ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Déduplication extrême d'OS avec SIDUS : un petit pas pour la reproductibilité ?

Emmanuel Quemener

Ecole Normale Supérieure de Lyon,
Centre Blaise Pascal - 46, allée d'Italie
69007 Lyon - France
Emmanuel.Quemener@ens-lyon.fr

Résumé
Lors d'une expérience numérique menée sur un système informatique complet, comment s'assurer que le composant «*Operating System*» reste bien identique, d'un redémarrage à l'autre sur une même machine, ou au même instant sur un ensemble de nœuds ? Initialement, ce n'est pas en réponse à cette épineuse difficulté que SIDUS a été développé : SIDUS l'a été essentiellement pour simplifier l'administration de la majorité des plateaux techniques du Centre Blaise Pascal à l'ENS-Lyon.
En effet, SIDUS simplifie la gestion de nœuds de calcul, de stations de travail ou de nouveaux matériels, limite l'empreinte du stockage de l'OS sur ses machines, voire offre à un utilisateur un environnement scientifique complet en quelques secondes. Scalable et polyvalent, SIDUS a comme principales propriétés : l'unicité de son socle sur un serveur et l'usage des ressources locales de chaque client.
Dès lors, si seules les ressources locales sont exploitées et si ce socle technique unique demeure inchangé, alors la comparaison entre les expériences numériques que nous menons gagnent en pertinence. A contrario, si la reproductibilité fait défaut, les origines sont à rechercher ailleurs que dans l'OS.
Dans un cadre opérationnel, nous aborderons deux thèmes : l'un présentera une variabilité issue de contexte matériel et comment la limiter. La seconde s'attachera à montrer comment SIDUS, par sa flexibilité, permet une comparaison rapide et pertinente de cartes graphiques dans un contexte de calcul scientifique.

Mots-clés : déduplication, variabilité temporelle, GPU, reproductibilité.

---

1. Introduction

Nous aborderons, en premier lieu, la variabilité inhérente à celle de toute expérience numérique. Puis nous chercherons des solutions permettant d'extraire la partie «*Operating System*» du système informatique dans son ensemble. Ensuite, SIDUS, la solution que nous proposons, sera décrite à l'aide de questions fondamentales, notamment comment l'installer voire l'administrer simplement. Enfin, au travers d'expériences dans un cadre opérationnel, nous illustrerons d'un côté la nature de cette variabilité, ses sources et comment les combattre pour gagner

en pertinence, de l'autre comment SIDUS permet de comparer les processeurs graphiques utilisés en calcul scientifique.

2. La reproductibilité sous le regard d'un expérimentateur

2.1. Rapide état des lieux de l'informatique scientifique

Chaque traitement, chaque simulation et même chaque visualisation lancés sur un système informatique sont des «expériences numériques». Depuis plus de 30 ans, depuis l'extinction massive des calculateurs analogiques, nous pensions vivre dans le fabuleux monde du déterminisme numérique. En fait, une rapide introspection sur le traitement des opérations en virgule flottante modère ce point de vue[3] : des erreurs d'arrondi affectent une majorité de nos calculs et les différentes approches de parallélisme les rendent de moins en moins prédictibles. A cela s'ajoute désormais un nouveau type de variabilité : les processeurs disposent de plus en plus de cœurs, les systèmes d'exploitation nécessitent de plus en plus de processus pour conserver actifs, en totalité, leurs services courants ; les réseaux utilisent des méthodes d'accès non déterministes à leurs médias de communication, les processeurs embarquent trois niveaux de cache et des douzaines d'ALU (unités arithmétiques et logiques) ou d'UC (unités de contrôle) ; les fréquences des processeurs voire de la mémoire changent continuellement. Pour «couronner le tout», les processeurs récents n'ont plus qu'une seule contrainte : respecter leur enveloppe thermique maximale (la TDP ou *Thermal Design Power*) ou des stratégies d'économie d'énergie au détriment de tout le reste !

«Cerise sur le gâteau», les applications tournent parfois sur des machines différentes (de la douzaine à des milliers) pour lesquelles il est très difficile de s'assurer qu'elles partagent exactement le même système d'exploitation.

2.2. Qu'espérer d'une infrastructure devenue complexe ?

Ce panorama établi, comment pouvons-nous réduire autant que possible la variabilité de notre système dans son ensemble, la variabilité dans les résultats eux-mêmes, ou, c'est le cas de notre étude, la variabilité sur les durées d'exécution sur nos systèmes ? Comment s'assurer que le système reste identique d'un redémarrage à l'autre (sur une même machine) ou que toutes les machines disposent, au même instant, du même système au bit près ?

Dans le même ordre d'idée, comment pouvons-nous réduire l'effort de portage de notre station de travail ou de petits clusters départementaux vers les grosses machines ?

2.3. De l'émergence d'un système unique

2.3.1. Le pragmatisme d'une distribution complète

La première étape relève du choix de «la» distribution offrant le plus large panorama possible de logiciels scientifiques précompilés et préconfigurés afin d'éviter à tout prix toutes inutiles recompilation et réinstallation du socle de nos codes : la distribution Debian (ou toute autre distribution aussi complète que la Debian) est une approche pragmatique. L'assurance qualité de Debian est aussi une des plus tatillonnes qui soit : ce serait dommage de ne pas en profiter et de réinventer la roue ! A titre de petite expérience, nous invitons les sceptiques à installer *from scratch* le logiciel de chimie quantique Abinit[7].

2.3.2. De l'instantané à l'instance unique

La seconde étape consiste alors à fournir ce système d'exploitation issu de cette distribution à nos clients. Tout d'abord, il nous faut figer cet OS pour le retrouver exactement dans le même état à une date ultérieure. Ces solutions de «photographie» de système sont légion, ce sont essentiellement des solutions de sauvegarde (MondeRescue, SystemImager, CloneZilla) ou sont proches du monde HPC comme Kadeploy. Ensuite, une fois les systèmes sauvegardés, ces outils se proposent de restaurer un système sous forme d'une ou plusieurs partitions. Enfin, le redémarrage de toutes les machines ainsi fraîchement installées : toute variabilité est absente d'un système à l'autre.

Toutefois, la gestion de ce type d'infrastructure devient rapidement très lourde dès lors que le système devient imposant. A titre d'illustration, une distribution *Debian Science* complète occupe un espace de quarantaine de giga-octets. Même avec d'improbables taux de compression (de 90%), démarrer une centaine de machines va générer 400 Go de trafic pour le serveur, avec à la clé vingt bonnes minutes d'écriture sur chaque disque dur, donc une demi-heure d'immobilisation, sans même évoquer la partie création de la partition «maître»...

Une autre approche vise à s'affranchir de la partie «stockage local» en offrant un système directement par le réseau. Là, le protocole iSCSI est parfait pour offrir un système en «mode bloc». Le boot PXE permet, par TFTP, la récupération d'un noyau Linux, d'une séquence de démarrage InitRD. Ensuite, le démarrage permet le montage distant par un client iSCSI au InitRD. Les «cibles» iSCSI existent suivant deux implémentations sous Linux. Pour leur duplication, les mécanismes d'instantanés sont à trouver auprès des gestionnaires de volumes logiques comme LVM, ZFSonLinux voire BTRFS. Ainsi, pour chaque instantané, chaque «image système» est clonée puis va servir chaque nœud individuellement par iSCSI. Cette méthode est efficace, mais exige la maîtrise d'un environnement *chrooté* pour la création du système de base. Généralisée sur une centaine de nœuds, elle devient difficile à dimensionner : ce n'est pas tant le réseau mais le serveur qui se retrouve victime d'un fabuleux *boot storm*. Tout est donc à recommencer à chaque redémarrage pour s'assurer de la cohérence avec l'image originelle.

La solution que nous proposons dépasse toutes ces limitations en exploitant le partage réseau le plus simple qui soit : NFS. C'est un environnement configuré une seule et unique fois et déployé sur tous les clients, où chaque modification est instantanément propagée sur toutes les autres : SIDUS pour *Single Instance Distributing Universal System*, n'est pas qu'un projet. Il est pleinement opérationnel depuis 4 années au Centre Blaise Pascal (maison de la simulation lyonnaise) sur une centaine de machines permanentes et au Pôle Scientifique de Modélisation Numérique depuis 2 ans (un des méso-centres lyonnais) sur plus de 330 nœuds.

2.4. Qu'allons nous présenter ?

Dans une première partie, nous détaillerons SIDUS par le truchement de réponses à des questions fondamentales. Qu'est-ce que SIDUS ? Où & quand est-il exploité ? A qui peut-il servir ? Comment s'installe-t-il ? Comment s'administre-t-il ?

3. SIDUS : un couteau suisse ?

SIDUS est l'acronyme de *Single Instance Distributing Universal System*.
Son origine latine «d'ensemble de corps stellaires» est une allégorie. Ainsi SIDUS partage le même système d'exploitation avec des machines aux ressources matérielles différentes. Tout

comme les étoiles d'une constellation, aux observables physiques différentes, appliquent les mêmes mécanismes de fusion nucléaire. SIDUS a comme principales propriétés :
– son unicité de configuration : deux machines démarrant sous SIDUS ont exactement le même système d'exploitation ;
– son exploitation des ressources locales : les processeurs et mémoire vive sollicités sont ceux de la machine locale.

SIDUS n'est donc :
– ni LTSP pour *Linux Terminal Server Project* : LTSP propose une gestion simplifiée de terminaux légers en offrant un accès X11 ou RDP à un serveur. Ce dernier supporte ainsi toute la charge de traitement. A contrario, SIDUS exploite entièrement (ou à discrétion de l'utilisateur) toute la machine qui s'y raccroche. Seul le stockage du système d'exploitation est déporté sur des machines tierces ;
– ni FAI pour *Fully Automatic Installation* : FAI et Kickstart proposent une installation complète simplifiée permettant de limiter voire d'éliminer toute action de l'administrateur. A contrario, SIDUS propose un système unique dans un arbre intégrant à la fois le système de base et toutes les applications installées manuellement ;
– ni un LiveCD sur réseau : un LiveCD démarre un système minimaliste, nécessairement figé. Il est toujours possible de créer son propre LiveCD mais c'est une opération lourde. Avec SIDUS, il est possible d'installer à la volée sur tous ses clients un nouveau composant instantanément ou de reconfigurer l'instance ;
– ni monolithique : dans le cadre de formations en informatique, trois solutions s'offrent aux utilisateurs : exploiter les machines mises à disposition, utiliser leur équipement personnel, installer un environnement virtuel complet figé au téléchargement et donc difficilement modifiable. A contrario, SIDUS offre un environnement unique aisément configurable ;
– ni original : SIDUS exploite des services disponibles sur n'importe quelle distribution : DHCP, PXE, TFTP, NFSroot, DebootStrap, AUFS. Ces quelques mots clés permettant d'installer SIDUS. Il utilise en outre des astuces de distributions de LiveCD et fonctionne sur la distribution Debian depuis sa version Etch.

En définitive, SIDUS est :
– universel : toutes les plates-formes x86 ou x86-64 fonctionnent instantanément ;
– efficace : installation en quelques dizaines de minutes, démarrage en quelques secondes ;
– économe : à l'origine, 1 cœur, 1Go de RAM, 40Go d'espace disque, et un réseau (GigaBit) Ethernet
– scalable : éprouvé sans difficulté sur une centaine de nœuds, en production maintenant sur plus de 330 nœuds ;
– robuste : avec un réseau standard routé, des *uptime* de plusieurs mois dans sa version COMOD (pour *Compute On My Own Device*) ;
– polyvalent : avec la Debian et tout «science», un excellent socle pour toutes les sciences.

3.1. Où & Quand , ou quelles *success stories* ?

Sur quels types d'équipements SIDUS peut-il se déployer ? C'est là que s'illustre sa polyvalence :
– **Sur les postes utilisateurs** : machines mutualisées ou stations de travail individuelles ? Tout a commencé avec une douzaine de clients légers Neoware gonflés en mémoire et overclockés. Ce sont maintenant plus de 20 machines équipées de cartes graphiques différentes et offertes à 300 utilisateurs de l'ENS-Lyon ;
– **Sur les nœuds de cluster** :

  - après un démonstrateur de 24 nœuds en mars 2010 au Centre Blaise Pascal, SIDUS sert actuellement 76 nœuds permanents, sur 3 architectures matérielles différentes,
  - après une période de qualification d'une année au Pôle Scientifique de Modélisation Numérique sur quelques dizaines de nœuds, SIDUS équipe maintenant plus de 330 nœuds, dont le nouvel équipement Equip@Meso,
  - après seulement quelques heures de configuration à l'Institut de Génomique Fonctionnelle de Lyon, sur 6 nœuds, comme puissance de traitement d'un serveur Galaxy ;
- **Sur les postes virtuels** : depuis 2011, l'Université Joseph Fourier organise chaque année une école d'été sur le calcul numérique en physique. Au programme, 10 jours intenses ponctués de travaux pratiques : offrir un environnement homogène quasi-instantanément est indispensable. Co-organisateur de ces écoles, le CBP met en place deux images virtuelles de systèmes : l'une autonome utilisable après l'école d'été, l'autre par SIDUS nécessitant seulement une connexion réseau filaire. Ainsi, les professeurs peuvent-ils quotidiennement adapter leurs TP. C'est une évolution de cette version qui est utilisée, depuis l'été 2012, par le laboratoire de chimie de l'ENS-Lyon et proposée aux laboratoires de biologie LBMC et IGFL ;
- **Sur les machines suspectes** : le démarrage par le réseau offre une investigation de la mémoire de masse système éteint : inutile d'utiliser un LiveCD sur lequel manque toujours son outil *forensics* préféré ;
- **Sur les machines de prêt** : les fabricants de matériels proposent souvent des équipements d'évaluation. La phase d'installation peut être pénible sur des matériels très (voire trop) récents. Avec SIDUS, le système démarre comme sur les autres équipements déjà en service : quelques minutes pour 20 nœuds.

3.2. Pour qui : Quels avantages ?

- **Côté utilisateur** : la machine démarre avec seulement les ressources associées. La version VirtualBox fonctionne au moins sur Linux, Windows et MacOSX : accélération 3D et partage avec l'hôte via un dossier partagé sont disponibles. L'utilisateur retrouve exactement le même environnement que sur les nœuds : l'intégration des codes est donc grandement facilitée. Côté performances, les pertes liées à la virtualisation oscillent entre 10 et 20% (pour VirtualBox) et autour de 5% pour KVM ;
- **Côté administrateur** : une opération impacte l'ensemble de l'infrastructure, de l'ordre du simple sync sur l'arbre SIDUS. L'installation se déroule en quelques dizaines de minutes pour un système complet. Si des différences entre les systèmes sont minimes, un usage simple de scripts ou de Puppet suffit. Si la différence entre les systèmes est importante, un autre arbre SIDUS est construit, voire cloné instantanément avec des mécanismes de *snapshots* : LVM, ZFSonLinux ou BtrFS ;
- **Côté expérimentateur** : ingénieur système ou scientifique, l'environnement SIDUS lui offre la reproductibilité. Deux nœuds démarrant sur le même socle SIDUS disposent exactement du même système au bit près. Une même machine redémarrant SIDUS va retrouver le même système que celui précédemment démarré, quelles que soient les modifications entreprises. Cela permet ainsi, que les machines soient identiques ou pas, de mener des tests vraiment pertinents.

3.3. Combien : Quelles ressources ?

A titre d'exemple, le serveur des clusters du CBP, également passerelle, héberge les services DHCP, DNS, TFTP, NFS et le serveur de batch OAR. Au démarrage de toute l'infrastructure (76 nœuds), le serveur NFS encaisse sans broncher jusqu'à 900 Mb/s.

Le serveur d'instance SIDUS du PSMN, durant la phase de test, était une machine archaïque (Sunfire v40z) : elle remplissait sans souci le service de plus de 330 nœuds avant son remplacement par une machine plus récente.

3.4. Comment installer le système ?

3.4.1. En quelques lignes...

Le pré-requis est réduit, à commencer par un réseau idéalement comparable au débit d'un disque dur. Côté services, sont nécessaires : serveurs DHCP, DNS, TFTP et NFS. Les deux derniers vont «porter» SIDUS. La version opérationnelle du CBP exploite en outre des serveurs tiers LDAP (identification/authentification) et NFSv4 (espaces utilisateurs). Côté client, seul suffit un démarrage PXE opérationnel (par la carte, ou par GPXE sur CDROM ou clé USB).
L'installation comporte 8 phases :

1. préparation du système
2. installation de base (socle Debian, debootstrap)
3. installation des paquets complémentaires (TOUT Debian-Science)
4. purge des paquets non désirés
5. adaptation du système à l'environnement local
6. pointage du système vers les serveurs tiers : authentification et partages utilisateurs
7. création de la séquence de démarrage
8. détachement de SIDUS du système hôte

Durant l'installation, les phases coûteuses sont le téléchargement des paquets et le paramétrage de quelques composants (Perl et LaTeX). Elle dure au mieux 45 minutes pour un arbre complet de 32 Go. Quelques précautions sont cependant nécessaires, liées au montage des dossiers systèmes et à l'inhibition du démarrage des services à leur installation dans SIDUS.

Comment maintenant l'offrir sans le dupliquer ? Une première approche a été d'utiliser un cortège de montages volatils (à base de TMPFS) : elle n'est pas viable. La préférence s'est portée sur mécanisme de LiveCD très répandu : la séquence de démarrage intègre ainsi la superposition de deux couches par le liant AUFS (évolution de UnionFS), l'une lecture seule (le CDROM pour le LiveCD et NFS chez nous), l'autre lecture/écriture (en TMPFS).

Mais comment bénéficier de SIDUS et disposer d'un paramétrage conservé d'un démarrage à l'autre ? La première approche avec un montage NFS exclusif pour chaque nœud a été abandonnée, remplacée par un montage iSCSI associé à chaque nœud. Actuellement, au CBP, les machines SIDUS nécessitant une persistance (comme les nœuds Distonet) utilisent le mécanisme NFSroot+iSCSI=AUFS, les autres NFSroot+TMPFS=AUFS.

Pour conclure, les machines mises à disposition sont assez hétérogènes : les nœuds de clusters (disposant d'équipements réseau rapides), les stations de travail (embarquant des cartes graphiques) ou les machines virtuelles (exigeant un partage des données et une accélération graphique) demandent quelques adaptations. Une première solution serait la persistance, mais trop lourde pour les grands parcs de machines : seront alors préférées l'utilisation de scripts de démarrage, l'exploitation d'un arbre SIDUS séparé ou l'installation de composants tierces.

Dans la suite de l'article, nous ne détaillerons que les points nous semblant indispensables, les autres étant disponibles dans la documentation officielle[14] ou dans la presse spécialisée[15].

3.4.2. Préparation du système

Nous devons préparer un peu notre système afin d'accueillir SIDUS. Nous avons la main sur plusieurs services pour déployer nos clients : serveurs DHCP, TFTP, NFS. Nous entretenons de très bonnes relations avec notre service IT ou nous sommes assez libres pour accéder sans contraintes aux serveurs LDAP et DNS bien définis :

- le service DHCP fournit à notre client une adresse IP mais diffuse deux informations complémentaires : l'adresse du serveur TFTP via la variable `next-server` et le nom du binaire PXE, souvent nommé `pxelinux.0`.
- le service TFTP entre alors en scène. Il offre par TFTP tout le nécessaire permettant le démarrage du système : le binaire `pxelinux.0`, le noyau et le démarrage du système du client. Si nous avons besoin d'offrir des paramètres à tel ou tel client, nous construisons un document spécifique dont le nom sera construit à partir de son adresse MAC (préfixé de 01 et dont les «`:`» sont remplacés par des «-»).
- le serveur NFS s'invite alors dans la boucle : il va offrir la racine du système par son protocole (donc NFSroot). C'est donc dans cette racine, par exemple `/srv/nfsroot/sidus` que nous allons installer notre système client.

Sur nos configurations nous utilisons respectivement isc-dhcp-server, tftpd-hpa et nfs-kernel-server pour les serveurs DHCP, TFTP et NFS.

3.4.3. Installation de base par Deboostrap

Debootstrap permet l'installation d'un système dans une racine. Il exige plusieurs paramètres comme la racine d'installation, l'architecture matérielle, la distribution et l'archive FTP ou HTTP Debian à solliciter pour le téléchargement. Là commence la «spécialisation»de notre installation, à l'origine construite autour d'une distribution Debian. Cet outil est familier de toutes les distributions *Debian-like* : il sera donc disponible chez les dérivées du système à la spirale (à commencer par Ubuntu). Il sera cependant assez facile de réaliser cela sur les Redhat-like, Fedora intégrant par exemple un clone, Febootstrap, mais que nous n'avons pas souhaité tester.

Debootstrap accepte aussi en entrée une liste d'archives (vous savez que Debian est très tatillon sur les licences en séparant les archives en main, contrib et non-free), une liste de paquets à inclure et une liste de paquets à exclure. Nous aurions été ravis de pouvoir, ici, préciser la liste complète des paquets à inclure ou à exclure, mais, malheureusement, cette approche est une voie sans issue : nous installerons donc, dès cette commande debootstrap, un ensemble d'outils indispensables (par exemple le noyau, des micro-logiciels pour un support étendu de tous les matériels et un ensemble d'outils d'audit).

3.4.4. Précautions & création d'un «cordon ombilical» avec l'hôte

A la suite de cette commande, nous devons prendre quelques précautions : normalement, si le paquet Debian est un service, ce dernier démarre après son installation. Nous devons donc inhiber le lancement de ce service par la définition d'un hook. Ce hook sera supprimé à la fin de l'installation ; des paquets exigent l'accès à la liste des processus, du système, des périphériques, de la mémoire virtuelle, des pointeurs de périphériques. Nous devons donc « binder » le montage de ces dossiers du système hôte au système SIDUS. Les dossiers concernés sont : `/proc`, `/sysfs`, `/dev/shm`, `/dev/pts`.

3.4.5. Création de la séquence de démarrage

Comment partager SIDUS sans le dupliquer ? Nous allons nous inspirer d'un mécanisme utilisé dans certains LiveCD : le montage de la racine du système consiste en la superposition de deux

couches, l'une en lecture seule (le système NFSroot) et l'autre en lecture/écriture (un TMPFS dans le cas le plus simple). Les deux couches sont liées par la glue AUFS, le projet successeur de UnionFS.

Tout réside dans un seul et unique hook : rootaufs, placé très tôt dans le démarrage initrd. Son principe repose sur cinq étapes :

1. création de dossiers temporaires `/ro`, `/rw` et `/aufs` ;
2. déplacement de la racine NFSroot du point de montage originel vers dans `/ro` ;
3. montage d'un partage distant, d'une partition locale ou distante, ou d'un volume TMPFS dans un autre point de montage `/rw` ;
4. superposition des deux dossiers `/ro` et `/rw` dans le dossier `/aufs` ;
5. déplacement de `/aufs` vers le point de montage originel.

Ce script `rootaufs` se place dans `$SIDUS/etc/initramfs-tools/scripts/init-bottom`
Le script originel a été inspiré par le projet rootaufs[21] de Nicholas A. Schembri. Il a été profondément modifié pour l'adapter à notre infrastructure : une version est disponible en ligne[20].
Un dernier petit effort avant de disposer d'un système fonctionnel : nous allons créer par update-initramfs un InitRD spécifique pour notre boot NFS contenant les modifications suivantes :
– `aufs` dans `/etc/initramfs-tools/modules`
– `eth0` comme `DEVICE` dans `/etc/initramfs-tools/initramfs.conf`
Il suffit ensuite de copier le noyau et le InitRD dans la définition du serveur TFTP.
A l'origine, nous avons exploré la possibilité d'offrir un second partage NFS en lecture/écriture pour la persistance des modifications associées aux clients d'un redémarrage à l'autre. Cette version, bien que fonctionnelle, exigeait l'ouverture d'un partage NFS atomique pour chaque client : imaginons la charge supplémentaire pour le serveur !
Nous avons donc préféré une autre approche de la persistance, sous la forme d'un disque réseau de technologie iSCSI. Ainsi, nous créons un volume partagé iSCSI par client. Pour des raisons de simplicité, le volume offert porte l'IP du client et nous ne fournissons que le serveur de volume iSCSI. Les login et mot de passe d'accès par défaut sont dans le fichier rootaufs.

3.4.6. Rupture du «cordon ombilical» avec le système hôte

De manière à installer correctement SIDUS, nous avons été contraints de lier étroitement système hôte et chroot. Il est donc nécessaire de :
– démonter les dossiers système : `/proc`, `/sysfs`, `/dev/shm`, `/dev/pts` ;
– effacer les dossiers temporaires ;
– purger des processus liés au dossier d'installation de l'instance SIDUS au besoin.

3.5. Comment administrer le système ?

Si l'administration est plus lourde que son installation, le bénéfice du premier efface la perte récurrente du second. Finalement, avec SIDUS, chaque phase d'administration intègre les mêmes mécanismes qu'à l'installation : protection contre le démarrage des nouveaux services et montages de dossiers systèmes. Le reste est identique.

En définitive, une instance SIDUS s'administre comme tout système chrooté : il y a cependant des précautions à prendre, toutes les fonctions d'administration n'exigeant pas les mêmes contraintes. Souvent, un chroot sur la racine de SIDUS suivi de la commande système suffit.

Une nouvelle approche, basée sur l'ouverture d'un SSH directement dans l'instance SIDUS, est en préparation.

4. Expérimentations

4.1. Deux expérimentations, deux contextes, un socle unique

Les expérimentations suivantes illustrent le bénéfice de SIDUS dans deux contextes très différents, et pour des destinations que le sont tout autant. Dans le premier, SIDUS est exploité comme socle de nœuds de calcul matériellement identiques pour l'évaluation d'un système de fichiers distribués GlusterFS, dans un environnement de calcul haute performance. Dans le second, SIDUS demeure le socle mais pour des matériels différents : l'objectif était ici de comparer le parallélisme massif sur les composants dotés d'accélérateurs graphiques, de type GPU (destinés à de l'affichage) ou GPGPU (destinés à du calcul scientifique).
Si la première expérimentation nous montre qu'une trop grande intelligence laissée au processeur pour la gestion de l'énergie plombe les performances et génère de la variabilité, la seconde illustre que la variabilité peut, dans certains cas, être un critère discriminant pour comparer des matériels, voire des générations de matériels.

4.2. De GlusterFS sur InfiniBand aux effets néfastes du C-states

4.2.1. Contexte : du choix d'un espace temporaire partagé au protocole expérimental

Le Pôle Scientifique de Modélisation Numérique, centre de calcul de l'ENS-Lyon, a choisi depuis longtemps l'internalisation de la gestion de son infrastructure : tout nouvel équipement se borne généralement à l'achat de matériel.
Début 2013, l'arrivée du nouvel équipement Equip@Meso (dans le cadre des investissements d'avenir) exigeait l'installation et donc le choix d'un espace de partage haute performance entre les nœuds. Le Centre Blaise Pascal avait déjà, dans le cadre du projet DistoNet[17], étudié plusieurs systèmes de fichiers répartis, notamment GlusterFS : il était donc intéressant que le centre de production (le PSMN) demande au centre d'essais (le CBP) d'étudier GlusterFS sur des nœuds équivalents aux nœuds du futur équipement, disposant d'une interconnexion haute passante et basse latence sous Infiniband. Vingt nœuds Hewlett Packard SL230, disposant de 16 cœurs physiques et 64 Go de RAM, interconnectés en Infiniband, ont donc été mis à disposition pour cette étude.

4.2.2. Le choix d'un RAMdisk

Tout d'abord, la première étape fut le démarrage de ces machines dans l'environnement du CBP : cela ne prit que quelques minutes avec SIDUS, tout au plus une demi-journée avec la récupération de l'adresse MAC et la configuration du contrôle IPMI. Puis, d'un point de vue expérimental, il était indispensable de se libérer de tout goulot d'étranglement matériel de stockage en supprimant toute latence disque. Le choix de système de fichiers en mémoire vive s'imposait de lui même : TMPFS en est une implémentation courante. Une seconde possibilité se basait sur le module BRD créant un «mode bloc» lequel devait être formaté pour ensuite être offert. L'analyse de presque une dizaine de systèmes de fichiers ont montré que le système le plus efficace pour cet usage était, sans surprise, le plus simple : ext2. Ensuite, ll fallait offrir le volume GlusterFS pointant vers cet espace de *Ramdisk* et le monter à l'aide d'un ou plusieurs clients. Enfin, venait l'évaluation de la performance d'accès exploitant massivement l'outil IOzone3[11], sur ces espaces partagés. De manière à disposer d'échantillon représentatif, la première expérience exploitait dix clients se connectant sur dix serveurs différents. Les machines (clients comme serveurs) étaient strictement identiques, simplement sorties de leurs

cartons. Pour chacun des couples client-serveur, vingt expériences furent menées.
Les premiers résultats ont été très surprenants et sont illustrés sur les deux schémas «radar» de la figure 1, *BIOS Initial, Noeud 1 sur 11* et *BIOS Initial, Noeud 3 sur 13*. Dans le second cas, des performances autour de 500 Mo/s très stables et homogènes. Dans le premier, de bien meilleures performances, mais une variabilité très surprenante atteignant des valeurs du simple au double ! Etant donné que nous étions dans un environnement SIDUS, les configurations ne pouvaient pas ne pas être identiques ! Les machines démarraient le même noyau, lançaient les mêmes séquences de démarrage. A noter que, sur les dix couples, deux seulement présentaient ces variabilités, les huit autres restant d'une intéressante stabilité.

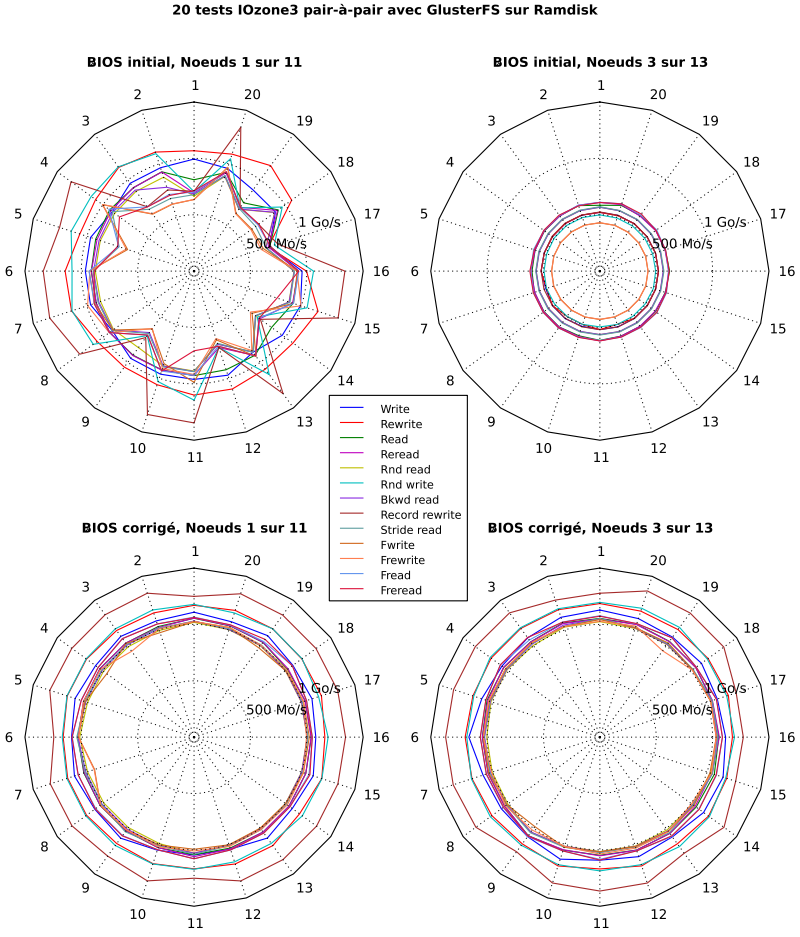

FIGURE 1 – Série de 20 tests IOzone3 entre un client et un serveur, pour deux couples différents de machines.

Après des heures d'investigation et de tests multiples, l'examen des BIOS respectifs des nœuds montra des différences : dans le cas où les machines présentaient des résultats stables mais peu performants, la gestion de la consommation était celle par défaut. Dans l'autre cas, teinté de performance mais de forte variabilité, un des deux nœuds était en *Max Performance*. Le passage de ces réglages en *Max Performance* a permis de retrouver les deux schémas «radar» de la fi-

gure 1, *BIOS corrigé, Noeud 1 sur 11* et *BIOS corrigé, Noeud 3 sur 13*. Tous les couples présentaient des taux de transfert d'une remarquable stabilité autour de 1 Go/s.

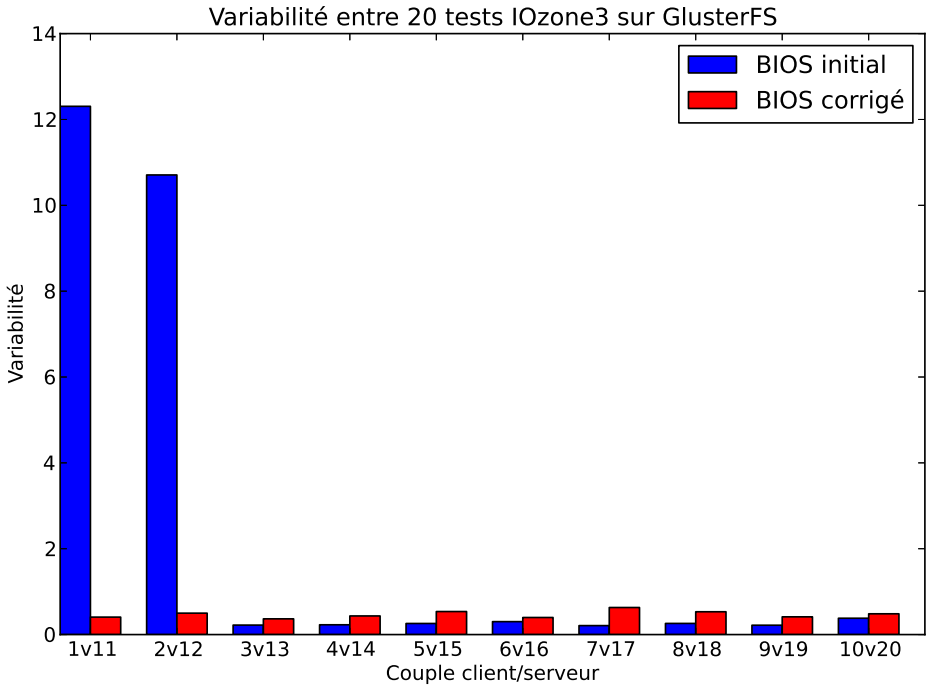

FIGURE 2 – Série de 20 tests IOzone3 entre un client et un serveur, pour deux couples différents de machines.

L'accélération générale (rapport entre la médiane des durées d'exécution totale pour les 20+20 expériences) et la diminution de la variabilité (rapport entre la médiane et l'écart-type sur les 20+20 expériences) sont illustrées sur la figure 2 pour les dix couples clients serveurs ayant participé à la campagne d'essais. La variabilité a été divisée d'un facteur 30 pour les deux premiers couples et la performance d'un facteur 2 pour les 8 derniers.

4.3. Lorsque la non-reproductibilité devient ... reproductible

Mi-2013, lors de l'intégration opérationnelle de GlusterFS sur l'Equip@Meso de l'ENS-Lyon, des variabilités d'un ordre de grandeur affectaient la communication entre deux nœuds via InfiniBand en IP, même avec des réglages identiques à ceux de la première expérience (mais sur des matériels différents). Trouver le bon paramétrage de BIOS (différent de celui évoqué ci-dessus) a été long et fastidieux mais a permis de limiter la variabilité à une valeur de quelques pour-cent. Ainsi, les mécanismes d'autoconfiguration énergétiques, trop «laissés» au processeur, génèrent une variabilité dont l'écart-type laisse perplexe : les intégrer dans le processus même de l'expérience numérique devient une exigence ! Pour parodier Clémenceau, «*La gestion de l'énergie est une chose trop sérieuse pour la confier au processeur lui-même !*»

4.4. La variabilité : de la jeunesse des BIOS à la vieillesse des composants

Des variabilités temporelles n'affectent pas uniquement les équipements récents. Le CBP fonctionne essentiellement sur des équipements d'âge «canonique», déclassés des centres de calcul. Leur exploitation quotidienne, sur des expériences, menées sur ces trois dernières années, suggère de multiples origines à cette variabilité sur les temps de calcul : cela va de l'état des ventilateurs, du vieillissement des alimentations, du taux de remplissage de messages de la pile IPMI, de la tenue dans le temps de la pâte thermique entre processeurs et radiateur, de la position dans la baie, de la «maladie des condensateurs», ... : ces différences influençaient les environnements électrique et thermique, mais nous ne nous doutions pas de l'importance de telles influences, essentiellement lorsque les réseaux ou de grosses quantités de mémoire sont sollicitées.

4.5. De l'émergence de «domaines de vol» en parallélisme

4.5.1. Du multi-cœurs au *myri*-cœurs : nouveau paradigme ?

Les processeurs graphiques de conception Nvidia ou AMD/ATI contiennent de plusieurs centaines (pour les plus anciens) à plusieurs milliers d'unités arithmétiques et logiques. Évaluer la scalabilité devient un enjeu fondamental, ne serait-ce qu'afin d'estimer, pour une application devenue indissociable de son jeu d'entrée, la durée totale d'exécution. La loi d'Amdahl [2] offre un modèle d'estimation simple, voire simpliste, de cette durée dès lors que le taux de parallélisation est connu. Cependant, ce modèle , dans un cadre opérationnel exploitant des processeurs classiques voire des accélérateurs, montre rapidement ses limites : la loi de décroissance se trouve souvent tourmentée et des pseudo-périodes font leur apparition.

Comment, dès lors, estimer la scalabilité d'un matériel particulier ou la simple capacité à adresser à chacune des unités de traitement le maximum de tâches ? Pour cette mission, nous avons choisi le programme le plus simple qui soit : l'estimation de Pi par la méthode de Monte Carlo, comme langage l'OpenCL, comme socle technique, les stations de travail en libre service et quelques nœuds du cluster de développement du Centre Blaise Pascal, comme environnement système, un SIDUS et comme élément de différenciation, des cartes graphiques différentes.

4.5.2. Le Pi par Monte Carlo : trop sommaire pour être pertinent ?

Les modèles de programmation sur GPU sont à l'origine vectoriels. Cependant, la «rastérisation», ces transformations de cette description tridimensionnelle de la scène à l'image proprement affichée sur l'écran, a exigé l'intégration de nouveaux modèles faisant la part belle à une opération : la multiplication matricielle. Il n'est donc pas étonnant que les primitives BLAS [9] et ses extensions LAPACK [10] fassent partie des premières librairies optimisées pour l'exploitation des GPU : les librairies BLAS et LAPACK avaient suivi les mêmes transformations. Ces librairies permettent généralement de comparer, à grande échelle, les machines parallèles, au point que le Top 500 l'exploite au travers du LinPack.

Cependant, comme socle de notre expérimentation, nous avons choisi de nous appuyer sur l'illustration classique de la méthode de Monte Carlo : le calcul de Pi, pourtant irrationnel, comme le rapport entre deux entiers. L'algorithme associé se compose donc d'une boucle comprenant le tirage de deux flottants aléatoires (les x et y), le produit de leur carré et l'indentation d'un compteur si ce produit est inférieur à l'unité. Pour paralléliser cela, il suffit de distribuer le domaine d'exploration (le nombre total d'itérations) équitablement entre les tâches simultanées que nous exécutons. L'efficacité repose essentiellement sur l'exploitation d'un générateur de nombres aléatoires correctement implémenté. Afin de disposer d'une portabilité maximale

de notre code (sous tous les langages), nous exploitons la méthode MWC[8] de Marsaglia nécessitant 2 germes et basée sur une succession d'opérations entières : le nombre pseudo-aléatoire fourni est un entier non signé sur 32 bits. Notre boucle se compose ainsi de :
– 2 tests : le branchement de l'itérateur et la vérification de la position dans le quadrant ;
– 5 opérations flottantes : 2 multiplications par une constante, 2 calculs de carré, 1 addition ;
– 24 opérations entières : 11 pour chaque nombre aléatoire, 1 pour l'itérateur, 1 pour le compteur.
La parallélisation se développe s'intègre trivialement en divisant le nombre total d'itérations par le nombre de tâches. Chaque tâche dispose ainsi d'un nombre équivalent d'itérations. Il n'y a donc aucune communication durant le calcul. L'estimation finale peut, soit se construire sur l'agrégation de la somme et la division finale, soit une estimation élémentaire laquelle est moyennée. Cette dernière méthode permet de traquer plus facilement les anomalies.

Côté implémentation, de manière à pouvoir comparer efficacement processeurs traditionnels, accélérateurs MIC et ou accélérateurs graphiques GPU ou détournés GPGPU, nous avons choisi l'OpenCL[5]. De manière à rendre le code le plus visible possible et permettre un usage instantané d'outils de métrologie et de statistiques, nous utiliserons comme langage principal Python accompagné de l'interface PyOpenCL développée par Andreas Klöckner[6]. Plusieurs implémentations de ce langage existe :
– une purement GPU de Nvidia[12], intégrée à chaque pilote Nvidia ;
– une polyvalente proposée par AMD[1], permettant aussi bien l'exploitation de GPU anciennement ATI et le processeur traditionnel ;
– une proposée par Intel[4], permettant l'exploitation d'accélérateur de type Xeon Phi et des processeurs les plus récents ;
– une implémentation Open Source très active, *Portable Computing Language*[13], laquelle nécessite le compilateur alternatif CLang.

Notons que, dans un cadre opérationnel, nous pouvons déjà faire part de quelques remarques sur ces différentes implémentations :
– l'implémentation Nvidia, toujours livrée avec le pilote courant (tout comme la librairie CUDA), se montre particulièrement polyvalente sur les cartes graphiques des plus anciennes (GT218) au plus récentes (Quadro K4000). Nos expérimentations ont été menées avec les pilotes dans leur version 331.39 ;
– l'implémentation OpenCL d'Intel ne fonctionne sur les processeurs récents. Nos évaluations ont montré qu'il était impossible de faire fonctionner la 3.2.1.16712 sur les processeurs Intel Pentium D 651, Core 2 6300, Xeon 5440. Cette version 3.2.1.16712 fonctionne parfaitement sur les processeurs suivants testés : Westmere X5650, E5-2665, i7-3840QM ;
– l'implémentation OpenCL d'AMD/ATI est comparable à celle de Nvidia : elle reste livrée avec le pilote graphique. Contrairement à Nvidia, AMD a choisi délibérement d'exclure les cartes jugées trop anciennes de ses pilotes récents : les cartes Radeon HD série 4000 sont désormais inutilisables avec les pilotes supérieurs à 12.6. Plus surprenant, il en est de même des cartes professionnelles FirePro V5900 ! Notons quand même que AMD/ATI se rattrape en proposant une implémentation CPU de OpenCL laquelle fonctionne sur tous les processeurs que nous avons eu entre les mains.

4.5.3. Le socle SIDUS : une instance «presque» unique

Le socle SIDUS, c'est-à-dire l'environnement système complet que nous utilisons pour évaluer les performances de ces différentes puces graphiques est une Debian Wheezy 7.4, disposant

d'un noyau 3.2.54. L'intrication du pilote graphique permettant de lancer des programmes en OpenCL sur un système Linux est profonde : au-delà du module noyau, il y a la liaison avec le serveur X, les librairies spécifiques OpenGL, les libairies OpenCL, le composant ICD propre à chaque implémentation de OpenCL. Les librairies de développement OpenCL sont, par contre, très différentes : en effet, programmer en OpenCL *«from scratch»* est si difficile que, autant AMD/ATI que Nvidia ont décidé, dès les origines, de fournir un SDK rendant le code peu portable...

Ainsi, de manière à disposer d'un environnement ne nécessitant que le minimum de reconfiguration au démarrage de la machine, nous avons choisi de construire trois instances parallèles : une servant de socle et universelle, ne disposant que de pilotes graphiques Open Source, une dédiée aux matériels avec circuits Nvidia, une dédiée aux matériels avec circuits AMD/ATI.

Par défaut, pour Nvidia, la Debian Wheezy propose les pilotes graphiques dans leur version 304.88 et l'environnement CUDA 4.2.9. Ces pilotes sont largement insuffisants pour permettre à des cartes récentes comme la GTX Titan, la Quadro K4000 ou la Tesla K40 de fonctionner parfaitement (voire tout court). Nous avons donc pris le parti de rétroporter les pilotes Nvidia et l'environnement `nvidia-cuda-toolkit` des dépôts expérimentaux de la Debian : nous disposons de l'environnement le plus récent tout en bénéficiant de l'infrastructure de gestion des paquets : pour nos tests, nous étions dans la version 331.38 du pilote et 5.5.22 de l'environnement CUDA.

Côté AMD/ATI, nous ne disposons que du pilote et de l'environnement 12.6 par défaut dans la Wheezy. Un rétroportage a permis l'exploitation de la version 14.1. Ce sont donc ces versions que nous avons intégrées de base dans nos instances SIDUS. Au démarrage de l'instance SIDUS sur chaque machine, c'est l'adresse MAC qui sert de discriminant pour démarrer l'une ou l'autre des instances. Notons pour conclure que, avant le démarrage du gestionnaire de connexion, nous appliquons un `nvidia-xconfig` ou un `aticonfig -initial` pour créer le `/etc/X11/xorg.conf` le plus cohérent possible avec le matériel intégré.

Les expérimentations sont menées sur 20 cartes graphiques différentes, anciennes et modernes, d'entrée de gamme à professionnelles, autant pour un usage purement graphiques que calcul scientifiques. Ainsi, nous avons :
- en entrée de gamme : chez Nvidia, les GT220, GT430, GT620 et GT640. Chez AMD/ATI, les Radeon HD6450 et Radeon HD 6670 ;
- en professionnelle : chez Nvidia, les Quadro 4000 et Quadro K4000. Chez AMD/ATI, la FirePro v5900 ;
- en *«gamer»* : chez Nvidia, GTX 260, GTX 680, GTX 690 et GTX Titan. Chez AMD/ATI, les Radeon HD 5850 et HD 7970 ;
- en HPC : chez Nvidia, les Tesla C1060, M2070, M2090, K20m et K40.

Chez Nvidia, nous pouvons rapidement extraire des générations de circuits graphiques :
- GT 220 : GT220, GTX 260 et Tesla C1060 ;
- Fermi (GF) : GT 430, GT 620, Quadro 4000, Tesla M2070, Tesla M2090 ;
- Kepler (GK) : GT 640, GTX 680, GTX 690, GTX Titan, Quadro K4000, Tesla K20m, Tesla K40.

Terminons cette description par le socle matériel : une majorité de ces stations sont des stations Dell Precision T5600, des postes de travail Dell Optiplex 745 et des nœuds de calcul Dell

C6100 connectés à un chassis GPGPU Dell C410x. Certaines machines sont équipées des mêmes cartes :6 T5600 se partagent 8 Nvidia Quadro 4000, 8 Optiplex 745 se partagent 4 AMD/ATI Radeon HD 6450 et 4 Nvidia GT620.

4.5.4. Premières expérimentations

Nous disposons donc d'un code Python, disposant de l'API PyOpenCL pour communiquer en OpenCL avec les cartes GPU ou GPGPU. Ce code est assez polyvalent pour exploiter les implémentations OpenCL pour processeurs traditionnels CPU, cartes accélératrices Xeon Phi et cartes GPU et GPGPU. Nous allons exploiter les deux modes de parallélisme (ou de distribution de tâches) que confère OpenCL (tout comme CUDA) : le mode *Blocks* (chez CUDA) ou *Work Item* (chez OpenCL) et le mode *Threads*. Nous étudions le temps de calcul nécessaire à 1 milliard d'itérations pour le calcul de Pi, calcul que nous répétons à 5 reprises, pour un nombre de tâches (en *Blocks* ou *Threads*) allant de 1 à 1024. La métrologie présentée se base sur les compteurs intégrés à l'API OpenCL.

4.6. Premières constatations pour tous les GPU/GPGPU

Les résultats préliminaires sont présentés, par ordre alphabétique de carte, sur la figure 3. Chaque graphique présente une performance en nombre d'itérations par seconde, rapport entre le milliard d'itérations et la médiane sur la durée des 5 essais, pour chacun des tests de 1 à 1024 tâches simultanées.

Ce que nous constatons :

- une croissance générale pour chacune des cartes (c'est heureux !) mais un plafonnement pour certaines d'entre elles ;
- les cartes présentées comme les plus puissantes (les GTX 690, GTX Titan, Tesla K20m et Tesla K40) sont inférieures en performance à la GTX 680 : une expérimentation au-delà de 1024 tâches s'avère indispensable pour explorer un parallélisme encore plus massif ;
- une limitation à 256 ou 512 *Threads* pour les circuits AMD/ATI et les circuits GT de Nvidia : chez Nvidia, cette limitation est associée à la vieillesse des matériels. Chez AMD, il s'agit d'une constante depuis les origines ;
- de larges oscillations en tâches *Blocks* pour des valeurs de tâches excédant le nombre maximal de *Threads* ;
- des similitudes visuelles marquées :
  - entre GT 220, GTX 260 et Tesla C1060 : ce sont des circuits *GT*,
  - entre GT 430, GT 620, Quadro 4000, Tesla M2070 et Tesla M2090 : ce sont des circuits *Fermi*,
  - entre GT 640, GTX 680, GTX 690 et Quadro K4000 : ce sont des circuits *Kepler*,
  - entre GTX Titan et Tesla K20m : ce sont des circuits *Kepler* mais 110,
  - entre Radeon HD 5850, HD 6670, FirePro V 5900
- une superposition presque parfaite des parallélismes *Blocks* et *Threads* pour les Nvidia GT 430, GT 620, Quadro 4000, Tesla M2070 et Tesla M2090 ;
- des phénomènes en dents de scie de période 64 pour les GT 430, GT 620, Quadro 4000, Tesla M2070 et Tesla M2090 ;
- un décrochage caractéristique à 640 tâches entre les parallélismes *Blocks* et *Threads* pour les GT 640, GTX 680, GTX 690 et Quadro K4000 ;
- une remarquable linéarité pour la Radeon HD 7970.

Ainsi, avec la figure 3, nous pouvons reconnaître visuellement le lien de parenté entre les cartes

graphiques d'un coup d'œil uniquement en analysant son parallélisme en OpenCL.

Est-ce que ces «comportements» au parallélisme sont reproductibles sur des cartes différentes ? C'est ce que nous illustrons sur les 3 groupes de cartes les plus nombreux à notre disposition, sur les Nvidia Quadro 4000 (figure 4), GT 620 (figure 6), Radeon HD 6450 (figure 5).

Sur la figure 4, nous retrouvons bien les mêmes dents de scie de période 64 à partir de 192 tâches et une bonne reproductibilité. Les performances maximales sont identiques à 0.3% près.

Sur la figure 6, nous retrouvons bien les mêmes dents de scie de période 64 à partir de 192 tâches et une bonne reproductibilité. Les performances maximales sont identiques à 0.1% près.

Sur la figure 5, nous retrouvons bien structement les mêmes oscillations à partir de 256 tâches et une bonne reproductibilité. Les performances maximales sont identiques à 0.1% près pour le les tâches *Blocks* et 0.3% *Threads*.

4.6.1. Quelle variabilité pour ces mesures ?

En complément de la comparaison de performances, remarquablement stables pour des GPU identiques dans des machines différentes, ou visuellement proches lorsque les circuits sont de la même famille, que se passe-t-il si nous analysons comme précédemment, non pas la performance, mais la variabilité définie comme le rapport entre l'écart-type et la médiane ?

Reprenons les 3 groupes de cartes les plus nombreux à notre disposition, soit les Nvidia Quadro 4000 (figure 7), GT 620 (figure 9), Radeon HD 6450 (figure 8).

Nous constatons ainsi que, visuellement, les variabilités :
- n'évoluent pas de manière continue en fonction du nombre de tâches ;
- n'évoluent pas de manière identique en fonction de la nature des tâches (*Blocks* ou *Threads*) ;
- sont «comparables» pour un même circuit vidéo, d'une machine à l'autre ;
- sont très différentes d'un circuit vidéo à l'autre, même s'ils font partie de la même famille.

Cette étude que nous avons menée sur les 3 groupes de cartes les plus nombreux, nous la réalisons sur l'ensemble des circuits à notre disposition, aussi bien pour les tâches de type *Blocks* (figure 10 que de type *Threads* (figure 11).

Sur les figures 10 et 11, nous avons représenté les variabilités de l'ensemble des cartes pour tous les circuits vidéo que nous avions à disposition. Quelles sont nos constatations ?
- lorsque les circuits sont identiques, les variabilités sont très proches :
  - pour le GK104 des GTX 680 et GTX 690 (la GTX 690 est en fait une GTX 680 doublée),
  - pour le GK110 de la GTX Titan et le GK110GL de la Tesla K20m dans un moindre mesure,
  - pour le Caicos des 4 machines équipées de HD 6450,
  - pour le GF108 des 4 machines équipées de GT 620,
  - pour le GF100GL des 4 machines équipées de Quadro 4000 ;
- lorsque les circuits sont proches, les variabilités présentent des similitudes :
  - pour le GT218 de la GT 220, le GT200 de la GTX 260 et le GT200GL de la Tesla C1060,
  - pour le GK107 de la GT 640 et le GK104 de la GTX 680,

  - pour le GF100GL de la Quadro 4000 et le GF110GL des Tesla M2070 et M2090 ;
- les différences sont plus marquées dans des tâches de type *Threads* ;
- les variabilités des cartes Tesla K20m ou GTX Titan sont importantes (de l'ordre de 10%) comparées aux autres cartes (de l'ordre du %) et restent très largement supérieures à celle de Radeon HD 7970, laquelle reste inférieure à 0.1% ;
- la Tesla K40 (avec un GK110BGL) confirme sa différence face à la Tesla K20m (avec un GK110GL) ;
- les performances de cartes différentes peuvent présenter des comportements comparables mais leurs variabilités sont susceptibles de les différencier.

4.6.2. De la variabilité comme signature caractéristique d'un processeur graphique

Que pouvons-nous conclure de cette analyse du comportement en parallélisme de ces 20 types de circuits graphiques ?

La première, c'est une certaine reproductibilité dans l'espace (sur des cartes différentes) de nos résultats.

La seconde, entre deux cartes, un comportement comparable en performance peut être très différent en variabilité : la variabilité est donc un critère discriminant pour comparer les circuits graphiques au sein d'une même famille.

La troisième, c'est qu'une seule carte, sur les 20 testées, présente une caractéristique proche de la loi d'Amdahl : la AMD/ATI Radeon HD 7970. Pour les autres, cela n'a pas de sens d'utiliser cette loi pour extrapoler des performances sachant que même une interpolation est impossible.

Ainsi, un test aussi élémentaire que ce programme facilement parallélisable permet de revenir à des caractéristiques étonnantes de ces nouveaux composants que nous ne pouvons plus ignorer en calcul scientifique. L'architecture de ces composants est si complexe que l'expérimentation devient indispensable, ne serait-ce que pour connaître son «comportement» : cette expérimentation permet de caractériser un circuit graphique aussi bien que les essais en vol déterminent le domaine de vol d'un avion.

En conclusion, qu'apporte SIDUS dans cette analyse ? En proposant un socle unique, rapidement déployable, imposant à toutes les machines examinées l'assurance que toutes utilisent les mêmes composants logiciels, SIDUS permet la mise en lumière de caractéristiques surprenantes : celle que la variabilité est reproductible !

5. Conclusion SIDUS

Outil d'administation ou de recherche ? Avec SIDUS, il n'y a plus de système sur les disques durs embarqués (et donc des Watts et des BTU à récupérer, de la maintenance en moins), plus de procédures lourdes d'installation ou d'administration : c'est donc indubitablement un outil d'administration de parc, quelles que soient la machine ou sa destination. Sa seule exigence : un démarrage en réseau. Pour quel bénéfice ? Du temps d'ingénieur à reventiler sur des tâches plus en relation avec le calcul scientifique.

De plus, face aux évolutions matérielles permanentes, c'est un outil offrant une comparaison rapide entre des machines différentes, des BIOS dissemblables, des noyaux distincts, voire des

environnements climatiques variés. C'est donc, par conséquent, un outil d'investigation qui élimine toute variabilité sur le socle logiciel et permet de mettre en lumière toute autre variabilité.

Ensuite, par sa simplicité, SIDUS facilite l'accès aux ressources de calcul scientifique. Ce n'est pas dans l'apprivoisement de la puissance de calcul que réside la difficulté d'apprentissage : c'est dans la maîtrise des outils. Démarrer un système complet graphique, pleinement opérationnel, en quelques secondes, comparable à un nœud de calcul, sans toucher quoi que ce soit à la machine hôte, favorise cette transition. Elle permet en outre l'exploitation de ressources locales (CPU & RAM) souvent largement surdimensionnées pour leur usage courant. Cette approche COMOD, certainement l'avenir d'un *sparse computing*, reste à généraliser pour rationaliser au mieux l'usage d'un parc informatique surtout en cette période de restriction budgétaire. Le portage de son poste de travail vers les équipements dédiés est instantanée : SIDUS accélère donc appropriation et intégration pour les acteurs de la recherche exploitant le calcul scientifique.

Quelles perspectives pour SIDUS ? Le mettre à disposition sur tout le site de l'ENS-Lyon, poursuivre dans la lignée de 2013[16][18][19] sa promotion dans la communauté HPC et sur la place lyonnaise, permettre son accès via un accès VPN, l'adapter sur des structures de grilles ou expérimentales (sur Grid'5000) : autant de projets à l'étude. Un autre axe de travail vise aussi le remplacement de la glu AUFS par OverlayFS, nouvellement intégrée aux noyaux Linux les plus récents.

Bibliographie

1. AMD. – Amd opencl zone. – `http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/`.
2. Amdahl (G. M.). – Validity of the single processor approach to achieving large-scale computing capabilities. *In : AFIPS Conference Proceedings*, pp. 483–485.
3. Corden (D. M. J.) et Kreitzer (D.). – Consistency of floating-point results using the intel(r) compiler or why doesn't my application always give the same answer ? – `http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler`, août 2012.
4. Intel. – Intel sdk for opencl applications xe 2013. – `http://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe`.
5. Khronos. – The open standard for parallel programming of heterogeneous systems. – `https://www.khronos.org/opencl/`.
6. Klöckner (A.). – Pyopencl. – `http://mathema.tician.de/software/pyopencl/`.
7. Kreitzer (D.). – Home - abinit. – `http://www.abinit.org`, janvier 2014.
8. Marsaglia (G.). – Some good(?) random number generators, with c code, comparisons. – `http://www.math.niu.edu/ rusin/known-math/99/RNG`.
9. NetLib. – Blas (basic linear algebra subprograms). – `http://www.netlib.org/blas/`.
10. NetLib. – Lapack - linear algebra package. – `http://www.netlib.org/lapack/`.
11. Norcott (W.). – Iozone filesystem benchmark. – `http://www.iozone.org/`, janvier 2014.
12. Nvidia. – Cuda zone opencl. – `https://developer.nvidia.com/opencl`.
13. POCL. – Portable computing language. – `http://pocl.sourceforge.net/`.

14. Quemener (E.). – Site institutionnel de sidus. – `http://www.cbp.ens-lyon.fr/sidus/`, janvier 2014.
15. Quemener (E.) et Corvellec (M.). – Extreme os deduplication using sidus. *Linux Journal*, no235, nov 2013, pp. 100–111.
16. Quemener (E.) et Corvellec (M.). – Os deduplication with sidus (single-instance distributing universal system). – `http://conference.scipy.org/scipy2013/presentation_detail.php?id=199`, juin 2013.
17. Quemener (E.) et Taulelle (L.). – Le projet distonet : le stockage distribué du cluster au poste de travail. *In : Journées Réseaux 2011*. – Paris, France, novembre 2011.
18. Quemener (E.) et Taulelle (L.). – Déduplication extrême avec SIDUS : un premier pas vers la reproductibilité ? *In : Journées SUCCES 2013*. – Paris, France, novembre 2013.
19. Quemener (E.) et Taulelle (L.). – Déduplication extrême d'os avec sidus. *In : Journées Réseaux 2013*. – Montpellier, France, décembre 2013.
20. Quemener (E. Y.). – Exemple de rootaufs. – `http://www.cbp.ens-lyon.fr/sidus/rootaufs`, janvier 2014.
21. Schembri (N. A.). – Google code de rootaufs. – `http://code.google.com/p/rootaufs/`, septembre 2010.

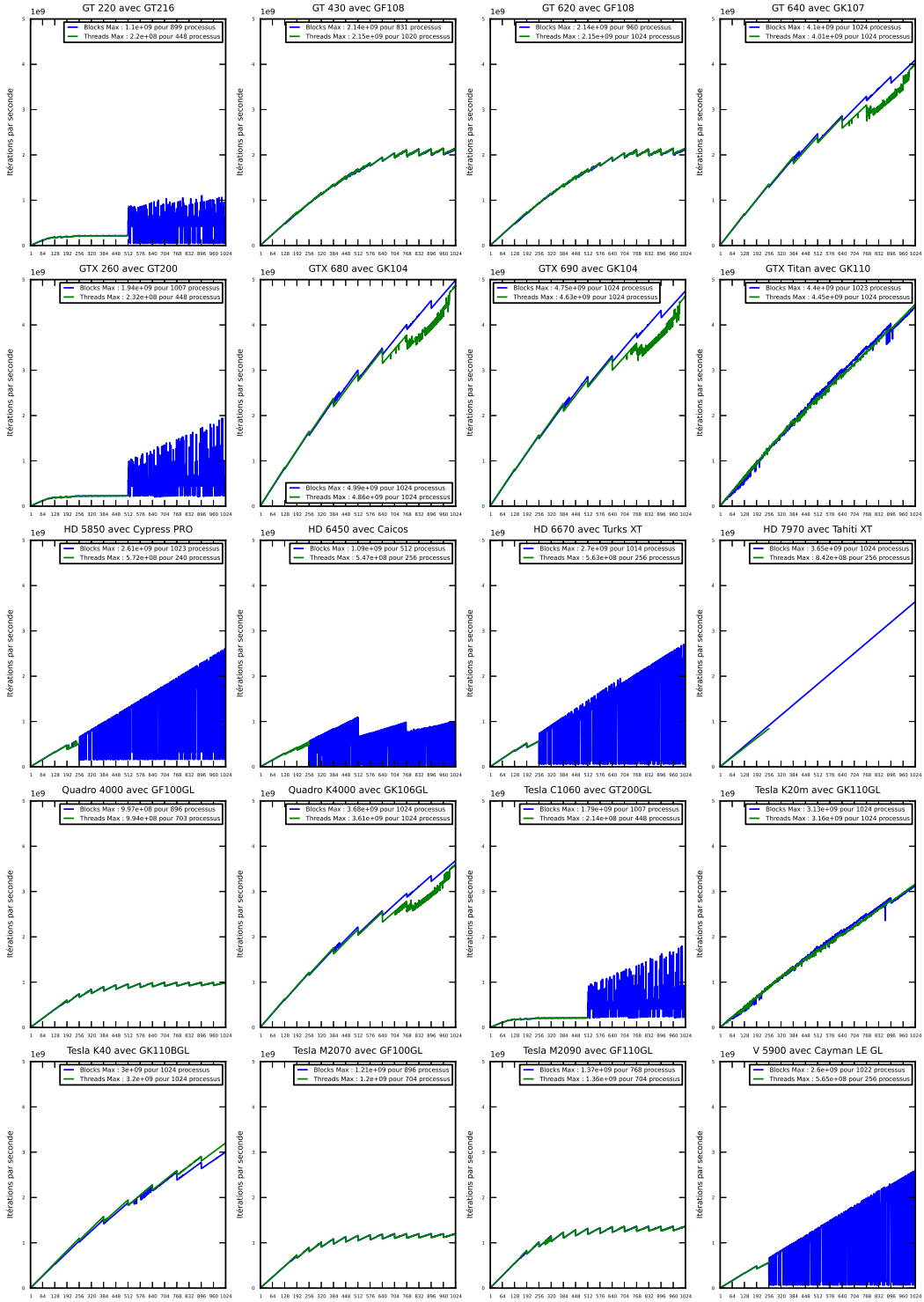

FIGURE 3 – Comparaison de performances (nombre d'itérations par seconde) pour 20 cartes graphiques différentes, destinées à de l'affichage (ou du calcul scientifique pour les Tesla). En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* (ou *Work Item*) et *Threads*. Pour chaque nombre de tâches simultanées, 5 expériences. La mesure est le temps de calcul utilisant les *timers* internes au GPU.

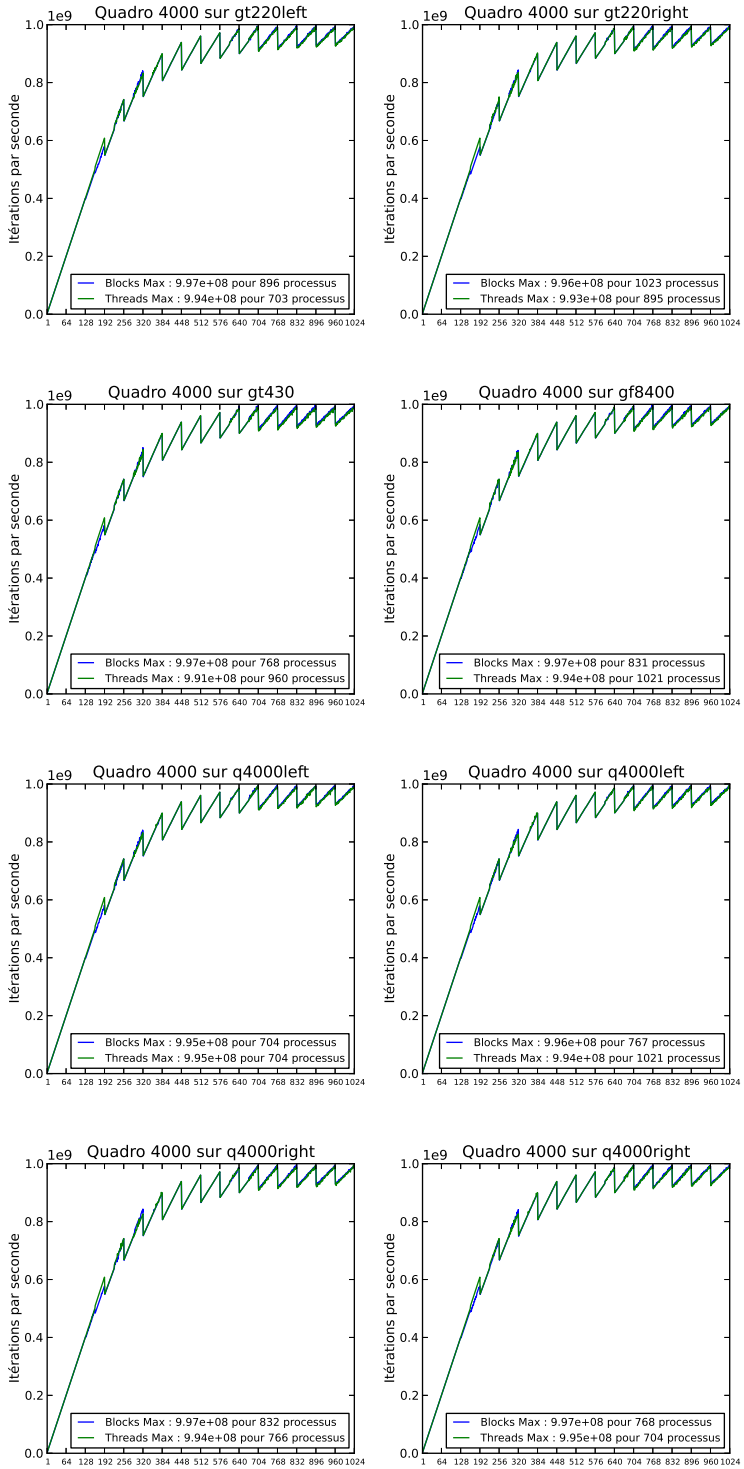

FIGURE 4 – Comparaison de performances (nombre d'itérations par seconde) pour un GPU particulier, une Nvidia Quadro 4000, sur 8 cartes différentes & 6 machines différentes. En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* et *Threads*. Pour chaque nombre de tâches simultanées, 5 expériences.

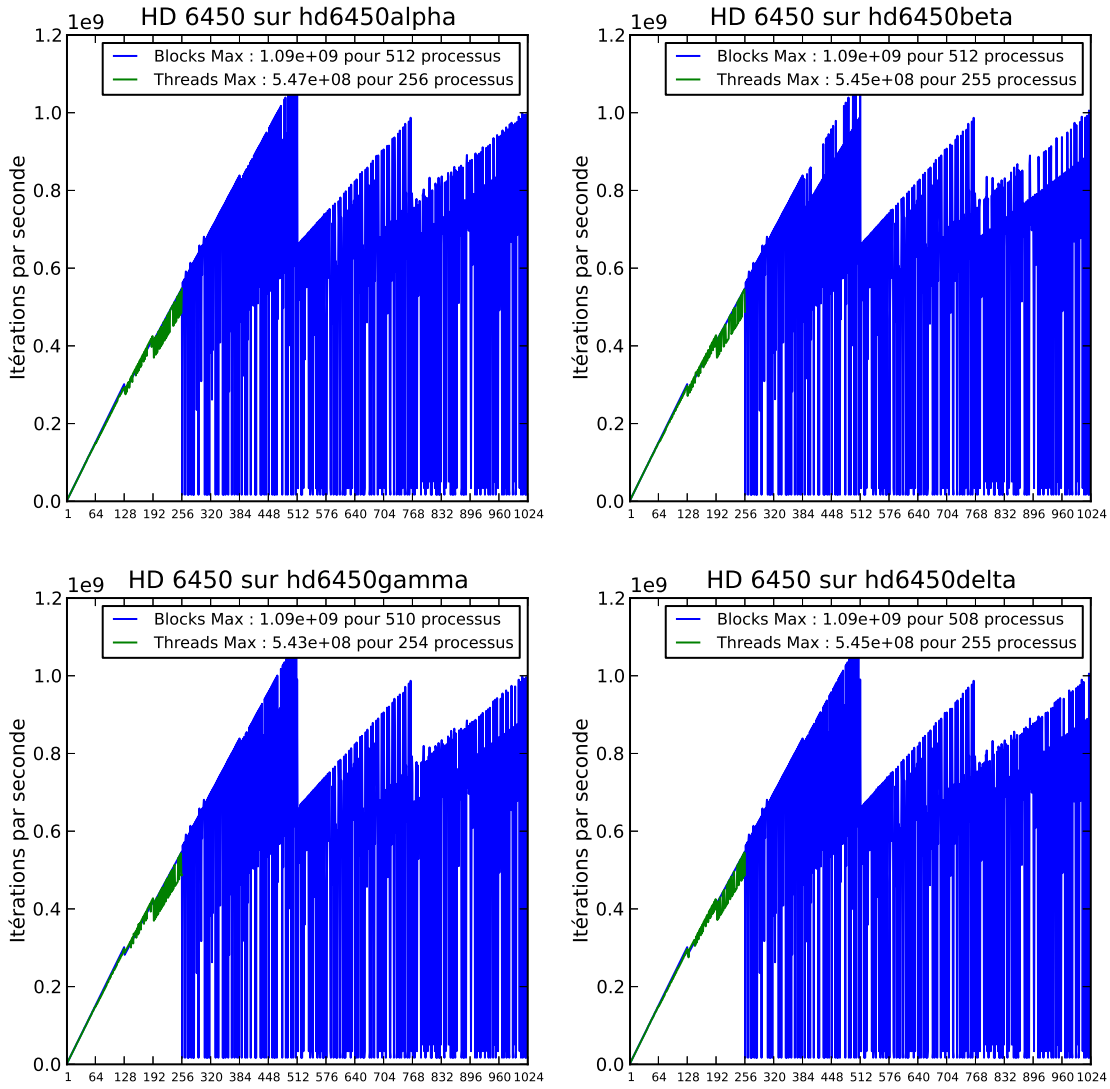

FIGURE 5 – Comparaison de performances (nombre d'itérations par seconde) pour un GPU particulier, une AMD Radeon HD 6450, sur 4 cartes différentes & 4 machines différentes. En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* et *Threads*. Pour chaque nombre de tâches simultanées, 5 expériences.

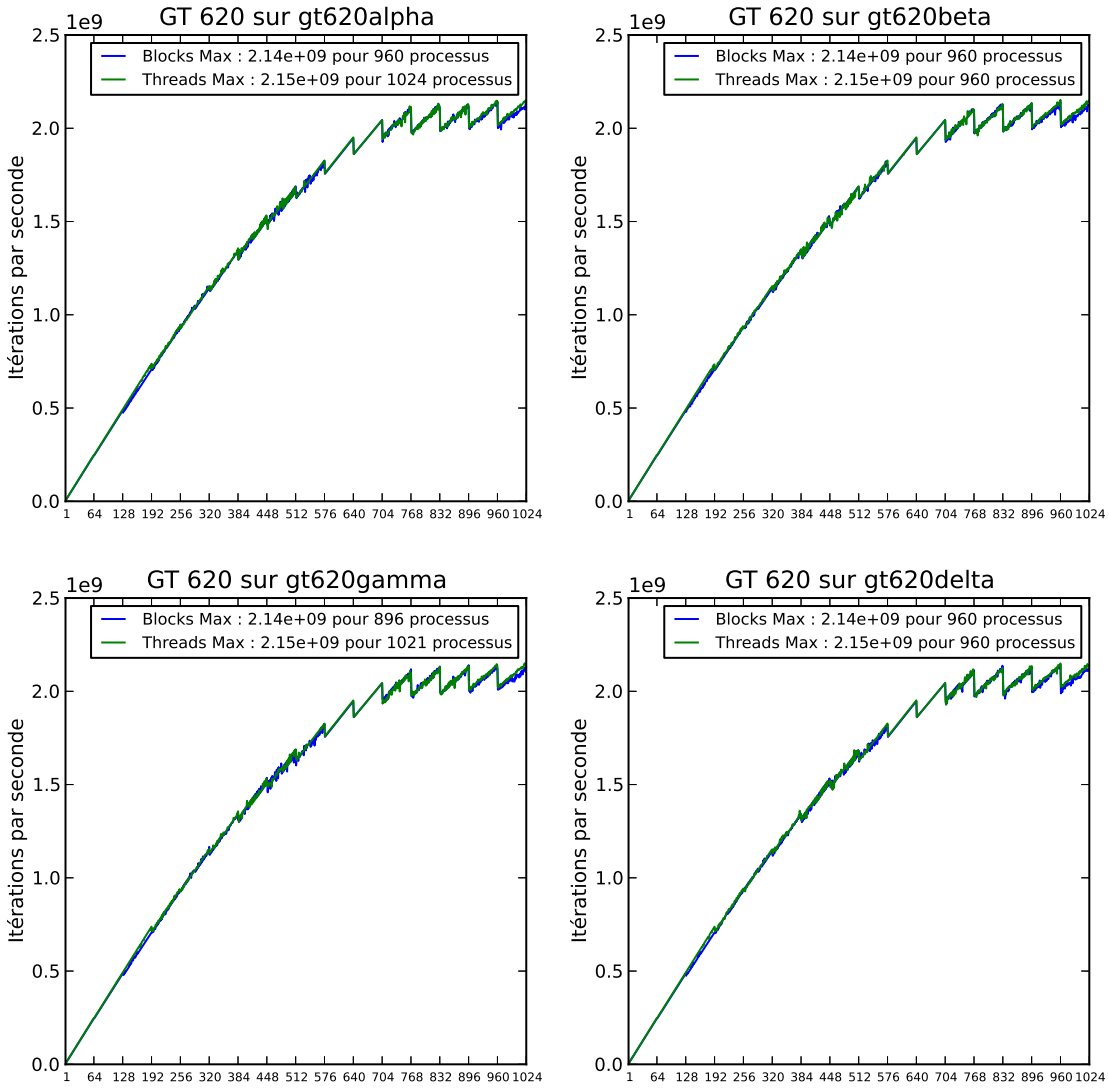

FIGURE 6 – Comparaison de performances (nombre d'itérations par seconde) pour un GPU particulier, une AMD Radeon HD 6450, sur 4 cartes différentes & 4 machines différentes. En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* et *Threads*. Pour chaque nombre de tâches simultanées, 5 expériences.

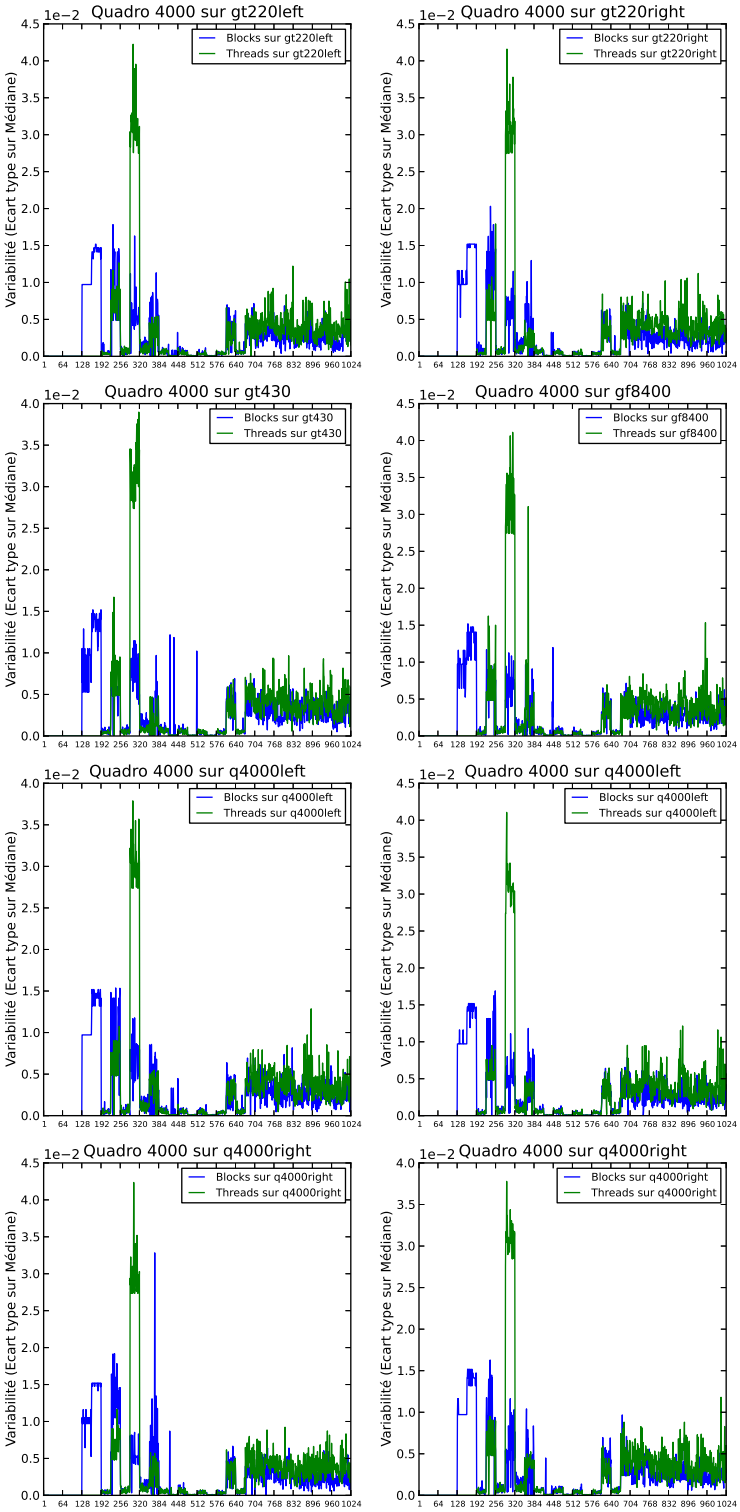

FIGURE 7 – Comparaison de variabilités (rapport entre l'écart-type et la médiane des 5 expériences) pour un GPU particulier, une Nvidia Quadro 4000, sur 8 cartes différentes & 6 machines différentes. En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* et *Threads*.

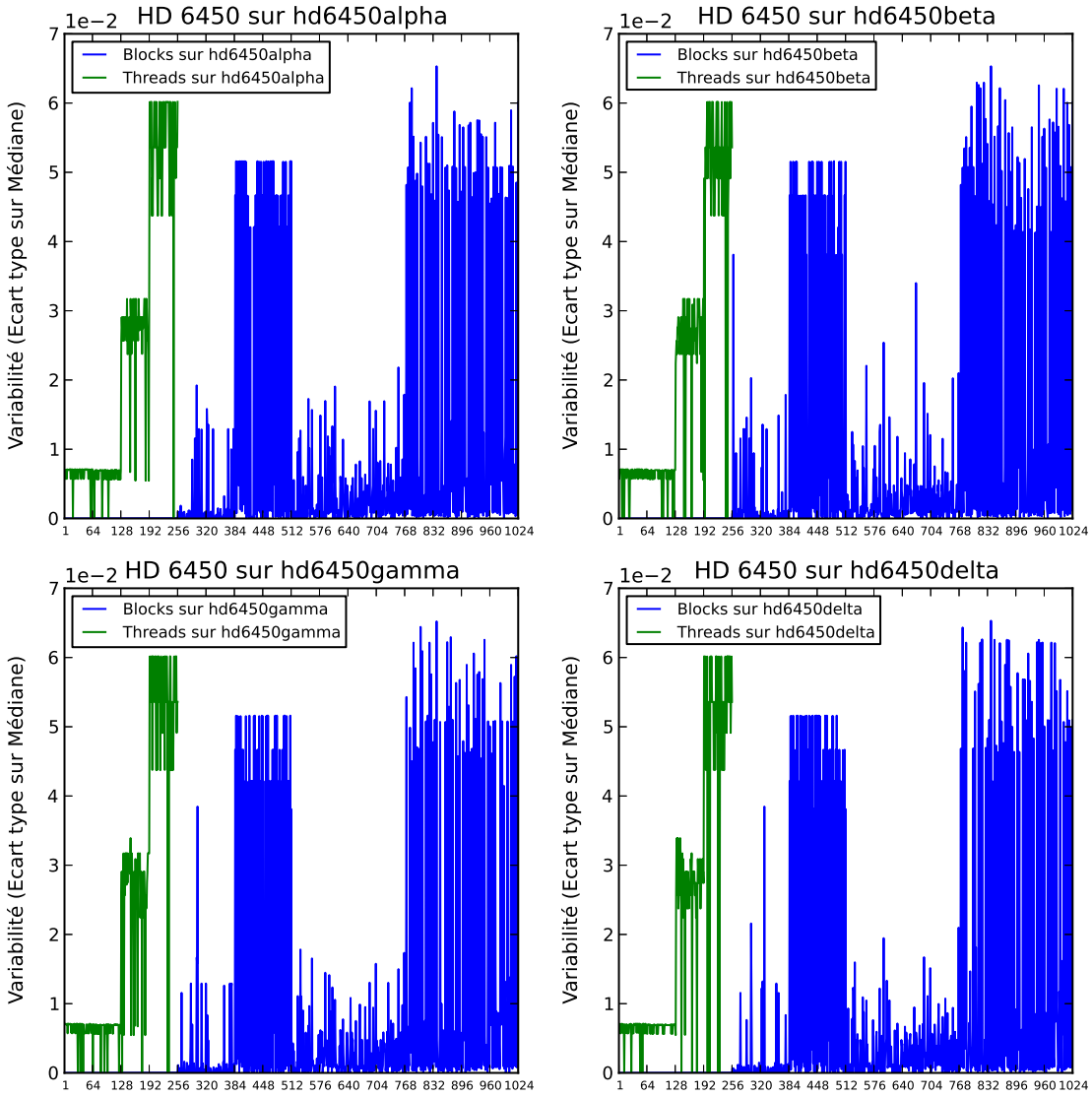

FIGURE 8 – Comparaison de variabilités (rapport entre l'écart-type et la médiane des 5 expériences) pour un GPU particulier, une AMD Radeon HD 6450, sur 4 cartes & 4 machines différentes. En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* et *Threads*.

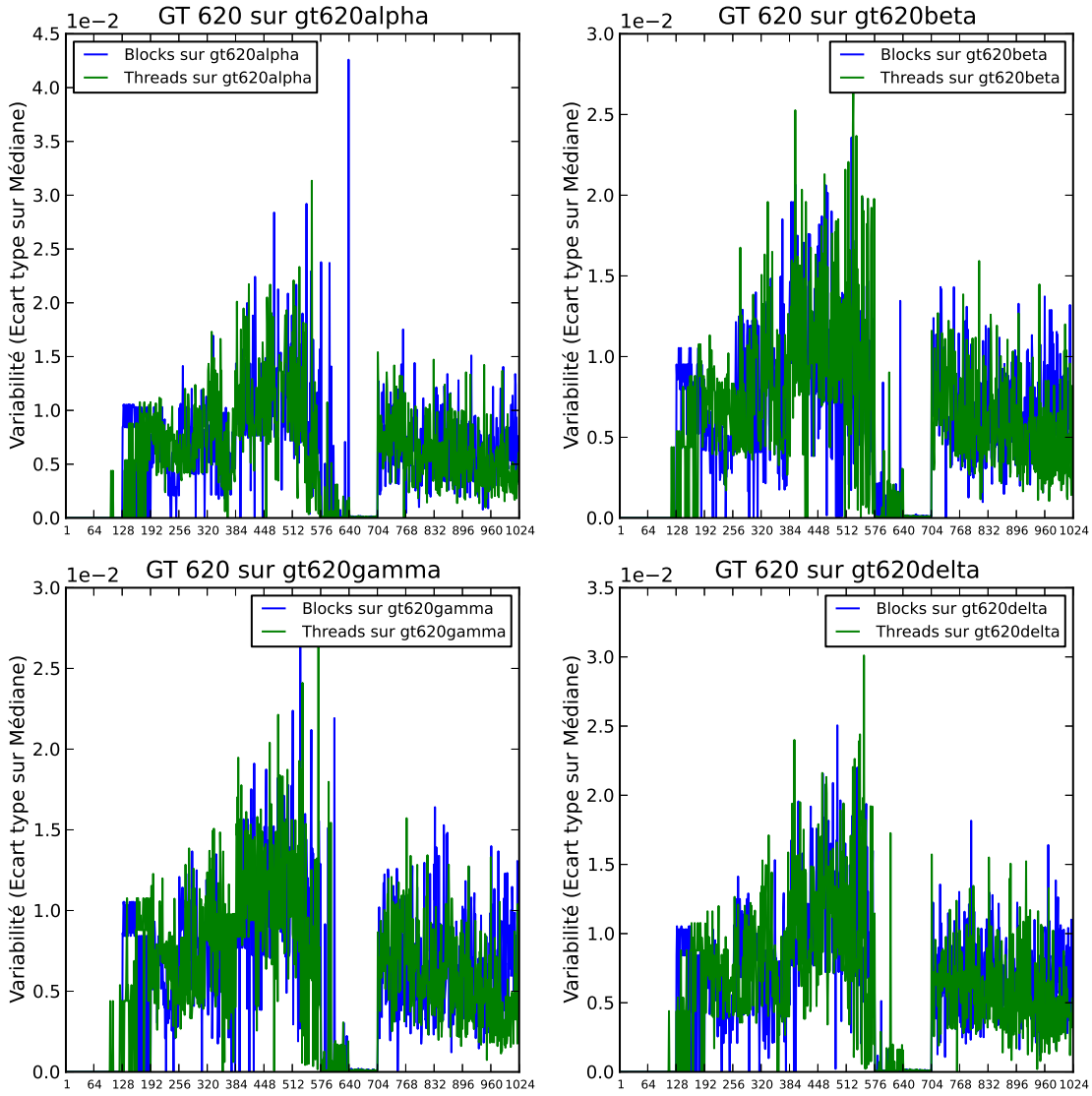

FIGURE 9 – Comparaison de variabilités (rapport entre l'écart-type et la médiane des 5 expériences) pour un GPU particulier, une Nvidia GT 620, sur 4 cartes & 4 machines différentes. En abscisse, le nombre de tâches simultanées, suivants les modes *Blocks* et *Threads*.

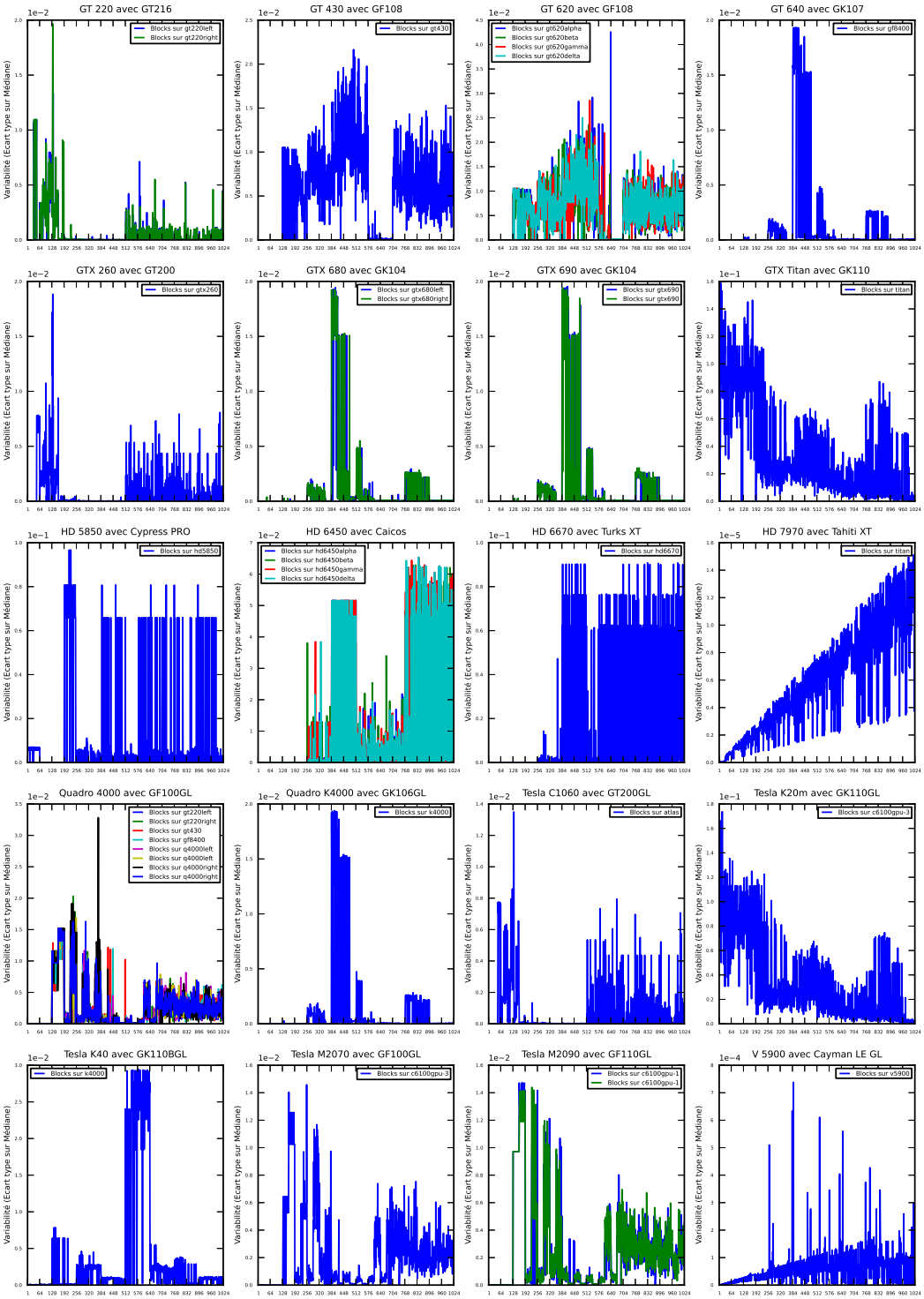

FIGURE 10 – Comparaison de variabilités (rapport entre l'écart-type et la médiane des 5 expériences) pour 20 cartes graphiques différentes. En abscisse, le nombre de tâches simultanées, suivants le mode *Blocks*.

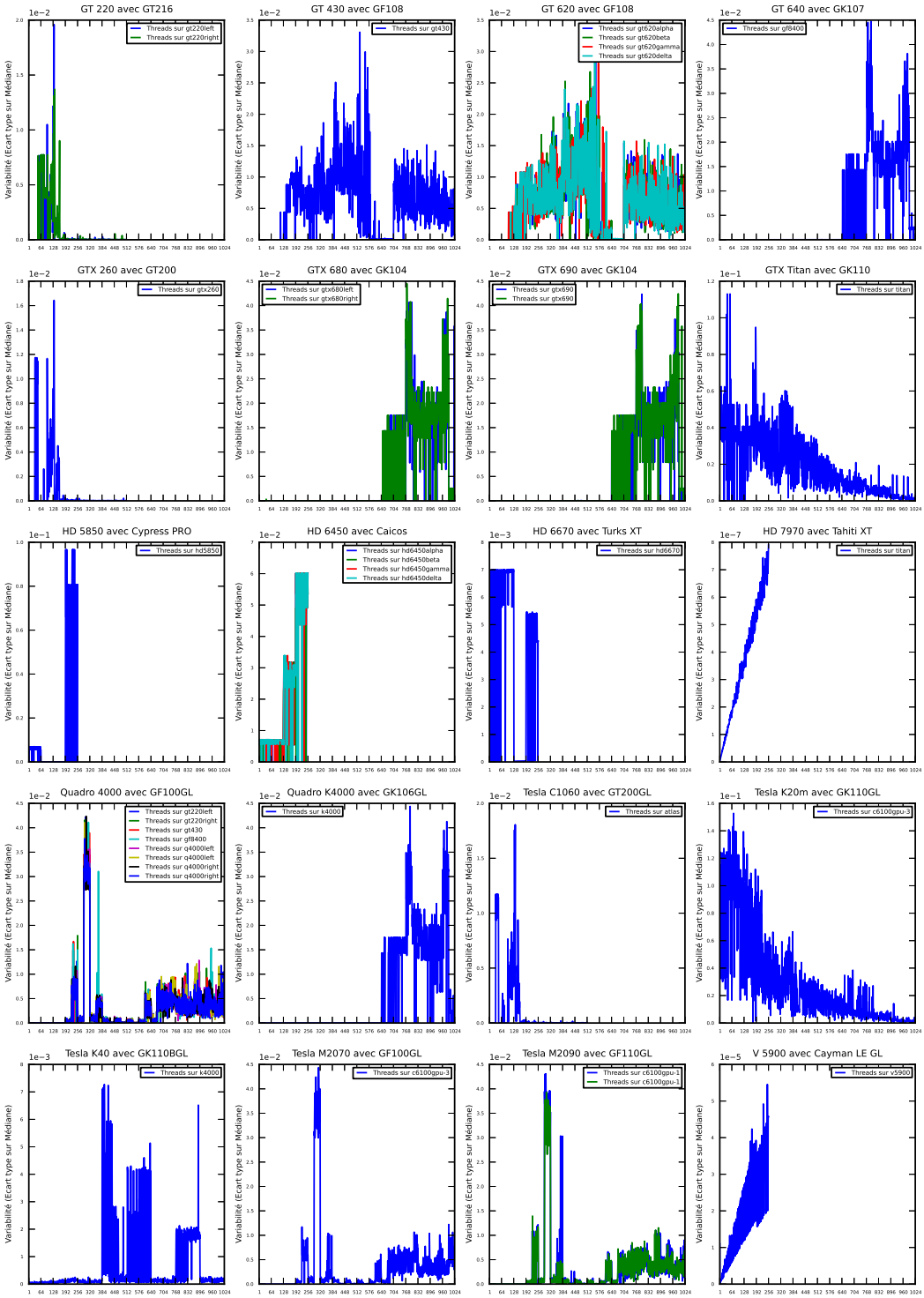

FIGURE 11 – Comparaison de variabilités (rapport entre l'écart-type et la médiane des 5 expériences) pour 20 cartes graphiques différentes. En abscisse, le nombre de tâches simultanées, suivants le mode *Threads*.