPU). ومن أجل خوارزمية أكثر فعالية يجب أن يتم تطويرها لتوليد الجدولة المبينة في الشكل (10)، حيث أزمنة الـ (GPU) و(CPU) تتداخل بشكل فعال [14].



الشكل (9) جدولة (CPU) بالوقت الميت بوجود (GPU)



الشكل (10) جدولة (CPU) بوجود (GPU) بطريقة مطورة

جدولة نسخ البيانات: كما هو مبين في الشكل (2)، البيانات يجب أن يتم نسخها إلى ذاكرة الجهاز قبل أن يتم تنفيذ التعليمات البرمجية للـ (GPU) وغالباً المطلوب منها

نسح نتيجة المعالجة إلى ذاكرة الجهاز المضيف. إن الوقت المستغرق لإجراء نسخ البيانات تعتمد على حجم هذه البيانات، بالإضافة إلى ذلك، عملية نسخ البيانات ينتج عنها مساحات غير متناسقة، والتي تؤثر على الأداء والاستجابة من تطبيقات معينة. على وجه التحديد، بسبب نسخ البيانات بين المضيف ومساحات ذاكرة الجهاز تتم عبر الوصول المباشر (DMA)، ولا يؤمن برنامج تشغيل الجهاز أي طريقة لعملية الوصول المباشر (DMA)، لذلك جدولة الـ (GPU) يجب أن تشارك في نسخ البيانات بالإضافة إلى تنفيذ سياقات الـ (GPU) [14].

5-2 مفهوم العنقدة في الـ (GPU):

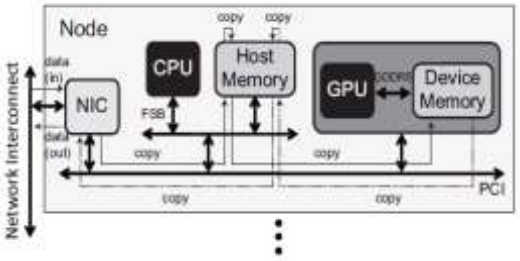
من ناحية أخرى تتضمن تحديات الأبحاث دعم لعنقودية (GPUs) المتعددة. عنقودية (GPU) هي التكنولوجية الرئيسة لاستخدام (GPU) في تطبيقات (HPC). حالياً، نقوم بتطوير هذه التكنولوجيا بالاعتماد على نموذج إدارة موارد الـ (GPU) ومخططات الجدولة للـ (GPU) لتأمين الدعم من الدرجة الأولى لمراكز البيانات التفاعلية المعتمدة على (GPU)، أجهزة الكمبيوتر العملاقة، والأنظمة الالكترونية الفيزيائية. هذه التطبيقات هي البيانات المتعددة وكذلك المعالجة المتعددة. وبالتالي يتأثر الأداء باتصال البيانات مع بعضها عبر الـ (GPUs) .

البنية العنقودية للـ (GPU) هي بشكل عام هرمية. كل عقدة تركب من عدد قليل من الـ (GPUs) العنقودية على اللوحة. والعديد من هذه العقد تكون إضافية في النظام. ولأن هذين النوعين من العقد تستخدم تقنيات مختلفة، فإنه يجب أن يتم توفير الدعم لأنظمة مختلفة [15].

عنقودية (GPU) على اللوحة: إدارة الـ (GPUs) على اللوحة يمكن أن يكون إما في فضاء المستخدم عند بدء التشغيل أو من خلال نظام التشغيل. وهي عادةً تكون مهمة التطبيق الذي يحدد أي (GPU) يجب استخدامها في المعالجة. نسخ البيانات بين(GPUs) يمكن أن يتم التعامل معها في فضاء المستخدم بنسخ البيانات عن طريق ذاكرة المضيف. بكل الأحوال، فإن نظام التشغيل هو المسئول عن تكوين الـ (GPUs)، إذا كانت سرعة نسخ البيانات المباشرة بين فضاءات اثنين من ذاكرة الجهاز المختلفة بطريقة الناقل (PCI) أو واجهة (SLI) هي المطلوبة. على سبيل المثال، مواصفات الـ (CUDA 4.0) تتطلب مثلاً واجهة للاتصال بين البيانات، غالباً يشار إليها عبر الـ (GPU) مباشرةً في عنقودية (GPU)، وبالتالي الجدولة يجب أن تشارك في نسخ البيانات من جهاز إلى جهاز بالإضافة إلى تنفيذ سياقات الـ (GPU) ونسخ البيانات بين المضيف وذاكرة الجهاز.

عنقودية (GPU) الشبكية: إن إدارة الاتصال بين الـ (GPUs) المتعددة عبر الشبكة أكثر تحدياً. الـ (GPU) التي تعتمد على تطبيقات (HPC) يمكن تجميعها لاستخدام آلاف العقد. والاتصال بين البيانات عبر الشبكة سوف يسبب مشكلة عنق الزجاجة من أجل زيادة أداء عنقدة (GPU) في عدد من العقد. الشكل (11) يوضح كيفية إرسال واستقبال البيانات عبر كرت الشبكة (NIC) و(GPU) في النموذج الأساسي عموماً، عندما الـ (NIC) تستقبل البيانات، فإن برنامج تشغيل الجهاز (NIC) ينقل هذه البيانات إلى ذاكرة المضيف بطريقة (DMA) [15,9].

وبرنامج تشغيل جهاز (GPU) يحتاج بعد ذلك إلى نسخ البيانات إلى ذاكرة الجهاز، ولكن فضاء العناوين المرئية للـ (NIC) وللـ (GPU) تختلف، وعادة ما يتم تطوير برنامج تشغيل الجهاز بشكل فردي. لذلك يقوم برنامج تشغيل الـ (GPU) بنسخ البيانات إلى مساحة أخرى من ذاكرة المضيف تكون مرئية للـ (GPU).



الشكل (11) كيفية إرسال واستقبال البيانات عبر كرت الشبكة (NIC) و(GPU)

5-3 (GPU) الافتراضية:

هي تقنية مفيدة اعتمدت على نطاق واسع في العديد من مجالات التطبيق من أجل عزل المستخدمين للنظام، وتجعل النظام مركب ويمكن الاعتماد عليه. وبالتالي فإن ال (GPUs) الافتراضية تؤمن نفس الفوائد لنظم الـ (GPU) المستعجلة. وتعم هذه التقنية في وقت التشغيل، VMMs، وإدارة (I\O) [6].

5-4 إدارة جهاز الذاكرة (GPU):

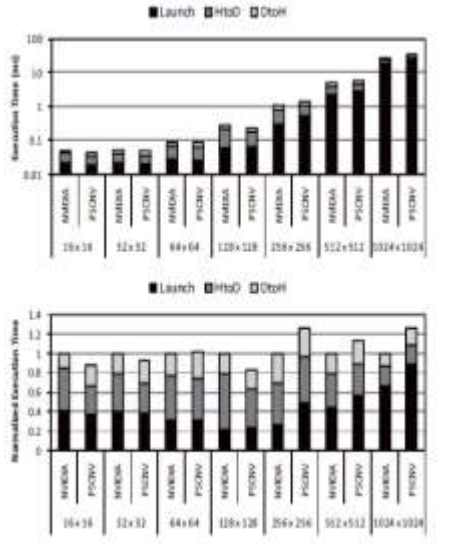
مساحات الذاكرة المخصصة من قبل برنامج المستخدم مغلقة. وهي لن تصبح متاحة للبرامج المختلفة ما لم يتم تحريرها بشكل واضح، النتائج في حجم الذاكرة المخصصة تحدد بحجم ذاكرة الجهاز. وهذا غير فعال في نموذج إدارة الذاكرة. وغالباً (GPU) تدعم الذاكرة الافتراضية لعزل مساحات العناوين بين قنوات (GPUs)، أنظمة التشغيل يجب أن تستثمر هذه الذاكرة الافتراضية ظاهرياً لتوسيع فضاء ذاكرة الجهاز المخصصة، كما يعتمد عليها ذاكرة المضيف من خلال وحدات إدارة الذاكرة (MMUs). يجب أن نتمكن من تأسيس نموذج هرمي لذاكرة الجهاز، وذاكرة المضيف [7].

5-5 التنسيق مع برنامج الإقلاع (وقت التشغيل):

عمليات (GPU) يتم التحكم بها عن طريق مجموعة أوامر (GPU) من برامج فضاء المستخدم. على سبيل المثال، بدأ قناة (GPU) ونسخ البيانات بين ذاكرة المضيف والجهاز يتم تشغيلها من قبل مجموعة محددة من أوامر الـ (GPU). ومع ذلك، فإن نظم التشغيل لا تدرك ما هي أنواع أوامر (GPU) التي تصدر من برامج فضاء المستخدم. أنها تتحكم فقط بنقاط التخزين الحلقية لموقع الذاكرة حيث مجموعة أوامر (GPU) تخزن ببرنامج بدء التشغيل بفضاء المستخدم بحيث أن الـ (GPU) يمكن أن ترسل لهم [7].

5-6 تطبيق وتنفيذ المصدر المفتوح:

إن تطور أدوات مفتوحة المصدر هو واجب أساسي لتبادل الأفكار حول تنفيذ النظم وتطوير البحوث. نظام (Linux) على سبيل المثال هو نظام مفتوح المصدر يستخدم في بحوث أنظمة التشغيل. وتجدر الإشارة إلى أن أداء البرمجيات مفتوحة المصدر تشكل منافسة مع برمجيات (NVIDIA) الاحتكارية، على الرغم من أن استخدامها يقتصر على تطبيقات الرسوم. ويبين الشكل (12) مقارنة بين أداء (NVIDIA) ذات المصدر المغلق أي الملكية الاحتكارية وبين (PSCNV) مفتوحة المصدر من أجل عملية ضرب مصفوفة صحيحة ذات الأحجام متغيرة [8].



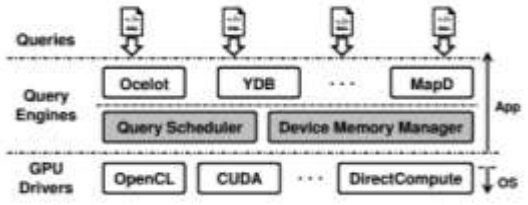
الشكل (12) مقارنة بين أداء (NVIDIA) ذات المصدر المغلق أي الملكية الاحتكارية وبين (PSCNV) مفتوحة المصدر من أجل عملية ضرب مصفوفة صحيحة ذات الأحجام متغيرة

6- الحلول الحديثة للمشاكل التي تواجه نظم التشغيل في إدارة موارد وحدة معالجة الرسوم:

أ- استخدام تصميم MultiQx-GPU:

تصميم MultiQx-GPU يعتمد على مبدأين أساسين. الأول وهو Versatility (تعدد الاستخدامات): التقنيات المستخدمة في الأنظمة يجب أن تكون قادرة على التعامل مع قواعد بيانات مختلفة وإطارات عمل مختلفة لإدارة موارد GPU. وتقنيات قواعد بيانات GPUs ما تزال في تطور مستمر. والأنظمة المختلفة يكون لديها محركات استعلام مختلفة. فطريقة تصميم MultiQx-GPU تسمح بسهولة إدارة كل هذه الاختلافات. ولكن هذا يتطلب من هذا التصميم أن يقوم بتسجيل الخصائص الأساسية لمعالجة استعلامات GPU ليقوم ببناء إطار عمل متكامل يتوافق مع GPU المستقبلية.[18]

المبدأ الثاني الأساسي وهو High-Efficiency الكفاءة العالية: مازال عتاد GPU لا يدعم بشكل مباشر تعدد المهام، والمبدأ الذي يعتمد فيه للقيام بذلك هو نفس مبدأ الـ CPU والذاكرة الافتراضية VM وتبديل السياق. هذا يجبرنا في تصميم الـ MutliQx-GPU لزيادة طبقة إضافية من مستوى البرمجيات لدعم الاستعلامات المتعددة. الشكل (13) يوضح موضع MultiQx-GPU في قاعدة بيانات الـ GPU الكلية. هذا التصميم لا يغير من واجهات البرمجة المستخدمة سابقاً، وهو يبقى بشكل كامل في فضاء التطبيقات، ولا يعتمد على أي وظيفية في مستوى نظام التشغيل. وبذا يمكن أن تسمح لنا بالانتقال من نظام محرك استعلام إلى أخر وبين إطار عمل GPUs المختلفة مثل CUDA و OpenCL و DirectCompute ليسمح للـ GPU بمشاركة موارده.[18]

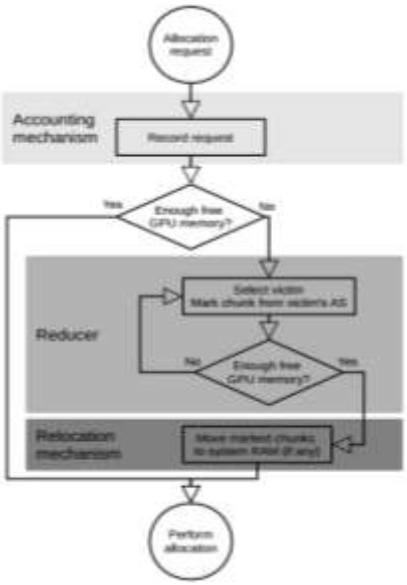


الشكل (13) يوضح موضع MultiQx-GPU في قاعدة بيانات الـ GPU الكلية. (حيث الصندوقين المظللين هما العنصرين المضافين في تصميم MultiQx-GPU)

ب- تقنية تبادل GPU (GPUswap):

حيث هذه التقنية تتألف من 3 عناصر أساسية مبينة في الشكل (14). 1- آلية المحاسبة وهي التي تتتبع المعلومات عن كل برنامج والذاكرة المحجوزة لأجله، 2- مخفض الذي يقوم بناءً على المعلومات التي يؤمنها المحاسب له بتقرير أي جزء من الذاكرة سيتم نقله إلى الذاكرة العشوائية RAM الخاصة بالنظام، 3- آلية تبديل والتي تنفذ من قبل المخفض حسب قراره. ففي كل مرة تطبيق يطلب ذاكرة GPU، آلية المحاسبة تسجل ملاحظات عن هذا الطلب لتتبع الذاكرة المحجوزة لكل برنامج من ذاكرة GPU فإذا لم يكن هنالك مساحة حرة كافية في الذاكرة لتخدم الطلب، المخفض يقوم بتقرير أي جزء من الذاكرة يجب أن يعاد حجزه في ذاكرة RAM الخاصة بالنظام ليفرغ ذاكرة GPU لتخديم الطلب. في النهاية، آلية إعادة حجز الذاكرة تقوم بالعمل الفعلي بإعادة الحجز.[19] والعكس بالعكس، في أي وقت برنامج يحرر جزء من ذاكرة GPU، GPUswap تختار جزء من الذاكرة مناسب ليعاد حجزه في ذاكرة GPU بعد أن كان موضوع مؤقتاً في ذاكرة RAM الخاصة بالنظام. GPUswap هي بشكل كامل شفافة بالنسبة للبرنامج وتسبب تأخير بسيط في التطبيقات التي تعتمد على

GPU، وهـي مصـممة للـتحكم فـي سـياق نـواة نظـام التشغيل، بغض النظر عن عمل GPU.[19]



الشكل (14) العناصر الأساسية لتقنية GPUswap

الآفاق المستقبلية:

في الوقت الحالي GPUs لا تدعم الآلية اللازمة لنظام التشـغيل لإدارة مـوارد GPU بشـكل كامـل. ورغـم الحلـول الحاليـة فإن هذا البحث يعتبر بداية ومفتاح للانطلاق في حلول جديدة ممكن أن تحقق فائدة أكبر وخاصة في حال الاعتماد على التقنيات المفتوحة المصدر وذلك لإدارة موارد GPU بالشكل الأمثل.

References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In Proceedings of the ACM International Symposium on High Performance Distributed Computing, pages 165–174, 2008.

[2] M. Bautin, A. Dwarakinath, and T. Chiueh. Graphics Engine Resource Management. In Proceedings of the Annual Multimedia Computing andNetworking Conference, 2008.

[3] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. In Proceedings of the IEEE
International Parallel and Distributed Processing Symposium, 2010.

[4] H. Cho, B. Ravindran, and E.D. Jensen. Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems. IEICE Transactions on Communications, 85:807–813, 2002.

[5] Linux Open-Source Community. Nouveau Companion 44. http://nouveau.freedesktop.org/.

[6] Linux Open-Source Community. Nouveau Open-Source GPU Device Driver. http://nouveau.freedesktop.org/.

[7] M. Dowty and J. Sugeman. GPU Virtualization on VMware's Hosted I/O Architecture. ACM SIGOPS Operating Systems Review, 43(3):73–82, 2009.

[8] G. Elliott and J. Anderson. Real-Time Multiprocessor Systems with GPUs. In Proceedings of the International Confrence on Real-Time and
Network Systems, 2010.

[9] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU Accelerated Storage System. In Proceedings of the ACM InternationalSymposium on High Performance Distributed Computing, pages 167–178, 2010.

[10] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling Task Parallelism in the CUDA Scheduler. In Proceedings of the Workshop onProgramming Models for Emerging Architectures, pages 69–76, 2009.

[11] V. Gupta, A. Gavrilovska, N. Tolia, and V. Talwar. GViM: GPUacceleratedVirtual Machines. In Proceedings of the ACM Workshop
on System-level Virtualization for High Performance Computing, pages 17–24, 2009.

[12] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus:Coordinated Scheduling for Virtualized Accelerator-based Systems. In Proceedings of the USENIX Annual Technical Conference, 2011.

[13] Intel. IntelMicroarchitecture Codename Sandy Bridge.http://www.intel.com/.

[14] S. Kato and Y. Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In Proceedings of the IEEE Real-Time Systems Symposium, pages 459–468, 2009.

[15] S. Kato, Y. Ishikawa, and R. Rajkumar. CPU Scheduling and Memory Management for Interactive Real-Time Applications. Real-Time Systems, 2011.

[16] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In Proceedings of
the IEEE Real-Time and Embedded Technology and Aplications Symposium, pages 191–200, 2011.

[17] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In Proceedingsof the USENIX Annual Technical Conference, 2011.

[18] Wang K, Zhang K, Yuan Y, Ma S, Lee R, Ding X, Zhang X. Concurrent analytical query processing with GPUs. Proceedings of the VLDB Endowment. 2014.

[19] Kehne J, Metter J, Bellosa F. GPUswap: Enabling oversubscription of GPU memory through transparent swapping. InACM SIGPLAN Notices 2015 Mar 14 (Vol. 50, No. 7, pp. 65-77).

Received	2017/11/01	إيداع البحث
Accepted for Publ.	2017/12/11	قبول البحث للنشر