Journal of Manufacturing Systems 43 (2017) 296–304

Contents lists available at ScienceDirect

Journal of Manufacturing Systems

journal homepage: www.elsevier.com/locate/jmansys

Voxel model surface offsetting for computer-aided manufacturing using virtualized high-performance computing

Roby Lynn [a,*], Didier Contis [b], Mohammad Hossain [c], Nuodi Huang [d], Tommy Tucker [e], Thomas Kurfess [a]

[a] Georgia Institute of Technology, George W. Woodruff School of Mechanical Engineering, Atlanta, GA, USA
[b] Georgia Institute of Technology, College of Engineering, Atlanta, GA, USA
[c] Georgia Institute of Technology, Department of Computer Science, Atlanta, GA, USA
[d] Shanghai Jiao Tong University, School of Mechanical Engineering, Shanghai, China
[e] Tucker Innovations, Inc., Charlotte, NC, USA

ARTICLE INFO

Article history:
Received 23 September 2016
Received in revised form
12 November 2016
Accepted 16 December 2016
Available online 4 January 2017

Keywords:
Cloud manufacturing
Distributed manufacturing
Computer-aided manufacturing
Computer numerical control
Virtualization
High-performance computing
Voxel
GPGPU
CUDA

ABSTRACT

Curve and surface offsetting is a common operation performed on solid models when planning toolpaths for a machining operation. This operation is usually done in a computer-aided manufacturing (CAM) software package to define the path along which the center of a cutting tool will follow to create a given feature. The CAM software then translates the toolpath created from the offset into G-Code, which is the standard programming language of CNC machine tools. The toolpath planning process can be computationally intensive; thus, a powerful workstation is required to run the CAM software effectively. These standalone workstations can be inconvenient due to their cost and size. Many organizations have been turning to virtualization as an alternative to multiple standalone workstations; virtualization allows for many users to access desktop environments that are hosted from a single remote server. This has the benefit of isolating the user from both OS and hardware requirements for certain software, and also allows them to run the applications they need from anywhere. This research explores the emerging area of virtualized general purpose computation on graphics processing units (GPGPU); this technique is used to support the development of a voxelized CAM package that allows for rapid toolpath generation for complex parts. The surface offset performance is benchmarked on various local and virtualized platforms to evaluate the losses from virtualization. Results indicate a minor loss of performance between virtualized and local GPUs, which is to be expected due to the abstraction of hardware from a virtual machine. Additionally, the development of a GPU-sharing implementation using a server operating system is described and analyzed; results indicate that, as compared to single virtual machines, the GPU-sharing approach demonstrates higher computational efficiency with the addition of compute load to the GPU.

1. Introduction

Machining is a decades-old manufacturing process that is in wide use today because it enables manufacturers to create parts with complex geometry and tight tolerances. Originally, machining was performed using manual equipment where the motion of the cutting tool was controlled with handwheels. Today, modern machining operations are performed using computer-numerical control (CNC) machine tools, where the tool motion is controlled by servomotors. The machine tool is programmed using G-Code, which is a collection of movement commands for the machine tool that describe the sequence of tool motions needed to create a part. The G-Code for the part can be created either manually or through the use of CAM software. For complex parts that require many machine axes, such as 5-axis parts, CAM software is essential to the rapid creation of accurate toolpaths.

CAM software, while incredibly useful, can also be difficult to use properly. The CAM programmer must have machining experience to be able to determine appropriate machining parameters for order-of-operations; additionally, the operation of the software itself requires specialized training [1]. Many manufacturers and other machine tool professionals have expressed the need for

* Corresponding author.
E-mail addresses: roby.lynn@gatech.edu (R. Lynn), didier.contis@coe.gatech.edu (D. Contis), mhossain7@gatech.edu (M. Hossain), huangnuodi@126.com (N. Huang), tommy@tuckerinnovations.com (T. Tucker), kurfess@gatech.edu (T. Kurfess).

http://dx.doi.org/10.1016/j.jmsy.2016.12.005
0278-6125/

easier-to-use CAM software [2]. Additionally, the machining simulations provided by current CAM software are not able to accurately represent gouging and toolmarks that are left behind once machining is complete. A better approach is needed that is both easier to use and provides more realistic simulation of volume removal during the machining process to provide better feedback to the CAM programmer.

Typical CAM software relies on analytical models to describe part geometry, such as a boundary representation (B-rep). While these models are compact and easy to store in memory, their complexity is limited by the precision of the computer that is used to process them. Additionally, limitations exist in the modification of the part surfaces when simulating material removal during the machining process [3]. To overcome these limitations, this work proposes the use of a voxel-based part model where the entire part is discretized into small cubes. Similar to a two-dimensional image, which is made up of small pixels, the voxel model is made up of small cubes called voxels. The structure of the voxelized part model, known as a Hybrid Dynamic Tree (HDT), allows for simplified computations of Boolean volume subtraction during the simulation of the machining process, which can allow for more accurate visualization and feedback to the CAM operator [4,5]. The voxel model, however, requires a different strategy than an analytical model to process efficiently. Because the voxel model is effectively a large array, different operations on the array can be performed simultaneously to accelerate the speed at which the model is processed. These operations can be performed on a parallel computing platform, such as a graphics processing unit (GPU). The GPU implementation of the HDT has been shown to greatly outperform a comparable implementation that relied on a single CPU thread [6].

While the use of the voxel model allows for distinct advantages over traditional analytical models, the speed at which the model can be processed is dependent upon the performance of the parallel computing platform that is employed. For this implementation, NVIDIA graphics cards with compute unified device architecture (CUDA) functionality were employed. CUDA is an application programming interface (API) that allows for access to the GPU for computational instead of graphics purposes. Many modern computers are equipped with discrete GPUs, which allows for use of the software without additional hardware. However, many mobile devices, including laptops and tablets, do not have discrete GPUs that can be used for computation. For users without access to a GPU compute device, virtualization can be employed to allow them to access a computer with GPGPU capability.

Virtualization is the implementation of a simulated desktop on a computer or server. This is accomplished by abstracting the physical hardware of the machine. In the most basic sense, virtualization can be used to run an instance of a guest operating system (OS) on top of an already running host OS; for example, a user could start a Linux session using virtualization software inside Windows to allow them to simulate a Linux computer even though the native OS is Windows. Virtualization is frequently used to host multiple desktops on a single server; this allows for many users to access the same hardware and each have an individual desktop. Powerful hardware can be installed in the server, and users can access it from any internet connected device. Thus, the clients do not need physical access to a GPU compute device, as the hardware can be virtualized from the remote server. While the virtualization of GPUs for graphics rendering is commonplace, virtualization for GPGPU is less widespread. Commercial virtualization solutions support this functionality to a limited extent; however, there is a deficiency of solutions that allow for multiple users to share a single GPU for compute tasks. The present work explores a time-sharing approach to GPU virtualization that allows for many simultaneous users to perform toolpath planning on a single GPU.

The remainder of this paper is organized as follows: first, background information in presented on the current states of both the virtualization of engineering software and the use of GPU-acceleration in engineering computation; second, the state of virtualized GPUs for computation is discussed. Next, a user-friendly voxelized CAM package called SculptPrint is introduced. SculptPrint relies on the GPU to process a voxel model and allows for very accurate simulation of the machining process. Further technical details about SculptPrint and the voxel model are discussed. Next, the experimental setup to evaluate the performance of SculptPrint on a virtualized platform is discussed and results are presented. Discussions and conclusions evaluate the practicality, advantages and limitations of the virtualized implementation.

2. Related Work

Cloud manufacturing is the idea of leveraging cloud computing resources to facilitate manufacturing; this can be done through distributed computing, virtualization, etc. [7]. Multiple researchers have shown that virtualization is an important tool for realizing higher security and more widespread deployment of large applications in the manufacturing environment [8,9]. Virtualization allows users to remotely access high-end HPC (high performance computing) hardware from any device and leverage the power of that hardware without being physically close to it [10]. This is an example of infrastructure-as-a-service (IaaS), where the actual computing hardware is provided to users for them to run computations on. Typical engineering software, such as computer-aided design (CAD), computer-aided engineering (CAE) and CAM packages require hardware acceleration for optimal performance. This usually necessitates a powerful CPU, large amounts of RAM and a discrete GPU. The GPU is the most critical component for realizing high-performance parallel computation on a single machine [11]. To remove these hardware requirements from the client machine, engineering software can be virtualized and presented to the user on a virtual desktop.

2.1. Virtualization of engineering software

Hardware-accelerated virtual desktops have been used for CAD/CAM software virtualization in large organizations previously [12–15]. Recent improvements in display protocols, such as Citrix ICA, Microsoft RemoteFX and Teradici PCoIP, combined with hypervisor improvements and the emergence of graphics virtualization technologies, have supported the adoption of VDI (virtual desktop infrastructure) for CAD/CAM workload [16]. Virtualization allows for members of an organization to utilize hardware-accelerated software from thin clients or personal machines without powerful hardware; it also removes operating system restrictions imposed by certain software, as the client only duplicates the display for the user and the software itself is run on the virtual machine. Application security and ease of troubleshooting can be increased through the use of a private cloud that has a controlled user base. The implementation of virtualized desktops requires a hypervisor, for example Microsoft Hyper-V, Citrix XenServer or VMWare vSphere. The hypervisor is responsible for dividing the physical resources of the server amongst the VM sessions. The hypervisor requires computational resources to run, and thus performance of virtual machines is not identical to that of standalone machines [17]. While this can be a problem, more powerful server hardware can be selected to mitigate performance losses. Additionally, some virtualization solutions, such as Citrix XenServer, grant VMs direct access to hardware resources [18], which bypasses parts of the abstraction layer that causes slowdowns in traditional virtualization. When applied to GPUs, this is known as PCI-passthrough: the PCI device is

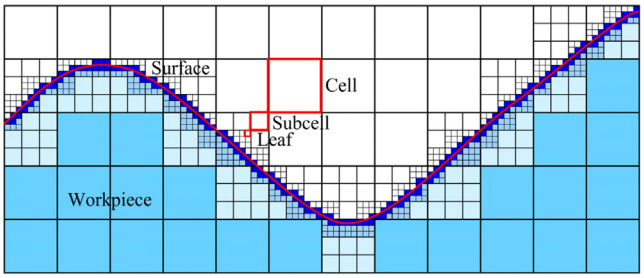

Fig. 1. Surface Representation in SculptPrint by Voxels.

directly passed through to a VM, which is granted exclusive access to the device. While this may be beneficial in the interest of VM performance, it also limits the number of simultaneous CUDA users to the number of physical GPUs on the machine.

2.2. SculptPrint: voxelized computer-aided manufacturing

This work explores the use of a new computer-aided manufacturing (CAM) software package on virtual desktops. This CAM package, known as SculptPrint, relies on a voxel model to represent a part. A voxel model is a digital representation of a part in which the model is broken into voxels, the three-dimensional equivalent of pixels. This is in contrast to an analytical mode, such as B-rep, which describes the part surfaces as functions of parameters [4,9,20]. This voxel model is stored in a unique data structure, called the HDT, that is a combination of a grid and an octree [21]. The voxelized part construction removes the need to fit curvature to the part geometry, which allows for representation of highly complex shapes that are only dependent upon the voxel size. Fig. 1 shows how the voxelized part representation is constructed to describe part geometry. The voxel division is adjusted dynamically to allow for more efficient storage: voxels inside of the part volume (shown in light blue) are consolidated into larger groups, while those on the surface (shown in dark blue) are as small as possible to give a high-resolution surface. The dynamic resolution of the HDT, which can be attributed to the octree structure of the voxels residing inside the part (i.e., not on the surface), allows for more efficient storage of the data structure; however, it also necessitates the use of different algorithms for processing that data than would be used for simple grid data structures [11].

SculptPrint provides a high degree of automation in the toolpath creation process, as rotational axis positions on 4- and 5-axis machines are determined automatically with an accessibility algorithm [19]. This algorithm frees the NC programmer from needing to perform manual collision checking and allows for true simultaneous 5-axis interpolation. Additionally, the use of the voxel model allows for the machining of complex surfaces and the accurate reproduction of textures from an input model. Fig. 2 shows both the rough and finish machining of an intricate aluminum part where the textures on the chest of the model must be reproduced accurately [22]. The image on the left shows the voxel model as shown in SculptPrint, and the image on the right shows the part after machining. Fig. 3 demonstrates the G-Code generation process for the previously shown part [22]. The simulation of the toolpath is shown in the left image, and the resulting G-Code to make the path is shown on the right. This interactive simulation of the program that will be sent to the machine tool allows the programmer to easily visualize the machining process at each line. SculptPrint has the ability to create G-Code to machine complex geometry on various machine tool configurations; it can create code for 2-axis turning, 3- and 5-axis milling, and 4-axis millturning. The G-Code output of SculptPrint can be used to machine a large variety of materials, provided that the appropriate cutting speeds and feeds are set.

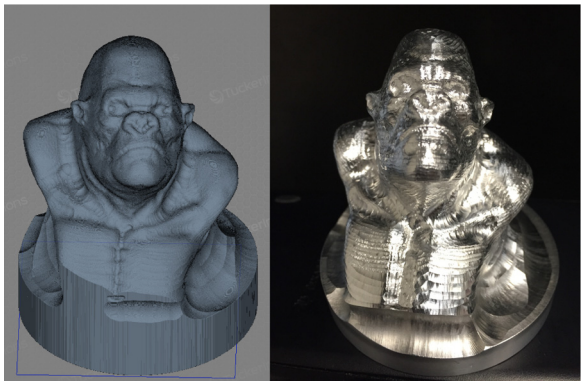

Fig. 2. Reproduction of Complex Surfaces and Textures.

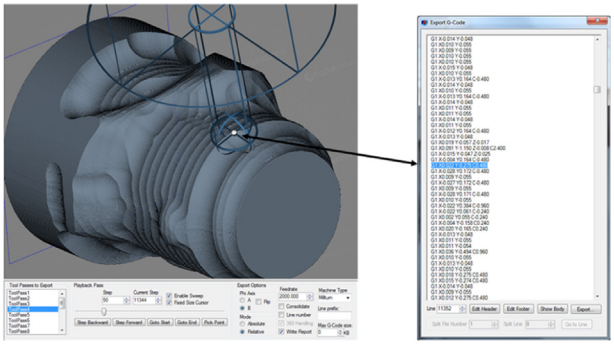

Fig. 3. G-Code Generation in SculptPrint.

Plastic parts are generally preferred for running programs for the first time, but metal parts with equally complex geometry can also be machined.

Efficient processing of the HDT is accomplished through parallelization, which is implemented on a GPU. While GPUs are present in many modern computers, some are not designed for general purpose computation (GPGPU). To allow a wide user base to access SculptPrint without needing a GPU capable of computation, this work proposes virtualization of a group of GPUs that can be used to serve virtual desktops to users.

2.3. HPC-acceleration of engineering software using GPUs

GPGPU has been used in the acceleration of engineering software previously; it is commonly applied to tasks in Finite Element Analysis (FEA) and can introduce significant computation time gains. Parts of the FEA process are inherently parallelizable, as they rely on solving large systems of linear equations and the assembly and manipulation of large stiffness matrices [23]. While some parts of the process are still carried out serially (i.e. using CPUs instead of GPUs), acceleration of some parts of the computation can still provide performance increases. The adoption of GPGPU for FEA acceleration has enjoyed a great deal of success, and is currently offered as functionality for commercially available FEA packages [24]. One of the major limitations of GPU-acceleration for FEA is the usual requirement that the GPU must be a physical resource as opposed to a virtual one. This implies that running GPGPU-accelerated FEA on a virtual machine has not been widely implemented. While this is not a technical limit due to FEA itself, the implementation of GPGPU on virtualized platforms requires further investigation

2.4. Virtualization of GPUs for HPC

The use of general purpose computation on graphics processing units is an emerging area that allows for the execution of parallelized algorithms on graphics hardware. Traditional virtualization solutions do not support the use of GPU computation on a VM, although newer approaches are beginning to provide this functionality. Shi, Chen and Sun developed an early implementation of virtualized GPGPU, known as vCUDA, that intercepted CUDA calls from a guest and forwarded them to a physical GPU [25]. This was the first proof that VMs can be accelerated with GPUs, and numerous other studies have followed. Vinaya, Vydyanathan and Gajjar demonstrated quantitative analysis of the advantages and limitations of three distinct CUDA-enabled virtualization technologies: rCUDA, gVirtuS and Xen PCI-passthrough [26]. This work showed that Xen PCI-passthrough provides the best performance of the three technologies surveyed, but it requires that an entire GPU be dedicated to each VM. Gupta, et al. presented a front-end virtualization solution, GViM, which allows multiple VMs to access the same physical GPU [27]. GViM's front-end virtualization solution is different from that of PCI-passthrough in that each VM does not have direct access to an accelerator, instead forwards GPU calls through a management layer. Gottschlag, et al. implemented a low-overhead approach to GPU virtualization, LoGV, which relies on an extension to the hypervisor and a guest kernel driver [28]. Xiao et al. demonstrated VOCL, a virtualization framework that is based on OpenCL and allows for GPU sharing [29].

The previously mentioned approaches may be difficult to implement for organizations with existing virtualization capabilities because they rely on additional software to handle CUDA call forwarding, they cannot support multiple VMs on a single GPU, or they require a form of Linux to function. The work described in this paper involves a Windows-based implementation of a virtualization system that allows for multiple VMs to share a single GPU. As a result, no additional software is required for implementation and Windows applications run natively on the guest OS. This approach relies on a terminal server operating system, which allows multiple users to use the same VM simultaneously. This allows for easier license management than the typical single-user-per-VM approach; the GPU-accelerated software only needs to be installed on the terminal server to be made accessible to all users, rather than installing multiple copies of the same software on different VMs. Additionally, it allows a single GPU to be used by multiple users for simultaneous compute tasks. This work extends performance evaluation of virtualized HPC into the realm of manufacturing. Benchmarking that directly evaluates the performance of a computer-aided manufacturing application can prove to be useful as the manufacturing industry continues to adopt virtualization.

3. Experimental procedure

3.1. Contact volume generation

One of the essential operations in the generation of a machining toolpath is the creation of a surface offset, which is the creation of a surface that is expanded or shrunk but a constant amount in the normal direction from a starting surface. Surface offsetting is used in SculptPrint for the generation of a contact volume, which defines the surface along which the center point of a ball endmill can reside without cutting away a target part volume. As an example, consider the part shown in Fig. 4 [30]. The image on the left shows the target voxel model of the part after it has been completely machined. The image on the right shows an example of this part that was machined using G-Code generated from SculptPrint. The resulting contact volume that was created for this part is shown in the left

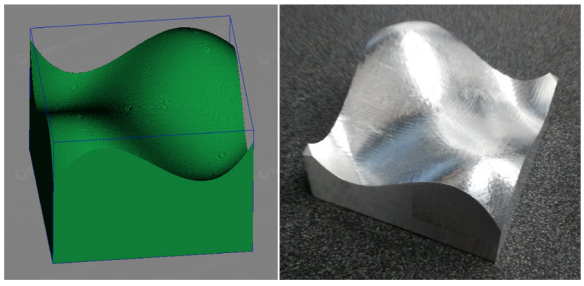

Fig. 4. Target Volume and Finished Part.

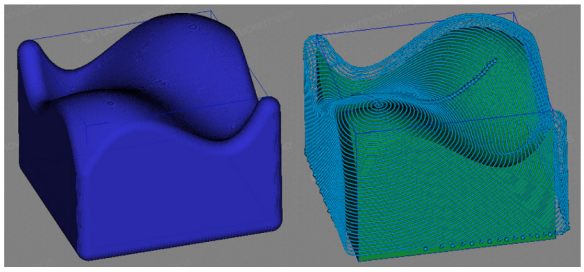

Fig. 5. Contact Volume and Resulting Toolpath for 1/4" Ball Endmill.

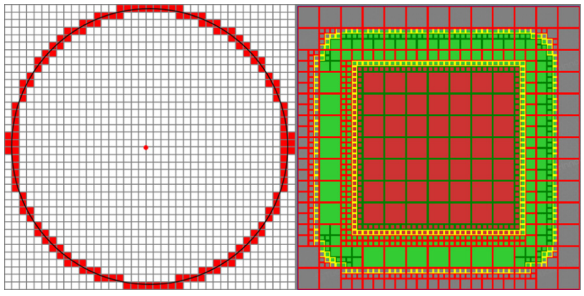

Fig. 6. Ring Structuring Element and Resulting Offset.

image of Fig. 5. This contact volume was created by offsetting the part surface by the radius of the tool used to machine the finishing pass; in this case, a 0.250" diameter (6.35 mm) ball endmill was used. Thus, the total amount of surface offset was 3.175 mm. The right image of Fig. 5 shows an isoscallop toolpath that was created using the contact volume. The light blue lines define the path of the center of the spherical tip of cutting tool along the part surface.

Offsetting the surface of a voxel model is accomplished in SculptPrint using an algorithm developed by Hossain, et al. [31]. The operation is analogous to two-dimensional image dilation, but it is performed in 3D with voxels instead of pixels. In the simple two-dimensional case, a dilation is performed by sweeping a structuring element along the boundary of a curve to be offset. For example, consider both the structuring element and the resulting dilated curve in Fig. 6 [31]. The structuring element is a discretized ring, and the input curve is a square. By placing the structuring element at each pixel of the input curve and activating all of the pixels of the structuring element that lie outside of the input curve, a dilated square is obtained. The boundary of the red cells in Fig. 6 is the input curve and the green cells are the newly activated pixels after the dilation. Performing this operation in three dimensions follows an analogous procedure as was just described in two dimensions; however, in three dimensions, the structuring element is a sphere and the input curve is a three dimensional surface that describes the boundary of the part to be machined. By sweeping the spherical structuring element along each voxel of the input surface and dilating, the offset surface is obtained. The newly offset part vol-

Fig. 7. GT Virtual Lab Cluster and NVIDIA GRID K1.

ume is known as the contact volume. Note that for this study, a spherical structuring element was used because a ball endmill will be employed during the machining process.

As it is effectively matrix manipulation, the process of offsetting is highly parallelizable. The time taken by the offsetting operation is not affected by the number of machine axes required to machine the part, but it is affected by offset distance and the total number of voxels comprising the part surface. The speed of this operation is directly dependent upon the performance of the GPU that is executing the CUDA offset kernel; the total time taken by the kernel to complete the operation can be timed and compared on various platforms to compute relative performance gains or losses.

3.2. Vlab: Georgia Tech's Virtual Laboratory

Today, Georgia Tech has a large infrastructure designed to support its virtual desktop farm (Vlab) accessible at any time from anywhere in the world and from any device. GPU-enabled servers were introduced in 2012 to support the use of graphics intensive applications by students. Currently, Windows 7 VMs with virtual GPUs and published shared desktop (Windows 2012R2) with GPU passthrough are accessible to students. University students can perform design and analysis for course and research work from anywhere by offloading computation and licensing requirements to remote machines. The CUDA computations for the virtual machines used in this work were supported by an NIVIDIA GRID K1; this card was chosen because it was purpose built for virtualization and has been successful in previous VM deployments. A photo of GT's Vlab farm is shown in Fig. 7, in addition to a photo of the NVIDIA GRID K1 card that was used for this implementation [30].

3.3. Virtual machine design and configuration

The experimental setup for this implementation consists of a Dell R720 with two Intel E5-2660 Sandy Bridge CPUs; each of these CPUs has eight cores and runs on a 2.2 GHz clock. The machine is equipped with a total of 192GB of memory, four Intel X520 10GB network adapters and one NVIDIA GRID K1. The GRID K1 used for the experimental setup is shown installed in the R720 in Fig. 8 [30]. The hypervisor that was chosen for this implementation is Citrix XenServer 6.5 SP1.

Two virtual machines (VMs) were created to serve this implementation: one VM was setup with Windows 7, and one was setup with Windows Server 2012 R2. Each VM was given 42GB of memory. Additionally, each of the VMs were assigned 2 virtual CPUs with four cores per CPU. The VM memory configuration from the Citrix administrative dashboard is shown in Fig. 9. Both the Windows 7 and the Windows Server 2012 R2 virtual machines were tested with the same CUDA operation to compare the performance on each of them.

The configuration of the Windows Server 2012 R2 VM allowed for multiple session hosting on a single virtual machine. As a result, more than one simultaneous user could run CUDA jobs on a single GPU. The Windows 7 VM was simply used as a comparison

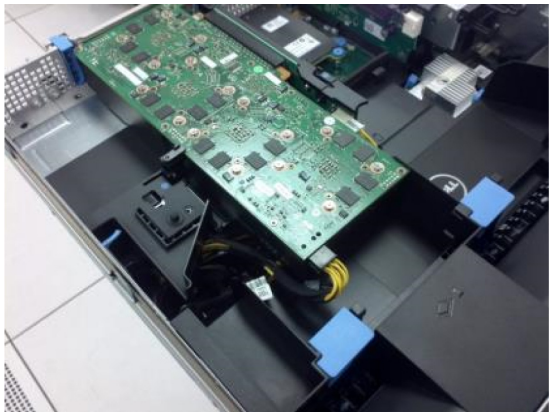

Fig. 8. GRID K1 Installed in Dell R720.

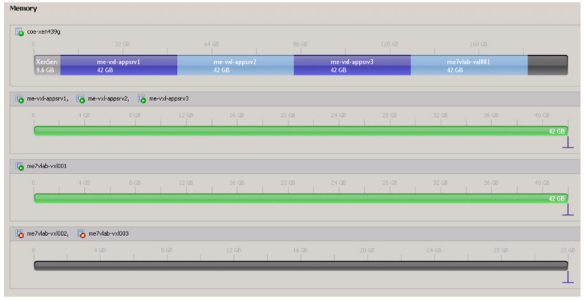

Fig. 9. Virtual Machine Configuration.

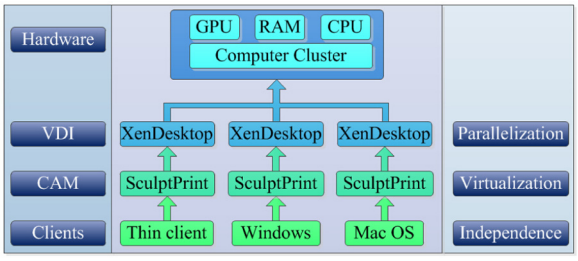

Fig. 10. Illustration of SculptPrint Deployment on Citrix XenDesktop.

between the two operating systems; it is otherwise not useful for this implementation because it can only support one user.

The limitation of one user per VM when using PCI passthrough discussed earlier has been solved through the use of scheduling in Windows 2012 R2 which allows users to share the CPUs and memory. Windows 2012 R2 supports a terminal server architecture which lets multiple users connect at the same time and have their own sessions displayed remotely (via RDP or Citrix ICA); in contrast, Windows 7, 8 and 10 only support one user session. It is important to note that the terminal server solution is used to achieve high user density and lower cost per user; this may not be the best approach when absolute performance is desired. For the use case where performance is key, virtual workstations with resource reservation (e.g. the Windows 7 VM) are ideal because it is possible to reserve specific resources for each VM.

SculptPrint was installed on both the Windows 7 and Windows Server 2012 R2 machines so that it was accessible to remote users. This allowed for performance comparisons between the two virtualized platforms, in addition to comparison with local platforms. An overview of the SculptPrint virtualization hierarchy as deployed on Citrix XenDesktop VMs is shown in Fig. 10 [30]. It should be noted that, when using the terminal server approach, each session

of SculptPrint launched by a user does not correspond to a separate installation of the software; rather, it is merely another instance that is managed by the server operating system.

3.4. Offset performance benchmarking on various GPUs

The goal of this work is to produce quantitative data that demonstrates the performance of the offsetting procedure on both local and virtual platforms. To this end, the same offset procedure was performed on a variety of platforms equipped with different GPUs. An overview of the tested graphics hardware is presented in Table 1 [32–35]. The benchmark platform on which the rest of the results were compared was the GRID K1. Note that the specifications listed for the K1 refer to the entire device, which doesn't necessarily correspond to what physical hardware was allocated to the VMs. In this implementation, because GPU passthrough was used and each VM was assigned a single GPU, the total number of CUDA cores should be divided by four to reflect the number of cores that was actually allocated to a particular VM. Specifically, both the Window 7 and Windows Server 2012 R2 VMs were given 192 CUDA cores each.

The first comparison platform that was chosen was the NVIDIA Quadro K620M, which is a professional mobile card that is designed for light usage and has somewhat similar specifications to the GRID K1 when it is divided to serve four VMs. The second comparison platform chosen was the Quadro 6000, which is a legacy high-end professional card; the Quadro 6000 offers similar performance to current mid-level professional cards. The third comparison platform chosen was the Quadro M5000, which is a current high-end professional card. This card was expected to provide the fastest offset performance of all the platforms tested. CUDA performance is directly dependent upon the number of cores that a GPU has, so significant performance differences are expected across the four platforms tested. In this case, higher performance translates into less time taken to perform operations necessary in SculptPrint for toolpath planning. A reduction in total time required to create a toolpath would result in quicker progression from design to finished part.

4. Results

4.1. Local offset performance

A comparison of the offset time for the various GPUs listed in Table 1 was used as a baseline to compare the predicted performance variations caused by different GPU hardware. Due to the computational overhead that is inherent in virtualization, overall performance is expected to be slower on a VM as compared to a local machine. However, passing through the GPU to the VM is expected to provide performance that is comparable to that of a local machine [26].

A direct comparison of different graphics hardware can be accomplished by computation of the maximum theoretical number of floating point operations per second (FLOPS). This gives an indication of the expected performance of a GPU and is accomplished using Eq. (1),

$$\text{TFLOPS} = \frac{N_{\text{FMA}} M_{\text{CUDA}} F_{\text{GPU}}}{P_{\text{GPU}}} \times 10^{-9} \tag{1}$$

where M_{CUDA} is the total number of CUDA cores on the GPU device (the number of CUDA cores per VM in the case of the K1), F_{GPU} is the GPU clock frequency, P_{GPU} is the number of physical GPUs on the GPU device and N_{FMA} is the number of floating point operations that each CUDA core can perform per clock cycle. Due to architectural differences between these cards, N_{FMA} values are different between some of the cards that were compared. Each core can perform 2 FLOP per cycle for the K1, K620M and M5000 and 4 FLOP/cycle for the Q6000.

The kernel time taken to perform the 3.175 mm offset shown in Fig. 5 was timed for each of the four platforms. A comparison of both the theoretical FLOPS and the measured kernel time for the offsetting operation is shown in Table 2. The offset was performed on each platform four times, and the average time taken and resulting sample standard deviation are shown in the table.

Multiplication of the kernel time by the theoretical FLOPS gives a performance criterion, the normalized kernel time with 1 TFLOPS, which can be used to directly compare graphics hardware without regard for differences in clock speed, memory size, etc. Calculation of the normalized kernel time for the i^{th} GPU was accomplished using Eq. (2),

$$T_{\text{N}i,\text{Kernel}} = \text{TFLOPS} \times T_{i,\text{Kernel}} \tag{2}$$

where $T_{\text{N}i,\text{Kernel}}$ is the normalized kernel time for the i^{th} GPU and T_{Kernel} is the total kernel computation time for the i^{th} GPU. The results show that the GRID card performs less efficiently than the Q6000 and the M5000, but more efficiently than the laptop K620M. These results are consistent with previous work on this project using a larger offset [30]. The percentage efficiency gain is calculated according to Eq. (3),

$$\text{Gain} = \left[1 - \frac{\bar{T}_{\text{N}i,\text{Kernel}}}{\bar{T}_{\text{NK1},\text{Kernel}}}\right] \times 100\% \tag{3}$$

where $\bar{T}_{\text{N}i,\text{Kernel}}$ is the normalized average kernel computation time for the comparison card being evaluated (either the K620M, the Q6000 or the M5000) and $\bar{T}_{\text{NK1},\text{Kernel}}$ is the normalized average kernel computation time for the K1 card. The averages were taken for the four trials that were performed on each card. This calculation provides an indicator of the computational efficiency of each card, which is expected to be negatively affected by virtualization.

Some of the performance disparities can be attributed to slowdowns caused by the hypervisor layer. As expected, the performance differences are significant; the high-end Quadro M5000 took less than 10% of the kernel time that the K1 did for the offset operation. This does not come as a surprise, however, as the M5000 is a newer architecture and has both more CUDA cores and memory then the K1 does for each VM. Perhaps a more meaningful comparison to make would be between the normalized kernel times; this comparison shows interesting efficiency differences between the K1 benchmark and the other three comparison platforms. The highest-performance card tested, the M5000, demonstrated a 14.21% higher computational efficiency than the K1; on the lower end, the K620M performed the worst of all, with an average efficiency loss of 11.71%. This can be attributed to the fact that the K620M is a mobile card and was not designed for HPC.

4.2. Virtualized offset performance on windows server 2012R2

Windows Server 2012 R2 allows for the hosting of multiple sessions through Microsoft's Remote Desktop Protocol (RDP) on the same hardware from a single machine [36]. This approach has the advantage of bypassing the single-user-per-GPU limitation that was encountered when using Windows 7 VMs. However, the hosting of multiple sessions necessitates abstraction of the GPU to the user, so each VM running on the server OS does not have direct physical access to the GPU as it did when using Windows 7. As a result, GPU resources are shared between sessions on the machine and computation per user takes longer as more sessions are added.

Results from kernel time measurements on the Windows Server 2012 R2 platform are shown in Table 3. The kernel time result from one session on Windows Server 2012 R2 can be directly compared with the kernel time reported for the GRID K1 in Table 2, which was

Table 1
Card Specifications.

Card Type	Number of GPUs	GPU Clock (MHz)	Total Memory (GB)	Total CUDA Cores	CUDA Cores per VM
GRID K1	4	850	16	768	192
Quadro K620M	1	1029	2	384	N/A
Quadro 6000	1	574	6	448	N/A
Quadro M5000	1	861	8	2048	N/A

Table 2
Local Kernel Computation Results.

Card Type	Maximum TFLOPS	Mean Kernel Computation Time (s)	Mean Normalized Kernel Time with 1 TFLOPS (s)	Mean Gain (%)
GRID K1	0.3264	352.894 ± 0.054	115.184 ± 0.018	N/A
Quadro K620M	0.7903	162.820 ± 4.778	128.677 ± 3.776	−11.71
Quadro 6000	1.028	102.324 ± 0.004	105.190 ± 0.004	8.68
Quadro M5000	3.527	28.017 ± 1.147	98.815 ± 4.045	14.21

Table 3
Virtualized Kernel Computation Results.

Number of Sessions	Theoretical TFLOPS per Session	Mean Total Kernel Time (s)	Mean Kernel Time per Session (s)
1	0.3264	352.667 ± 0.031	352.667 ± 0.031
2	0.1632	594.155 ± 1.758	297.078 ± 0.879
3	0.1088	793.202 ± 16.356	264.40 ± 5.452
4	0.0816	938.824 ± 29.454	234.71 ± 7.364

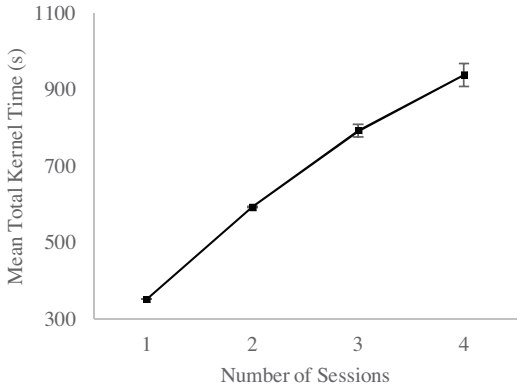

Fig. 11. Kernel Time with Multiple VMs.

measured on a VM running Windows 7. Comparison of these results shows that, as expected, offset performance on Windows 7 is comparable to that on Windows Server 2012 R2 when the entire GPU is dedicated to a single user. The remainder of the data in Table 3 shows average kernel time results for the multi-session implementation when more than one user shares a single GPU. These results were generated by performing the same 3.175 mm offset shown previously on multiple sessions simultaneously. The experiment was performed four times on each platform. For the trials with more than one session, a set of results was generated for each session; for example, the trial with four sessions generated a total of sixteen sets of data. The average kernel time reported in the table is the average of all data sets for a given configuration.

It is interesting to note that the average kernel time per VM actually decreases as more sessions are added. Intuitively, it would seem that the total average kernel time would be linearly related to the number of sessions; however, these data suggest that this is not the case. Fig. 11 shows a plot of the average kernel time per session as a function of the number of sessions. The decreasing slope of this curve is indicative of the fact that kernel time per session decreases as the number of sessions increases. The error bars indicate the sample standard deviations for each trial. The nonlinear trend observed in Fig. 11 can be attributed to the fact that a single kernel dispatch to the GPU may not utilize all of the available resources on the GPU.

Table 4
Virtualized Compute Efficiency.

Number of Sessions	Theoretical TFLOPS per Session	Mean Normalized Kernel Time per Session (s)	Mean Gain (%)
1	0.3264	115.124 ± 0.010	
2	0.1632	96.966 ± 0.287	15.76
3	0.1088	86.300 ± 1.780	25.03
4	0.0816	75.424 ± 1.375	34.48

Only a certain number of streaming multiprocessors (SMs) on the GPU are required for a given kernel; if the kernel does not make use of all of the SMs, some may sit idle. Dispatching multiple kernels to the GPU simultaneously allows for higher utilization of the device, which leads to a higher overall efficiency.

4.3. Multi-session efficiency gains

The decreasing slope of kernel time as a function of the number of sessions indicates that computational efficiency of the terminal server implementation increases as more sessions are added. Table 4 shows the mean normalized kernel time per session as a function of the number of sessions, in addition to the gain in computational efficiency that was calculated using Eq. (3), with the single session kernel time of 115.124 ± 0.010 s as the benchmark. To calculate the normalized kernel time, the theoretical TFLOPS figure was divided by the number of simultaneous sessions to give the theoretical TFLOPS per session. This result is encouraging, and shows that a heavier computational load on the K1 is ideal for highest efficiency.

5. Discussion

The large amount of kernel time taken by the K1 may suggest that virtualization of SculptPrint is not a feasible solution, but that must be weighed against two factors. First, the GRID K1 is an older card (released 2013) and does not represent the current state of GPU technology; second, the added benefit of allowing multiple users to run SculptPrint on one machine may indeed provide time and cost savings that offset the performance losses. Regardless of

the performance differences between these GPUs, these data show that implementation of SculptPrint using virtualized GPGPU is possible; however, more powerful server graphics hardware should be investigated to increase the speed of toolpath production.

The virtualization of a SculptPrint will prove to be useful in both academic and industrial settings; it allows for the utilization of the software for manufacturing education and production applications [22,37]. The use of one GPU to serve multiple sessions can lift some restrictions of the typical PCI-passthrough approach. The terminal server implementation investigated here has the potential to encounter problems with graphics driver timeouts; for instance, if many users are attempting to run a time-consuming CUDA job simultaneously, the operating system could forcefully terminate the driver if the response time exceeds a certain threshold [38]. This functionality, called TDR (timeout detection and recovery), can be managed by correctly setting the timeout threshold in the registry. Additionally, TCC (Tesla Compute Cluster) mode can be used to override the TDRs that Windows enforces by default.

Another area of exploration is the implementation of virtual GPU (vGPU) on Citrix XenDesktop; this would allow for multiple sessions of Windows 7 to share a single GPU. CUDA is not currently supported for vGPU in a meaningful way; it is possible when using an 8 GB frame buffer on a Tesla M60, but this is not useful because the M60 only has two GPUs and a total of 16GB of memory. The vGPU approach, however, could present problems of noisy neighbors, where the compute requests of some users take more time than anticipated; this scenario will cause graphics slowdowns for other users due to the time-sharing of the GPU. The vGPU approach could allow for higher performing VMs as it would not rely on a server operating system to manage GPU resources; however, further research is needed to determine is this is true. Similar to the terminal server approach described here, the a vGPU implementation would not require dedicating an entire GPU to a single user.

Given the rapid advancement of GPU computational capability, there is no doubt that SculptPrint's novel approach to voxelized CAM will become more practical over virtualization as time progresses. New hardware is constantly under development, which will bring performance improvements and allow for more cost-effective implementation of virtualized GPGPU. Cost savings in the VM implementation come from the fact that only the GPU needs to be upgraded in the server instead of buying new workstations for SculptPrint users every few years. As a concrete example, the newly-released NVIDIA Tesla M60 card is designed for virtualization applications and is capable of 7.6 TFLOPS [39]; at a price of approximately $5000, it has a significantly higher performance to price ratio than the GRID K1, which produces 1.3 TFLOPS for approximately $3000. If the M60 were split to serve four users as the K1 was, each would receive a theoretical 1.9 TFLOPS, which outperforms the mid-range Quadro 6000 tested in this work. There is no doubt that, as graphics hardware continues to evolve, the price per TFLOP will decrease and virtualized GPGPU for CAM will become more attractive. The ease of deployment of HPC-accelerated CAM software to a large user base is an exciting prospect, and could allow for many to quickly and easily create toolpaths for complex geometry.

6. Conclusion

A virtualized implementation of a novel CAM system has been developed and benchmarked. Implementation was performed at Georgia Tech on a Dell PowerEdge R720 machine using an NVIDIA GRID K1 card for GPGPU. Citrix XenDesktop was used to deliver the CAM system to the user on virtual desktops running either Windows 7 or Windows Server 2012 R2. Results for an offsetting operation performed on a Windows 7 VM were compared to those from three different local machines with dedicated graphics hardware. GPU performance and computation efficiency were compared amongst the four platforms, and it was shown that the GPU used for virtualized HPC performed less efficiently than dedicated workstation GPUs. The same offset operation was performed on Windows Server 2012 R2 with various numbers of simultaneous sessions and the results were compared to those measured on Windows 7. Performance figures suggest a nonlinear relationship with decreasing slope between the number of sessions and the total time taken to perform a surface offset. This is promising, as it was expected that the offset time would be linearly related to the number of sessions in the best case scenario. Areas for continued work were presented and analyzed for viability.

Acknowledgments

This work was supported by NSF CMMI grants 1547093 and 1329742. Special thanks to the College of Engineering Information Technology Department at Georgia Tech for support with the VM implementation.

References

[1] Warkentin A, Hoskins P, Ismail F, Bedi S. Computer-aided 5-axis machining. In: Systems techniques and computational methods. CRC, Press, Inc.; 2001. p. 3001–34.
[2] Machine Tool Professionals, Outlook on CNC Machine Investments. Centrifuge Brand Marketing Research, 2010.
[3] Touron B, Marcheix D, Gueorguieva S. A Note on Non-Manifold Object Sweeping. US: Springer; 1997. p. 116–25.
[4] Tarbutton JA, Kurfess TR, Tucker TM. Graphics based path planning for multi-Axis machine tools. Comput. Aided. Des. Appl 2010;7(6).
[5] Jang D, Kim K, Jung J. Voxel-based virtual multi-axis machining. Int J Adv Manuf Technol 2000;16(10):709–13.
[6] Hossain MM, Tucker TM, Kurfess TR, Vuduc RW. Hybrid dynamic trees for extreme-resolution 3D sparse data modeling. Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium 2016.
[7] Wu D, Rosen DW, Wang L, Schaefer D. Cloud-based design and manufacturing: a new paradigm in digital manufacturing and design innovation. Comput Des 2015;59:1–14.
[8] Helo P, Suorsa M, Hao Y, Anussornnitisarn P. Toward a cloud-based manufacturing execution system for distributed manufacturing. Comput Ind 2014;65(4):646–56.
[9] Buckholtz B, Ragai I, Wang L. Cloud manufacturing: current trends and future implementations. J Manuf Sci Eng 2015;137(4):40902.
[10] Morariu O, Borangiu T, Raileanu S. vMES: Virtualization aware manufacturing execution system. Comput Ind 2015;67:27–37.
[11] Hossain MM, Tucker TM, Kurfess TR, Vuduc RW. A GPU-parallel construction of volumetric tree. In: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms. 2015. p. 1–4.
[12] Devoir F, Siggers R. Exploring design considerations: CAD/CAM experience from the experts using citrix and VMWare. NVIDIA GPU Technology Conference 2015.
[13] NVIDIA and Citrix: Graphics-Accelerated Virtual Desktops and Applications. NVIDIA Whitepaper, 2014.
[14] Empowering World-Class Marine Designers. NVIDIA Case Study, 2014.
[15] Elevating the Design and Manufacture of World-Class Helicopters. NVIDIA Case Study, 2015.
[16] Poppelgaard T, RDP, RemoteFX, ICA/HDX, EOP and PCoIP – VDI Remoting Protocols Turned Inside Out. 2010.
[17] Li J, Wang Q, Jayasinghe D, Park J, Zhu T, Pu C, Performance Overhead Among Three Hypervisors: An Experimental Study using Hadoop Benchmarks.
[18] Citrix XenServer® 6.5 SP1 Documentation: Administrator's Guide.
[19] Konobrytskyi D. Automated CNC Tool Path Planning and Machining Simulation on Highly Parallel Computing Architectures. Clemson University; 2013.
[20] Tarbutton JA, Kurfess TR, Tucker T, Konobrytskyi D. Gouge-free voxel-based machining for parallel processors. Int J Adv Manuf Technol 2013;69(9).
[21] T. M. Tucker, T. Kurfess, D. Konobrytskyi, Hybrid Dynamic Tree Data Structure and Accessibility Mapping for Computer Numerical Controlled Machining Path Planning. 2015.
[22] Lynn R, Jablokow KW, Reddy N, Saldana C, Tucker T, Simpson TW, et al. Using rapid manufacturability analysis tools to enhance design-for-manufacturing training in engineering education. ASME 2016 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2016) 2016.
[23] Georgescu S, Chow P, Okuda H. GPU acceleration for FEM-based structural analysis. Arch Comput Methods Eng 2013;20(2):111–21.

[24] Crivelli L, Dunbar M. Evolving use of GPU for dassault systemes simulation products. NVIDIA GPU Tech Conference 2012.
[25] Shi L, Chen H, Sun J. vCUDA: GPU accelerated high performance computing in virtual machines. 2009 IEEE International Symposium on Parallel & Distributed Processing 2009:1–11.
[26] Vinaya MS, Vydyanathan N, Gajjar M. An evaluation of CUDA-enabled virtualization solutions. 2012 2nd IEEE International Conference on Parallel Distributed and Grid Computing 2012:621–6.
[27] Gupta V, Gavrilovska A, Schwan K, Kharche H, Tolia N, Talwar V, et al. GViM: GPU-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing. 2009. p. 17–24.
[28] Gottschlag M, Hillenbrand M, Kehne J, Stoess J, Bellosa F. LoGV: low-Overhead GPGPU virtualization. 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC) 2013:1721–6.
[29] Xiao S, Balaji P, Zhu Q, Thakur R, Coghlan S, Lin H, Wen G, Hong J, Feng W-C, VOCL: an Optimized Environment for Transparent Virtualization of Graphics Processing Units.
[30] Lynn R, Contis D, Hossain M, Huang N, Tucker T, Kurfess T. Extending access to HPC manufacturability feedback software through hardware-accelerated virtualized workstations. International Symposium on Flexible Automation (ISFA 2016) 2016.
[31] Hossain MM, Nath C, Tucker TM, Vuduc RW, Kurfess TR. A graphical approach for freeform surface offsetting with GPU acceleration for subtractive 3D printing. In: 11th Manufacturing Science and Engineering Conference (MSEC). 2016.
[32] NVIDIA GRID K1 and K2. NVIDIA Datasheet, 2013.
[33] NVIDIA Quadro K620M. NVIDIA Datasheet, 2014.
[34] NVIDIA Quadro M5000. PNY Datasheet, 2015.
[35] NVIDIA Quadro 6000. NVIDIA Datasheet, 2010.
[36] Kouril J, Lambertova P, Performance analysis and comparison of virtualization protocols, RDP and PCoIP 2 Analysis of the requirements for virtualization system, vol. II.
[37] Lynn R, Saldana C, Kurfess T, Kantareddy SNR, Simpson T, Jablokow K, et al. Toward rapid manufacturability analysis tools for engineering design education. In: 43rd SME North American Manufacturing Research Conference (NAMRC). 2016.
[38] Timeout Detection and Recovery (TDR). NVIDIA Nsight Visual Studio Edition 4.0 User Guide, 2014.
[39] NVIDIA Tesla M60. PNY Datasheet, 2014.